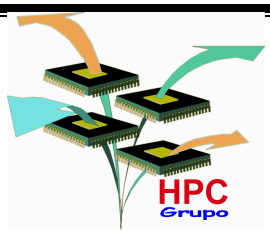

In-depth analysis of x86 instruction set condition codes influence on superscalar execution

Technical Report TR-UAH-AUT-GAP-2006-23-en

Virginia Escuder, Raúl Durán, Rafael Rico

Department of Computer Engineering, Universidad de Alcalá, Spain

March 2006

Versión en español:

Estudio detallado del impacto de los códigos de condición del repertorio x86 sobre la ejecución superescalar

Informe técnico TR-UAH-AUT-GAP-2006-23-es

Virginia Escuder, Raúl Durán, Rafael Rico

Departamento de Automática, Universidad de Alcalá, España

Abstract:

Instruction set design is a crucial aspect of computer architecture. The requirements to fulfill have evolved along time. For superscalar processing the most important feature is to avoid code coupling caused by data dependencies. However, instruction sets may have particular characteristics that produce a negative impact into the amount of available parallelism for which it is important to analyze them.

The popular x86 instruction set architecture includes some of those characteristics that may have negative effects in superscalar processing that may influence the final performance: dedicated use of registers, implicit operands, complex effective address computation mechanisms, condition codes usage, etc. It is therefore, an ideal candidate to use for evaluation purposes. Specifically, we analyze the impact produced by condition codes.

In this work we take two approaches to solve the problem. On the first one we perform a statistical analysis of the utilization of instructions and operands. On the second we perform a mathematical analysis based on graph theory that provides a quantification for the contribution due to condition codes to the overall coupling according to the different dependence types.

Finally, we evaluate the influence of condition codes utilization into the microperation level, proposing some solutions to achieve an improvement in performance.

Key words: instruction set architecture, instruction level parallelism, instruction trace, operands use, instructions use, graph theory.

1. Introduction

Instruction set design has always been a fundamental issue in computer science. In the popular text from Hennessy and Patterson [12], the reader may find an historical review of this subject that emphasizes this fact. Lately, commercial developments as EPIC [21] and some experimental projects such as "*EDGE Instruction Set*" [7] show that there is still a considerable interest in the matter.

Design criteria for building instruction sets has evolve in time: memory protection support, addressing modes closer to that of high-level languages, code size compactness, simplification of the architecture, conditional execution, etc.; some theoretical studies [6, 16] have had outcomes resulting on many other requirements as well.

However, evaluation of instruction sets architecture has not been explored as much as it could be expected, given the importance of the subject. Lunde's article [15] about the "*Empirical Evaluation of Some Features of Instruction Set Processor Architectures*" back in 1977 introduces an evaluation of the influence of instruction set's architectural aspects such the number of registers in machine performance In our opinion, this is paradigmatic and although the article was written long ago, we intend to rescue the idea behind it and propose the analysis of instruction sets architectures emphasizing that it has a definitive influence in the final performance.

In-depth analysis of the impact that instruction sets on their own have on performance has been abandoned in favor of considering a single unit for study: the instructions set and the hardware that should interpret it, under the assumption that this is a sounder computational approach. Another circumstance that has also contributed to the lack of research in this type of analysis is the extensive use (sometimes abuse) of simulation as the performance evaluation method. Simulation does not differentiate between the effects produced by the language itself and the physical resources used for the measure [27].

In this work, we outline the importance of analyzing instruction sets solely, without the contribution from any other factor that may also have an influence in the overall performance. We believe that a separate, theoretical evaluation of Instruction Set's Architectures is necessary and can benefit the design of compilers as well as the design of the physical layer of computers, in addition to maybe provide an additional criteria for its taxonomy.

Nowadays, one of the most important objectives in processor design is code decoupling, that is, avoid data dependency among instructions in order to obtain maximum concurrency in superscalar processing of code. Performance in the field of superscalar execution depends on many factors: the algorithm's intrinsic parallelism, the capabilities of the high level language used, the compilation process and the target machine's instruction set. It is therefore important to unlink the study from the physical layer and focus into the machine language layer by itself. Particularly, the instruction set layer can be responsible for an over-ordering of the code that has no solution in the physical layer or that may cause increased execution complexity and power consumption. Instruction sets have limitations such as dedicated use of some registers, implicit operands utilization, complex address computation, condition codes utilization, etc., that may introduce negative effects into the amount of available parallelism.

These are the factors we pretend to analyze and measure in order to state what are the desired features of Instruction sets for optimal superscalar execution, how and why an instruction sequence gets tangled in data dependences, etc. However, most research in instruction sets evaluation is limited to the analysis of instruction utilization like [8] in *VAX* or [1] in x86. Only a few evaluate data distribution [13].

2. The x86 instruction set

The x86 Instruction Set Architecture (ISA) was designed to fulfill basically two objectives: to decrease the semantic gap between the high level and the machine languages and to obtain a compact executable code. These criteria are now obsolete but the instruction set has been maintained for binary compatibility reasons. Nevertheless, it behaves inefficiently in superscalar implementations. The x86 ISA shows many features that may compromise the actual concurrency of the original computational task like dedicated use of registers, condition code dependent branching and effective address computation where up to three registers may be involved. The sources of potential code coupling have been identified out of the distribution of data use in programs [20].

These negative features may cause data dependences not present in the original computation, resulting on an over-ordering of instructions which is produced before it is submitted to execution. Consequently, instructions are rendered to the physical layer under more restrictive conditions for parallel execution than could be expected from the original computation task and so we think it is a penalty imposed by the machine language layer.

The x86 ISA performs poorly in superscalar environments compared to non-x86 sets for different architectural proposals. The IPC (Instructions Per Cycle) is 0.5 to 3.5 in different x86 execution models [17, 23]; compared to an IPC of 2.5 to 15 (and peaks of 30) of non-x86 processors [24, 25, 26]. That makes us think that the architecture of the instruction set is a limiting factor on its own for the available parallelism at the instruction level layer.

As the x86 ISA includes the features limiting parallelism mentioned above and because of its spread usage (it has been maintained across time thanks to binary compatibility) we think it is a good candidate to apply the analytical methods we propose. Among the most interesting aspects to consider we have the impact produced by the use of condition codes and the impact produced by the computation of effective addresses on the amount of available instruction level parallelism.

Condition codes dependences is one of the factors generating code coupling and it has been

recently estimated [20] that it is responsible for close to 13% of parallelism loss. The estimation is made by completely eliminating the influence from condition codes in a testbench, so it serves as a reference for an ideal upper limit in the speed-up that could be reached in case that the use of condition codes could be completely avoided in the code. This is the start point of a more extensive analysis that should also reveal what type of data dependences are produced in instructions, what programming features tend to increase this influence, or how much coupling needs the actual computation task. And consequently, what are the possible alternatives to improve superscalar performance.

3. Condition codes in instruction set's architectures

Using condition codes is an alternative for implementing conditional control flow. The evaluation of the branch condition is performed using one or more condition-code bits. These bits are grouped, for practical reasons, into a status register where they get updated upon the execution of processing instructions; setting or unsetting each individual bit, the collection completely describes the result. A processing instruction typically precedes a conditional branch and therefore it creates a dependence which requires serial execution. Architectures using this schema are called status register architectures; the x86 is one of them. Considering superscalar execution, condition codes increment the ordering of instructions as they pass information from one instruction to the next.

Theoretically, there are other two alternatives for implementing conditional control flow: evaluation of the contents of a register named in the branch instruction against a criteria also contained in the branch instruction, and atomization of the comparison and branching actions into a single instruction. The first alternative, commercially adopted in the *Alpha* and *MIPS* architectures for instance, is simple and also optimal for superscalar execution while the second one, used in the *PA-RISC* and *VAX* processors, makes the pipeline design more complex as it results from the union of two operations in one.

Advantages and drawbacks of the three approaches are clearly exposed in the classical book from Hennessy and Patterson [12].

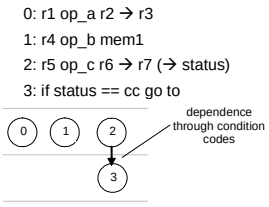

Fig. 1. Condition codes impact on parallelism in a typical basic block.

To better understand the condition codes impact in parallelism we focus the analysis in the basic block structure. Figure 1 is an example where a basic block is shown together with its corresponding condition code's data dependence graph, that is, the graph depicting the contribution from condition codes in the dependence relation. Instruction 2 generates a true dependence with instruction 3, (a Read After Write dependence). True dependences have computational meaning and require serial execution: the block needs 2 computing steps to execute in a superscalar environment. This example shows how using condition codes intrinsically decreases the parallelism in a basic block.

Let's now analyze Fig. 2 where we have another very similar instructions basic block where there are two processing instructions instead of only one. Both of them write into the status register, so the data dependence graph describes that three computing steps are necessary to process the block thus revealing a lesser degree of available parallelism in the block. Comparing with Fig. 1, the true dependence remains, however, the new data dependence is an output dependence (Write After Write) and it is due to the limitation of resources for writing: both instructions need a single physical resource to write information. The typical solution to the problem in a superscalar environment is to use register renaming techniques implemented in hardware, which eliminates the imposed serialization. It is important to note that this dependence is imposed by the architecture of the instruction set and it is not a real (computational) dependence.

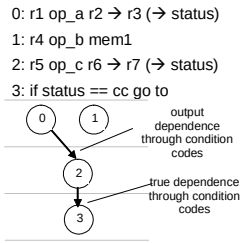

Fig. 2. Condition codes impact on parallelism in a typical basic block with two status writes.

The *PowerPC* is another status register architecture but, in contrast to the x86, its instruction set was designed to avoid the negative effect produced by Output dependences due to condition codes: data processing instructions format include a bit used to indicate whether the condition bits must be updated or not. This effectively limits the coupling produced by condition codes to the cases where it really has computational meaning, and the compiler is in charge of driving the decision. Figure 3 shows the result of the usage of this mechanism. Now the graph only shows a true dependence arc, similar to the initial basic block example and so the block may be processed into less number of computing steps than it is in Fig. 2.

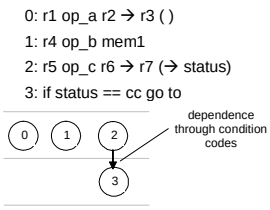

Fig. 3. Condition codes impact on parallelism in a typical basic block with two processing operations but just a solely status write.

The x86 ISA has not solved the output dependences problem because it needs to keep backward binary compatibility. Consequently, it generates data dependences with no computational meaning causing negative effects when executing in superscalar environments, complicates compiling processing and demand an execution-time solution in hardware that would imply additional cost, power consumption, and IC area.

4. Condition codes in the ISAx86

Condition codes are used for conditional branching and they are located into the status register. This register hosts bits with different meaning that can be classified into one of the following two groups:

- control flags
- status flags.

Control flags include miscellaneous information related to the operation modes of the processor, including step mode execution, interrupt masking information, use of auto-increment memory pointers, etc. These control flags do not contain computational information and therefore are not taken into account in our analysis. Status flags qualify the result of processing operations. A single or a combination of status flags correspond to what we generally refer to as a condition code. Status register thus has this dual consideration being a unique storage location while each bit has its own independent meaning and management procedure.

Condition codes are typically used for conditional branching in status register architectures but in the case of the x86 ISA status flags can also be used as input operands for some operations like rotation through the carry flag, BCD adjusting operations, or extension of operand's sizes longer than a word. In these particular cases, where information flows from one operation to the next, it exists a true dependence holding computational meaning and the instructions involved cannot be executed independently thus causing a performance degradation. It is necessary therefore to include these cases into the analysis and evaluate its influence into the general code coupling.

To start the analysis, we classify instructions into five groups according to the type of manipulation they perform on the status flags (O: overflow, S: sign, Z: zero, A: Auxiliary Carry, P: parity, C: Carry) of the status register.

Group I												
	Read						Write					
Transfer	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
LAHF		X	X	X	X	X						
POPF							X	X	X	X	X	X
PUSHF	X	X	X	X	X	X						
SAHF								X	X	X	X	X
INT	X	X	X	X	X	X						
IRET							X	X	X	X	X	X

Table 1. Transfer instructions accessing condition codes.

Group I:

Table 1 shows the first group of instructions accessing condition codes. These are data movement instructions whose purpose is to copy condition codes into the accumulator or to the top of the stack and viceversa. We include software interrupt routine calls and returns too because these instructions save and restore the status register onto/from the stack.

Group II:

The second group is made up of processing instructions using condition codes as an extra input operand to perform some data transformation. Table 2 shows these instructions specifying if the access is for reading, writing or both and further classifies the instructions into three classes:

- Adjust: used in BCD representation for adjustments (in ASCII and decimal),
- Add/Sub for extended arithmetic that require double word operands,
- Rotations through the carry flag.

The first two types of instructions are rather infrequent: BCD representation is rarely used directly and the word size of current general purpose machines is large enough for the integer arithmetic of most programs commonly used.

Group II												
	Read						Write					
Adjust	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
AAA				X						X		X
AAD									X	X		X
AAM								X	X		X	
AAS				X						X		X
DAA				X		X	X	X	X	X	X	X
DAS				X		X		X	X	X	X	X
Add/Sub	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
ADC						X	X	X	X	X	X	X
SBB						X	X	X	X	X	X	X
Rotation	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
RCL						X	X					X
RCR						X	X					X

Table 2. Process instructions reading and writing condition codes.

Group III:

This is for processing instructions (arithmetic or logical) accessing status flags exclusively for writing in order to qualify the result of the operation performed. They may generate true dependences whenever a subsequent instruction reads the status register and may also generate output dependences. Table 3 lists opcodes mnemonics and the status flags written by each operation.

Group III												
	Read						Write					
Arithmetic	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
ADD							X	X	X	X	X	X
CMP							X	X	X	X	X	X
DEC							X	X	X	X	X	
DIV							X	X	X	X	X	X
IDIV							X	X	X	X	X	X
IMUL							X	X	X	X	X	X
INC							X	X	X	X	X	
MUL							X	X	X	X	X	X
NEG							X	X	X	X	X	X
SUB							X	X	X	X	X	X
Logic	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
AND							X	X	X		X	X
OR							X	X	X		X	X
ROL							X					X
ROR							X					X
SHL/SAL							X	X	X	X	X	X
SAR							X	X	X	X	X	X
SHR							X	X	X	X	X	X
TEST							X	X	X	X	X	X
XOR							X	X	X		X	X

Table 3. Process Instructions accessing condition codes in write-only mode.

Group IV:

Conditional branch instructions are in this group. They access status flags in read-only mode in order to evaluate if the condition specified for the branch is true or false. The evaluation may require to access more than one status flag and combine them logically with and-or relations. This group contains the instructions that may create true dependences with other instructions writing the flags.

Table 4 shows group IV instructions: its mnemonics and the flags read in each case. Shadowed cells are used to mean complementary conditions, that is, those looking for a false (0) instead of a true (1) value in the same flags. There are only 8 different access patterns to status flags. As can be observed, the auxiliary carry flag (A) is never used by these instructions: its utilization is limited to instructions in group II performing BCD representation adjustments.

Group IV												
	Read						Write					
Branching	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
JB/JNAE						X						
JBE/JNA			X			X						
JE/JZ			X									
JL/JNGE	X	X										
JLE/JNG	X	X	X									
JNB/JAE						X						
JNBE/JA			X			X						
JNE/JNZ			X									
JNL/JGE	X	X										
JNLE/JG	X	X	X									
JNO	X											
JNP/JPO					X							
JNS		X										
JO	X											
JP/JPE					X							
JS		X										

Table 4. Conditional branch instructions.

Group V:

Table 5 presents the final group which is for special instructions implementing loops, prefix instructions, string handling, conditional carry interrupt and carry flag handling instructions.

Group V												
	Read						Write					
Conditional loop	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
LOOPNZ/LOOPNE			X									
LOOPZ/LOOPE			X									
Prefix	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
REPZ/REPE			X									
REPNZ/REPNE			X									
String	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
CMPS							X	X	X	X	X	X
SCAS							X	X	X	X	X	X
Interrupt	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
INTO	X											
Carry flag	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
CLC												X
CMC						X						X
STC												X

Table 5. Other instructions accessing condition codes.

In all cases, the access to condition codes is done implicitly, that is, the condition is not part of the instruction codification and depends exclusively on the opcode; it can not be avoided by the programmer and there is no mechanism to disable the access when it is not meant to have computational meaning.

JCXZ, LOOP and REP are special instructions because the branch decision does not depend on the status flags but on the contents of register CX and the condition evaluation checking occurs within the execution phase of the instruction.

5. Experimental framework

We propose a dual, complementary approximation to analyze the impact caused by condition codes on superscalar execution. First we perform a statistical study to determine the usage of instructions accessing condition codes, and second, we apply an analytical method based on graph theory to obtain code coupling quantification. Both approaches have a common input data set: the execution trace of a set of programs used as testbench for the experiment.

a. Statistical approach

The statistical analysis is based on instruction counts to obtain the frequency of appearance of instructions accessing condition codes using their access patterns to status flags shown in the previous section.

As the first use of condition codes is for evaluation of condition for branches, it seems convenient to use the basic block as a natural bound to quote the sequences code. Then, as a second result, we explain the behavior of programs within the basic block from the condition codes perspective.

b. Analytical approach

The following process intends to obtain a more precise quantification applying graph theory as explained in [11]. Choosing a matrix representation, we represent code sequences and their relations as data dependences and then apply mathematical relations lo gather conclusions. It is a specific formalization to instruction level parallelism where we define restrictions and properties and gather particular operations and transformations. Details about these operations and what they mean can be found in [10].

One of the most powerful tools derived from the use of this method is the possibility of composing a matrix representing the total dependence relation from a set of different contributing matrices which correspond to the different sources of data dependence. So, it is possible to isolate and estimate the impact produced by different data types on the whole set and also perform several interesting combinations and obtain its specific contributions to the total.

c. Testbench

To continue with the study made in [20] we use the same testbench. At that time, some measures for the impact of condition codes into the potentially available parallelism were obtained. In the present work, we reinforce the result obtained as well as perform further analysis and reach to more elaborate conclusions.

The testbench is a set of DOS utility programs (*comp*, *find* and *debug*) compiled in real mode as well as some popular common applications like file compressor *rar* (v.1.52) and the *tcc* C-language compiler (v. 1.0). Program *go* from the SPECint95 suite has also been included using two different compilation options: one optimizes for size and the other optimizes for speed. Details about this testbench can be found in [18].

The programs were run in step by step mode and under a specific workload conditions to avoid

excessively long traces. Nevertheless, more than 190 million instructions were executed.

As we work with traces, the sequence of instructions corresponds to the actual execution sequence, and all the branch instructions are followed by the instruction effectively executed after the branch. Other experimental setups not based on traces, have to predict the branching in order to select instructions that should make up a sequence for a certain analysis. In contrast, we may choose code sequences of any length and these will always correspond to perfect branch predictions.

6. Statistical results

a. Distributions of instructions

There are some studies about the instructions frequently found in programs like the ones we use in our test suite. The distributions found in these studies, especially in [1, 13] are similar to the results of the detailed analysis utilization (presented in [19]) that we performed in our testbench. In the present case study we are only interested on the instructions accessing condition codes.

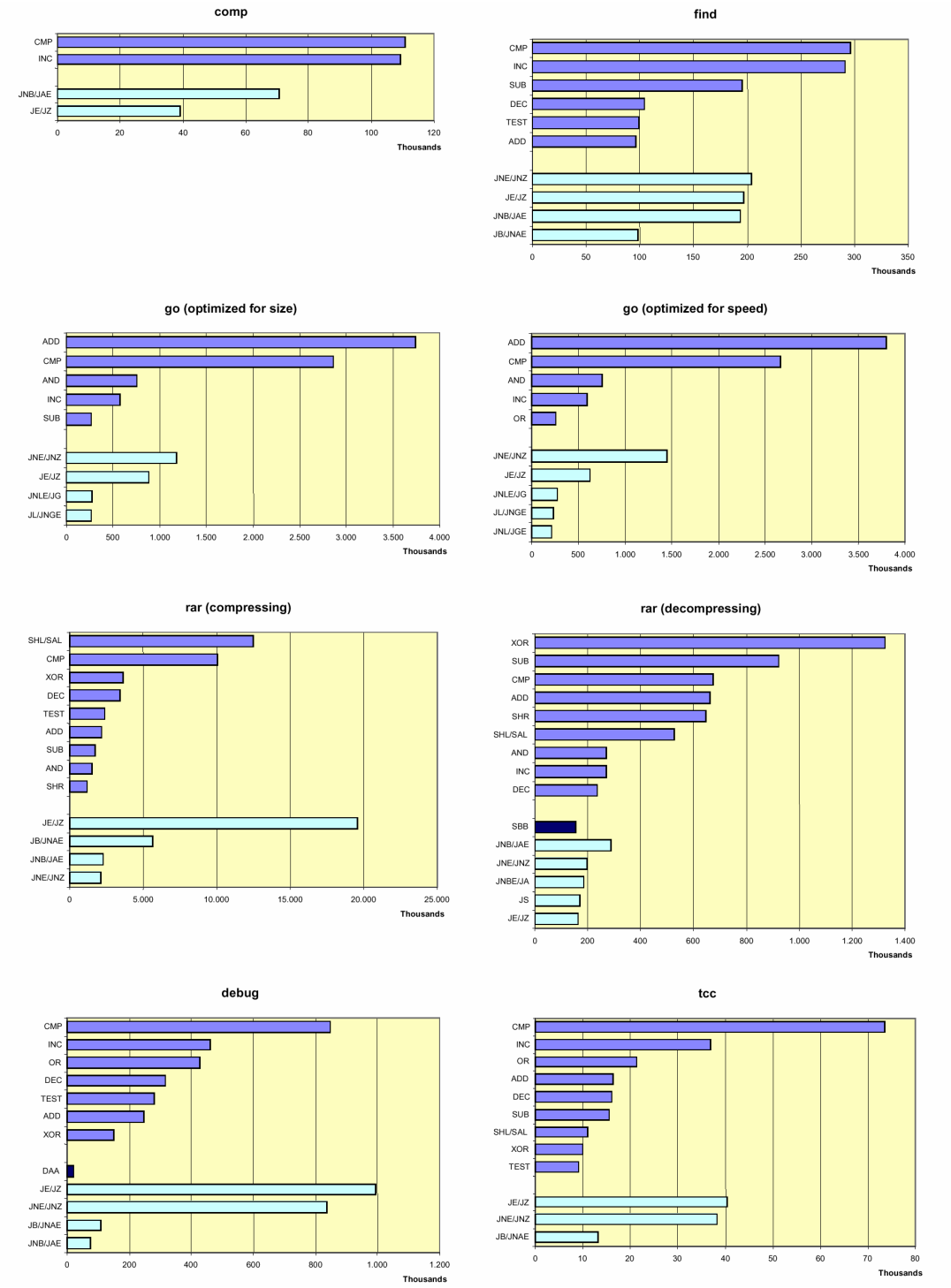

Fig. 4. Distribution of instructions using condition codes. The dark gray group bars are instruction writing condition codes. The light gray group bars are instruction reading condition codes for the purpose of branch condition evaluation. Black bars are for instructions using flags as operands.

The goal is to determine how coupling is produced among instructions because of their dependences to condition codes. We must then classify instructions in the traces according to the groups defined in Section 4 that they belong stating the type of access to the flags (read or write) and purpose. This is shown in Fig. 4 where it can be observed that only instructions for group III and IV appear significantly; group II appears occasionally and only in two of the programs of the test suite.

All traces then, contain two significant groups: group III which corresponds to instructions accessing flags in write-only mode in order to qualify the results of an operation, and group IV corresponding to conditional branch instructions whose purpose is reading the flags to evaluate a condition to branch upon. Consequently, we can state that, as mentioned in Section 3, the real usage given to condition codes by programs is passing information to a subsequent conditional branch instruction. Under these circumstances, it becomes specially relevant the partition of programs into sequences of instructions that make basic blocks, in other words, branch instructions can be used as boundaries to partition programs and observe coupling patterns.

The instructions from Group II appearing in the traces from *rar-decompressing* (SBB) and *debug* (DAA) read a particular flag (carry C and auxiliary carry A) and, consequently create true dependences with previous instructions writing into condition codes.

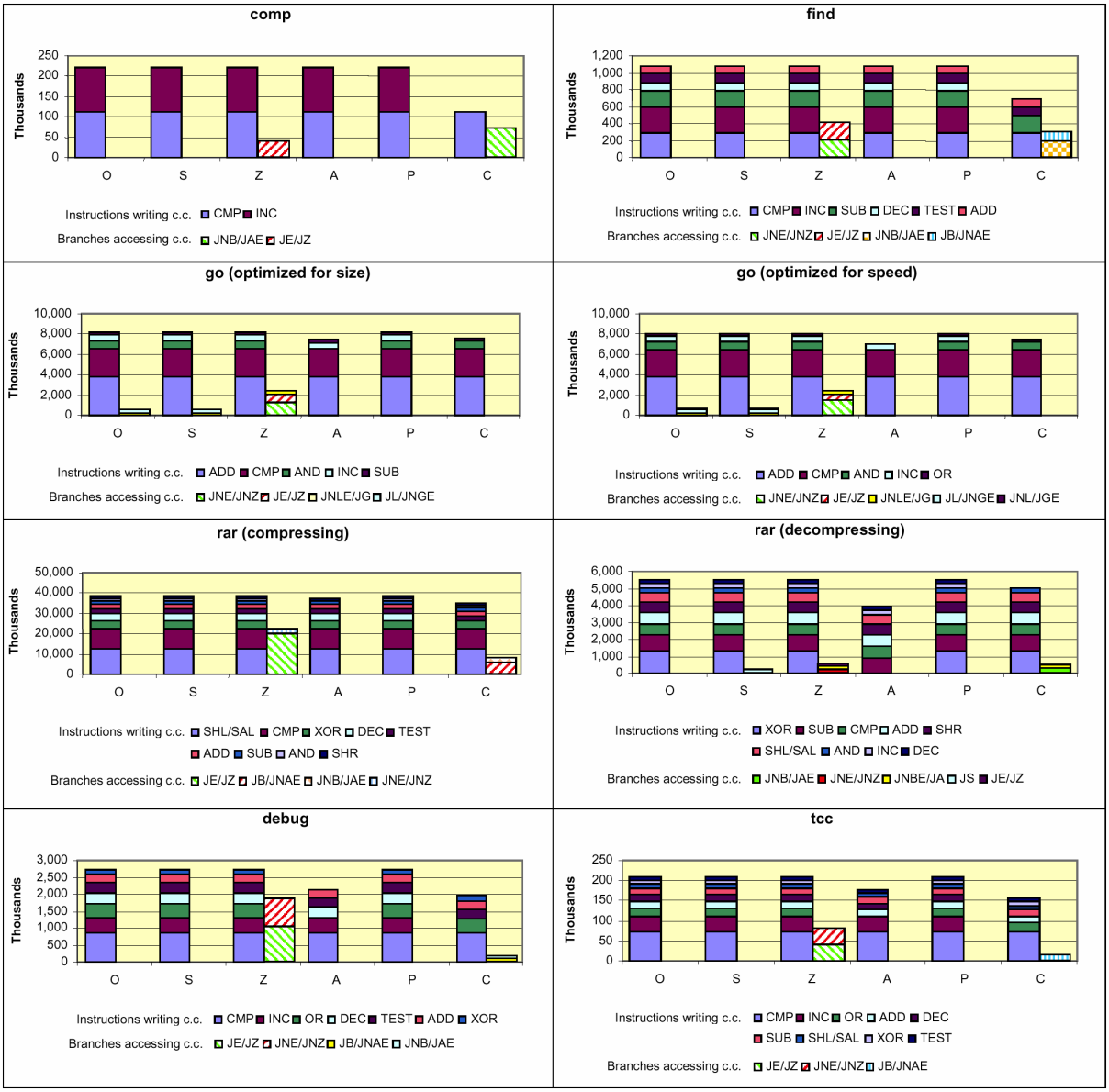

Fig. 5. Per instruction contribution to reading and writing condition codes.

Now, if we concentrate on conditional branching, Fig. 5 shows the total count of instructions in the traces writing each condition flag as well as the total count of each different conditional branch instruction reading these flags (O, S, Z, A, P and C). It can be observed that there are many more writes than reads and so, more condition codes are affected than really needed for the branch decisions used. The difference between the number of reads and the number of writes for the same flag is large in all the programs except

for *rar compressing* and *debug* where it shows a difference of less than 50% for the Z (zero) flag.

In general, writes affect all flags with few exceptions like carry (C) and auxiliary carry (A). In contrast, the status flags simultaneously evaluated (read) for branch decisions range from one to at most three, being the Z (zero) flag and the C (carry) flags the most frequently used.

b. Coupling in the basic block

A basic block is defined [2, 3, 4, 12] as a sequence of linear instructions without any branches. This structure is frequently used in compiler theory as it is a basic unit to apply local optimizations to. The structure is also advantageous for analyzing the impact produced by condition codes to the potential parallelism of a program code as it is a good scenario to identify and understand different data coupling patterns. As we are interested in the coupling produced by condition codes access only, we shall identify how these accesses impose an order of precedence in the execution of the block.

Table 6 shows the average size (in number of instructions) of the basic block found for each testbench program. It also shows the average number of processing instructions in the block. It is not possible to statistically define the mix of instructions composing the basic block in each trace except for the program comp where we can practically assume with 100% certainty that instructions COMP and INC are always present in the mix of basic blocks from the trace. This is consistent with the purpose of the program too.

Program	Average instructions per BB	Average processing instructions per BB
Comp	6.00	1.94
Find	7.68	1.56
Go tamaño	10.31	3.14
Go velocidad	10.16	2.89
Rar comprimiendo	3.19	1.25
Rar descomprimiendo	12.56	5.52
Debug	3.92	1.35
Tcc	8.98	2.28

Table 6. Average block size and average processing instructions per basic block for each program in the testbench.

Considering condition code data dependences only, each basic block necessarily contains a true dependence between the last instruction updating the status flags before the conditional branch instruction that reads it and the branch instruction itself. From a statistical point of view, the number of true dependences will increase with the number of basic blocks present in the code. In other words, the smaller the basic block, the higher the number of true dependences in the trace. Therefore, according to Table 6, programs *rar-decompressing* and *debug* having the smallest basic block size show the highest potential for parallelism due to the lack of coupling. This idea is also confirmed by the results found in [20].

Another typical coupling in the basic block is due to output dependences: the order imposed by processing instructions performing successive writes to the same resource, in this case, the status register. The limitation on the amount of available parallelism due to this type of dependence is a direct consequence of the instruction set architecture and it has no computational meaning at all. A good example is found again in the trace for program comp where the basic block uses instruction CMP before the branch instruction to set the condition code to jump upon, while instruction INC, always present in the block too, is only used for updating an address pointer and is not related to the jump condition; however INC has an output dependence with instruction CMP as a side effect.

Statistically, the average length of output dependences chains grow with the number of processing instructions in a basic block. Data in Table 6 states that the traces for *rar-decompressing* and *go* show the blocks with higher number of processing instructions, therefore these programs are good candidates to exhibit high coupling caused by output dependences. Large basic blocks also tend to contain large number of processing instructions.

Finally, true dependences caused by instructions writing into the status register followed by other processing instruction consuming this data is practically nonexistent, as we observed in Fig. 4 for group II instructions.

In summary we can reasonably state the following:

- Larger basic block sizes decrease the hazard of true dependences caused by condition codes.
- Larger basic block sizes may increase the length of output dependence chains caused by condition codes.

7. Quantitative evaluation method based on graph theory

The statistical analysis based on instruction distributions provide a qualitative knowledge of the impact caused by condition codes to superscalar execution. In the current section we explain how we provide a quantitative measure for such an impact using an analytical method derived from graph theory.

Applying graph theory to evaluate instruction sets architectures has several advantages:

- it provides a simple description of the problem
- it allows to predict behavior
- it simplifies the transmission of knowledge
- it separates the study of instruction set's characteristics from the hardware that should interpret it.

It is interesting to outline that the method's approach provides for the evaluation of performance of each layer of the computational process in an isolated manner, without being conditioned by the rest of layers.. The most popular evaluation method tool used today is simulation but simulators inherently mix the behavior of the Instruction set with the behavior of the physical layer. It is important to develop methodology to support the principle of isolated evaluation as suggested in [22].

To model fine grain parallelism with graphs, like in other fields where they have been used successfully, we need to define the representation chosen, a set of applying restrictions and properties for the particular case, and a set of parameters whose

mathematical calculation correspond to a measure of the amount of available parallelism.

In our approach, we model instruction sequences that show data dependence as directed graphs and then use the matrix representation of these graphs for data processing. The matrix is called the dependence matrix, noted as *D*. The detailed mathematical development leading to the following summary of parameters definitions can be found in [10, 11]:

- Dependence matrix *D* is defined as:

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

It is normally evaluated for a sequence of *n* instructions in the code known as instructions window.

- Coupling *C* measures the amount of data dependences in an instructions window. It is computed used matrix *D* and its value is bounded by the following limits:

$$C = \sum_{i=0}^{n-1} \sum_{k=0}^{n-1} d_{ik} \qquad 0 \le C \le \binom{n}{2} \quad (2)$$

- The length of the critical path *L* measures the longest dependence path found in the n instructions window. The units we use are computing steps:

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

- G_p is the parallelism degree, a parameter derived from *L* representing the amount of available parallelism found in the window. The expression and bounds for it are:

$$G_p = n / L \qquad G_p \in [1, n] \quad (4)$$

- One of the most powerful properties of the method is the compositional nature of matrix *D*, which states that this matrix corresponds to the resultant of the contribution from different sources of dependence which are, in turn, individual matrices representing isolated sources of dependences. The expression for the combination is:

$$D = D_{s1} \text{ OR } D_{s2} \text{ OR } \ldots \text{ OR } D_{sn} \quad (5)$$

- As we defined L as the length of the critical path of the full combination of all source dependences resulting on *D*, then L cannot be any shorter than the largest critical path L_{si} found in any of the component matrices:

$$\max_i \{L_{si}\} \le L \le \min\left\{\sum_i L_{si}, n\right\} \quad (6)$$

This means that the resulting parallelism degree for the full composition will never be higher than the parallelism degree found in anyone of the components representing the different types of dependence sources.

a. Data dependence sources combinations

The complete space of contributing data dependence sources selected depends on the objective of the ongoing analysis. Then, computing the compositions resulting from the inclusion or exclusion of particular sources of data dependences, we obtain figures for its relevance.

In our case, we are focusing on condition codes as source of dependences, therefore, our space is divided into two data types: condition codes and the rest. So, we build matrices taking into account only one of the contributions and matrices including both contributions.

To quantify the contribution of this source of dependence to the total we also need to distinguish among the different types of dependences namely:

- True dependences: read after write
- Anti-dependences: write after read
- Output dependences: write after write

Anti-dependences and output dependences are produced because of the utilization of the same resource and can be eliminated if the resource is changed to avoid coincidences, therefore these are non-true dependences.

Id		dependence type: True	Anti	Output	data type: Condition codes	Others
1	a	✓	✓	✓	✓	✓
	b	✓	✓	✓	✓	∅
	c	✓	✓	✓	∅	✓
		✓	✓	✓	∅	∅
4	a	✓	✓	∅	✓	✓
	b	✓	✓	∅	✓	∅
	c	✓	✓	∅	∅	✓
		✓	✓	∅	∅	∅
3	a	✓	∅	✓	✓	✓
	b	✓	∅	✓	✓	∅
	c	✓	∅	✓	∅	✓
		✓	∅	✓	∅	∅
5	a	✓	∅	∅	✓	✓
	b	✓	∅	∅	✓	∅
	c	✓	∅	∅	∅	✓
		✓	∅	∅	∅	∅
2	a	∅	✓	✓	✓	✓
	b	∅	✓	✓	✓	∅
	c	∅	✓	✓	∅	✓
		∅	✓	✓	∅	∅
6	a	∅	✓	∅	✓	✓
	b	∅	✓	∅	✓	∅
	c	∅	✓	∅	∅	✓
		∅	✓	∅	∅	∅
7	a	∅	∅	✓	✓	✓
	b	∅	∅	✓	✓	∅
	c	∅	∅	✓	∅	✓
		∅	∅	✓	∅	∅
		∅	∅	∅	✓	✓
		∅	∅	∅	✓	∅
		∅	∅	∅	∅	✓
		∅	∅	∅	∅	∅

Table 7. All possible combinations of data dependences and dependence types obtained by inclusion/exclusion in the mix. The label is used to identify the mix of contributions in the text for the analysis.

Table 7 lists all possible combinations of compositions for the grouping proposed. As there are 5 possible contributions, we may select 32 combinations; symbols ✓ and ∅ indicate participation or exclusion in the contribution.

Some combinations don't make sense, for instance: considering no contributions at all or not considering at least one dependence type in a mix. In fact, only 21 combinations (shown in colored rows) out of the 32 are considered valid, and we can set them into 7 classes according to the types of dependences included in the mix. Then we have the following classes:

1. contribution from all dependence types
2. contribution from all dependences but true dependences (non-true dependences only)
3. contribution from all dependences but anti-dependences
4. contribution from all dependences but output dependences
5. contribution from true dependences only
6. contribution from anti-dependences only
7. contribution from output dependences only

Groups 3 and 4 are not useful as they combine true dependences with one of the non-true types and it has no relevant meaning. For each one of these five remaining groups (1, 2, 5, 6 and 7) we may further have three compositions according to the data types being considered; these are:

a) contribution from all data types
b) contribution from condition codes only
c) contribution from non condition codes only

Composition ID		Composition Mix	
1	a	ALL	All data
	b		CC only
	c		Non CC
5	a	TRUE	All data
	b		CC only
	c		Non CC
2	a	NON TRUE	All data
	b		CC only
	c		Non CC
6	a	ANTI-dependences	All data
	b		CC only
	c		Non CC
7	a	OUTPUT	All data
	b		CC only
	c		Non CC

Table 8. Listing of each composition identifier and its components.

So for each group we have information about the contribution from all data, from condition codes solely and from data other than condition codes. For example, as depicted in Table 8, the composition 1b records all types of dependences produced by condition codes only; composition 5a corresponds to true dependences due to accesses to all data resources, and 7c records output dependences found due to the access to all data resources but to the status register.

Interpreting parameters upon different compositions we can reach valuable information about the effective impact of condition codes on real programs. According to Equation 6, the critical path length of a dependence source component is a low limit for the coupling of the full composition. Therefore we will choose the longest path length among all possible partial components to be the lower bound for the composition.

According to this, we take into account the isolated contribution from condition codes to obtaining the critical path length for this source (L_{cc}). We consider this value as a provisional lower bound for L, the critical path length of the composition; note that the rest of dependences may produce a limit for the composition that may be equal or higher than L_{cc} but never less.

Then we find L_{ncc}, the critical path length of the rest of sources excluding condition codes, obtain a quantification of the relevance of the excluded contribution (condition codes). If L_{ncc}, is lower than L_{cc}, sufficiently, this means that there is room for improvement in the full composition by eventually minimizing condition codes contribution.

8. Quantifying the impact of condition codes accesses

The method described in Section 7 for the quantitative evaluation is performed automatically by a application designed for this very purpose [27]. It allows the analysis of variable size of instructions window. Given a profile for the dependence contributions, it builds the relevant dependence matrices and obtains the parameter set presented in Section 7 for each component and the total composition.

We selected static 512 instruction sequences windows. Sliding windows, the typical mode used for the physical layer of processors and simulators, is an excessively heavy load for the computation and it adds no additional precision compared to a scenario using sufficiently large static windows. We tested window sizes up to 2048 instructions and found practically no changes in results obtained while computing time substantially increased. On the other hand, relevant literature also confirms that, for a large size of instructions windows, the information obtained from sliding and static windows is the same [26]. We can also argue that there is a very significant difference of magnitudes between the number of instructions in a large window and the number of data locations defined in the ISA, even considering memory as a single resource, so the frontier effects caused by a static window can be neglected.

a. Critical Path Length

Table 9 shows the length of the critical path for the different compositions defined for all the programs traces of the testbench.

Generally, the critical path length grows when more data dependences sources contribute in the composition used. It is the case for class “a” (dependences caused by all data) from all groups show the highest values. Nevertheless, it is interesting to observe how subgroup b which records contribution from condition codes solely, shows a very low value (around 2 in almost all traces) in groups 5 and 6. These groups correspond to true dependences and anti-dependences. A critical path length of 2 means that there are only two sets of instructions: one set produces data consumed by the other set; and these instructions are independent within their respective sets.

Composition ID		comp	find	go (t)	go (v)	rar (c)	rar (d)	debug	tcc
1	a	398.70	321.90	306.98	311.72	419.67	405.71	390.39	336.11
	b	257.53	150.36	197.00	199.22	341.78	240.05	312.34	170.23
	c	337.30	304.35	273.23	274.56	238.99	324.11	247.66	289.81
5	a	248.32	229.91	94.30	85.14	132.12	177.16	132.41	151.87
	b	2.01	1.98	2.00	2.00	2.00	2.28	2.81	2.28
	c	247.51	229.83	92.88	83.65	131.40	175.96	124.35	147.76
2	a	233.41	271.40	215.56	220.97	247.30	320.78	258.55	237.68
	b	172.48	116.59	146.13	147.63	214.59	204.98	192.59	119.09
	c	177.85	261.52	161.10	161.86	212.68	230.88	238.80	211.57
6	a	148.86	212.85	110.81	112.82	137.22	189.31	137.25	175.41
	b	2.01	1.98	2.00	2.00	2.49	10.02	4.11	3.52
	c	148.65	212.04	108.48	109.91	136.04	183.69	132.72	174.17
7	a	178.34	230.24	175.53	176.87	230.57	235.39	248.43	182.82
	b	172.29	116.43	146.03	147.53	214.26	204.87	192.28	118.98
	c	147.62	228.36	136.96	139.04	197.98	135.48	234.92	164.16

Table 9. Critical Path Length in computing steps for a static 512 instructions window, for different compositions and for each program trace.

In the case of group 5b this corresponds to a read after write (true) dependence within a basic block where a processing instruction writes a condition code that is read by the branch instruction following it. In the case of group 6b we have a couple of instructions where the first one reads a condition (to evaluate a branch condition) and the next one writes it (a processing instruction after the branch); this corresponds to inter-block dependences crossing the boundary between two consecutive basic blocks.

b. Degree of parallelism G_p

Applying Equation 4 to data in Table 9 we can obtain the degree of parallelism G_p for each composition of data dependence. Figure 6 shows G_p for the contribution from all data dependence sources. The conclusion is that being G_p in the range of 1.22 to 1.67, it is only possible to obtain a global parallelism of about 50%. This result is in agreement with the results obtained in other research work about the x86 ISA [5, 13, 14, 17, 20, 23], which is an important fact to validate our methodology.

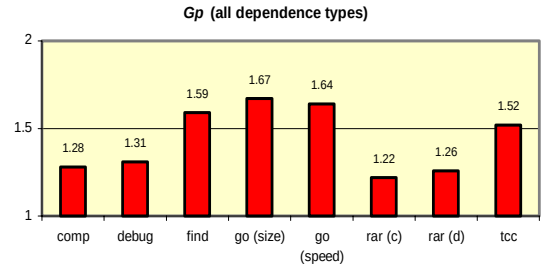

Fig 6. Parallelism degree G_p for each trace in the testbench when all data dependence type contributions are considered.

c. Impact analysis per dependence type

Figure 7 presents the impact of condition codes on the length of the critical path by contrasting the value of this parameter when they are included-in or excluded-from the different compositions of dependences sources. The impact can be observed in detail for all dependence types and for some compositions as well. The graphs are normalized, that is: 100% corresponds to the value for the critical path taking into account all data types contribution (condition codes and the rest of sources) for the particular dependence combination tracked.

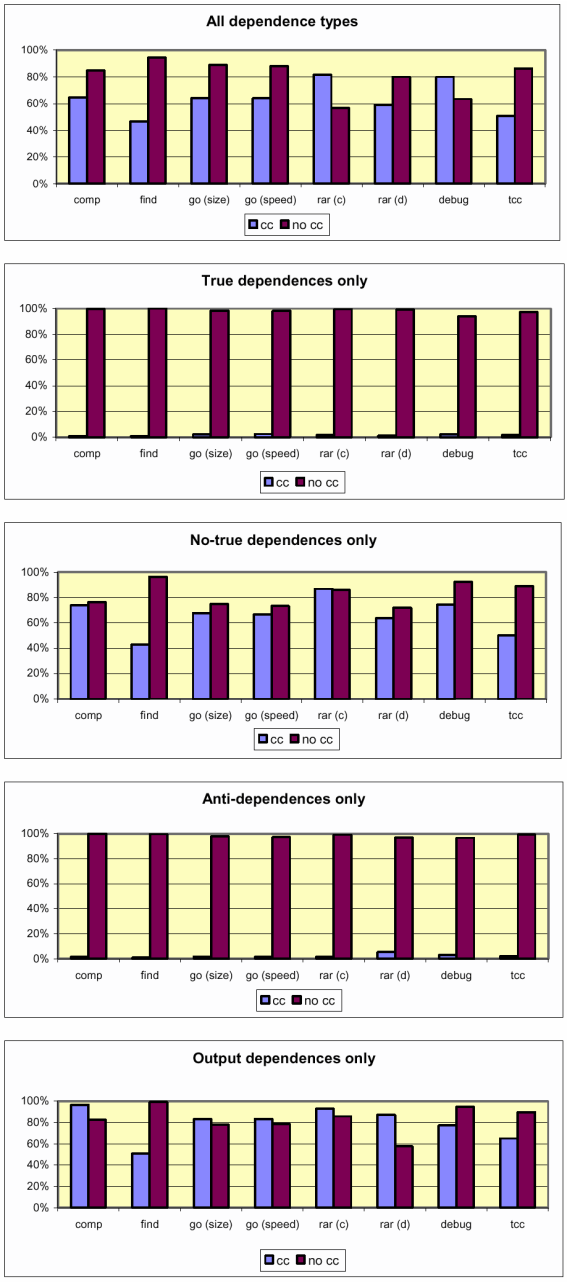

Fig. 7. Contribution of cc/no cc on different data dependence types.

The first graph shows the compositions for all types of dependences. In the case of *rar-compressing* and *debug* the composition of condition codes has a path length L_{cc} exceeding the value obtained for the composition of the rest of sources L_{ncc}. In these two cases, the path length L_{cc} is a low bound limit to the complete composition's critical path length L which means that any improvement in the general decoupling due to changes operating on rest of data will be jeopardized by the dependences caused by condition codes. These results are in agreement with the results presented in [20] where the absence of dependences caused by condition codes produces a very important performance improvement for these same programs.

The second graph shows that the impact caused by condition codes in true dependences is neglectable compared to the impact caused by other data types contribution. Now, among all program traces of the testbench, debug shows the longest critical path (2.81 computing steps as shown in Table 9) in the composition for only condition codes into true dependences. This agrees with the hypothesis introduced in Section 6 about higher hazards of true dependences (caused by condition codes) in smaller basic blocks given that, according to Table 6, the basic block size for *debug* is only 3.92 instructions. This data also demonstrates that instructions from Group II (processing operations using condition flags as input operands) are used very rarely and have no influence worth considering in the analysis for superscalar execution.

The third graph illustrates the weight of condition codes dependences over non true dependences, which is equivalent to other data types' contribution. Only for *rar-decompressing* code coupling in slightly higher.

Graphs 4 and 5 show that condition codes contribute basically as output dependences and cause practically no anti-dependences compared to other sources. The most important contribution occurs for the trace of program *rar-decompressing*. Apparently, this reinforces the hypothesis introduced in section 6 stating that large basic blocks may increase the length of output dependence chains due caused by condition codes.

d. Impact analysis per data type

Figure 8 provides a view of the condition codes source (solely) contribution to each dependence type on all program traces. Figure 9 shows the same information for the rest of dependence sources. In both cases, data is normalized considering 100% as the length of the critical path for all data dependence sources and all types of dependences, that is, the critical path length of the full composition.

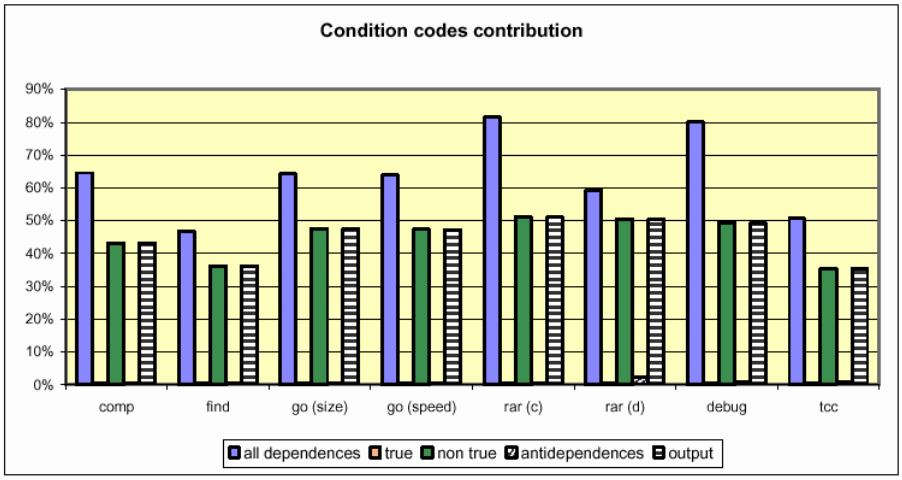

Fig. 8. Isolated contribution of condition codes to the different compositions of dependence types for each program trace of the testbench.

Again in Fig. 8 we observe that, mainly, condition codes produce output dependences and the columns for true and anti-dependences are very low. However, in general, the combination of these two (true dependences and anti-dependences) with the output dependences seem to enlarge the overall dependence chains. It seems like the few existing true and anti-dependences would link two or more output dependence chains producing a new longer chain.

Another interesting observation results from the fact that the resulting total dependences are much larger for program traces *rar-decompressing* and debug than it is for the rest of program traces. It seems like, when output dependences are accounted together with the other dependence types (true dependences and anti-dependences), the combination produces very different magnitudes of *L*. As both of these programs exhibit a block size quite smaller compared to that of the others programs from the testbench, there seems to be a correlation between the size of the basic block and this effect of irregular enlargements.

Figure 9 shows a more equilibrated contribution among the different dependence types when condition codes contribution is excluded. True dependences have a similar weight compared to non-true dependences being the later slightly higher for all program traces except for *comp*. Composing both dependence types (total dependences) increases the length of the critical path, although in an rather heterogeneous manner.

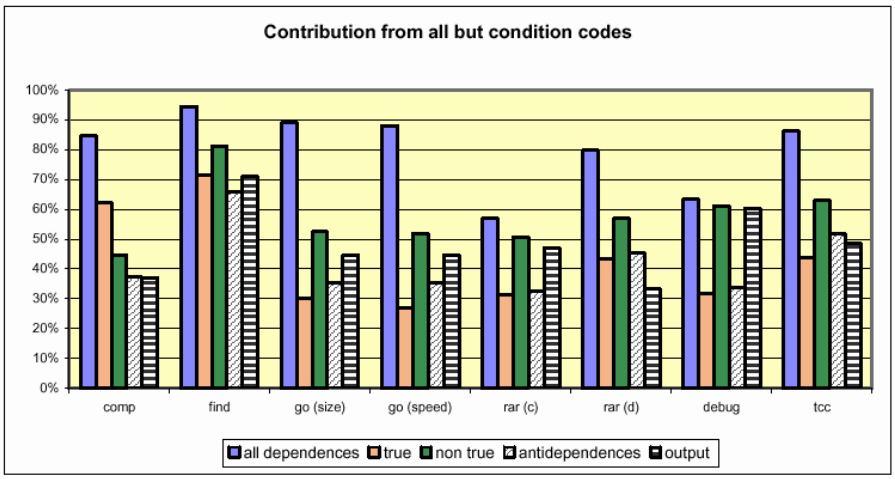

Fig. 9. Contribution of non condition codes data types to the different compositions of dependence types for each program.

9. Microperation level impact

Processors of the x86 family use a 2-level microarchitecture to improve performance. The top level acts as an interface to the CISC instruction set, translating instructions into RISC-type microperations which are executed in the low level machine. Decoding is performed in three different units: the simple, the general and the sequencer units. Instructions decoded by the sequencer unit are executed serially, while instructions decoded by the other two units are executed in a superscalar fashion.

Huang and Peng have analyzed the distribution of the number of microperations a single CISC instruction gets decomposed [13]. Results are summarized in Table 10. Most instructions (67%) are translated into only one microperation and from the rest almost 90% get translated into two microperations. The weighted average value is 1.41 microperations per CISC instruction. Other researches present very similar figures: 1.26 in [14] and 1.35 in [5].

Microperations per instruction	
1	67%
2	22%
3	7%
4	3%
more than de 5	<0.5%

Table 10. Distribution of number of microperations per CISC instruction.

According to these numbers, we can conclude that the transformation from CISC to RISC only increases the number of nodes in the dependence graph by a factor of 1.5, which means that the structure of the graph is not substantially changed.

Moreover, data coupling must be preserved across the transformation. The question is how does it affect the of dependence chains? To answer this question we need to analyze how a CISC instruction gets decomposed into several RISC instructions. Basically it depends on addressing modes. When a CISC operand is in memory, automatically, it gets translated into two operations: a RISC load/store operation used to transfer the operand to/from memory from/to the CPU registers and a RISC processing instruction that gets the operands from the CPU registers and performs the operation. Consequently, dependence chains experiment an enlargement that basically corresponds to the increase of the number of nodes in the graph. Figure 10 shows the potential transformations happening in the CISC graph when a shadowed node may split into two nodes. As the dependence relation it holds with the following node should be maintained the division may correspond to one of the pictures: (a) or (b). Case (b) is for an "atomic" transformation, that is, from the point of view of coupling the node has not forked and this corresponds to the most frequent case.

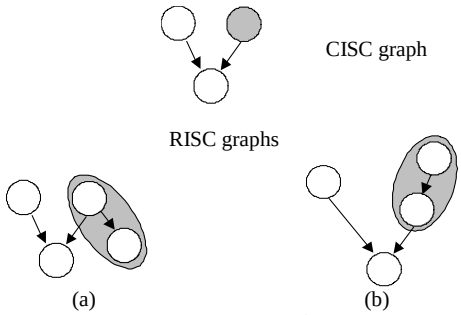

Fig. 10. Possible CISC to RISC graph transformations.

The slight enlargement of dependence chains experimented in the resulting RISC graph, combined with same increase of the number of instructions at the RISC level result in a practically null increase of the parallelism at this level compared to that present in the sequence of CISC code.

Parallelism available for execution in superscalar mode is limited by three factors: the parallelism available at the machine language level, the capacity to extract parallelism of the hardware that should interpret the machine language and the availability of resources at the physical layer. Assuming that physical resources are not a constraint and that the parallelism at the CISC language level does not change much after the transformation into RISC type code, we should now analyze the capacity of the physical layer to extract parallelism.

We refer to the capacity to extract parallelism as the ability to find independent operations in an

instruction window. In the present case, these operations are microperations. True dependences caused by condition codes cannot be dissolved (ignored/overlooked) and the execution pace should follow the partial ordering imposed by the dependence graph. Non-true dependences can disappear by using register renaming techniques, and the time of conversion from CISC to RISC is a good opportunity to apply it. In the case of condition codes, non-true dependences come basically as output (write after write) dependences, as we showed in previous sections. However, it must be taken into account that the register to rename is a special register (status register), and that these output dependences have no computational meaning and are a direct consequence of the architecture of the instruction set.

10. Solutions and its cost

We may present a few ideas to minimize the impact caused by condition codes of the x86 instruction set and analyze its cost, advantages and drawbacks. There are two types of solutions, one center on the physical layer implementation options and the other on possible extensions of the instruction set. Neither proposal mean a substantial change of the architecture of the instruction set: condition codes to evaluate branches and processing instructions using condition codes as operands are maintained. We rather target at cases where non true dependencies are produced.

About physical layer changes, we think that the register renaming strategy is an adequate technique of proven effectiveness and widely used. It would be advantageous to count on a status register pool whose size should match the instruction window size used to accommodate independent instructions to issue. Every new write into the status register would be performed on a different temporary storage element.

The renaming technique is simple but not very efficient in the case of condition codes because many bits are written but only a few are read, as shown in the distributions of condition codes usage analyzed in previous sections. Additionally, using hardware to increment parallelism is costly due to a considerable increase of required silicon area and power consumption. Unfortunately, all this overhead would be focused to solve output dependences which carry no computational meaning and these additional resources would show a low utilization profile or may be used most of the time without a real necessity. It would be another example of incrementing processor resources for supporting large out-of-order executions without effective benefits [9].

About changes in the instruction set, the proposal focuses on non-true dependences cases taking advantage of the CISC to RISC translation process. The CISC format should not change in order to maintain backwards binary compatibility, but the RISC kernel may be changed. If we establish a status register conditional write mode and enable the RISC layer to detect it, then it could be set to inhibit/enable writes to the status register upon execution of microperations. In this manner the RISC layer can be set to write the status register only for the microperation responsible to set the condition for a subsequent branch, and it can be set to inhibit writes for the execution of all other microoperations eventhough the originating CISC instructions do perform a write.

The decision to set and selectively inhibit the writing mode could be performed by the compiler and the strategy chosen may consist on the inclusion of a number of NOP operations acting as escape sequence. For example, three NOP instructions in the program start may flag the RISC processing logic to set the writing mode, and a single NOP instruction may cause it to use the non-writing mode for the next instruction.

A processor implementing this function may execute a program compiled for it taking advantage of the superscalar execution and avoiding output dependences caused by condition codes. The same compiled program would also execute in a regular processor (binary compatibility is maintained) although it would exhibit some execution time penalty due to the increment of instructions (NOPs).

To prevent an erroneous execution of programs not compiled for the modified processor, in case the program happens to have the write disabling sequence (i.e. three NOPs at the beginning), the processor should check that there is also an enabling sequence (one NOP) right before the first status register read (i.e. the first conditional branch).

11. Conclusion

In the x86 ISA, in addition to asses branch decisions, condition codes may also be used as input operands in some instructions, although this use is very unusual as evidenced by the usage distribution analysis done.

The distribution of flags accesses reveal that there are more flags written than later read and that each flag is written many more times than it is read. This circumstance conditions the resulting type of data dependence tyeing instructions.

The analysis made show a correlation between the size of the basic block and dependence hazards. Large basic block decrements the hazard of coupling due to true dependences caused by condition codes, although a larger block size may also produce lengthening of output dependence chains due to condition codes.

Quantifying the impact into instruction level parallelism produced by condition codes using our method produces the following results:

- Condition codes decrease the amount of available parallelism generating output dependences basically. These types of dependences can be avoided using register renaming techniques, but because they have no computational meaning and are only originated due to the architecture of the x86 ISA, it makes this hardware solution an absolute waste of resources.
- Data dependence sources other than condition codes cause an enlargement of the dependence chains. There is a correlation with the basic block size so

that when it is short, the enlargement is found higher due to the contribution of true dependences.

Transforming the stream of CISC instructions to RISC instructions does not produce a substantial modification on the impact caused by condition codes into the instruction level, neither in the microperation level.

Finally we propose a mechanism to avoid these unnecessary dependences with no computational meaning created by condition codes that would increase superscalar performance improving the parallelization of code execution. The proposal does not affect binary compatibility, can be driven from the machine language level and consist on enabling an internal execution mode in the hardware that implements conditional writing of the status register. This approach, compared to other hardware solutions, avoids incrementing the complexity of the physical layer and consequently does not have a negative impact on area and power consumption.

12. 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*, pages: 152 – 160, April 1989.

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

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

[4] A. Aho and J. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[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*, pp. 288 –297, 1997.

[6] P. Bose. *Instruction Set Design for Support of High-Level Languages*. Ph. D. Dissertation, University of Illinois at Urbana-Champaign, 1983.

[7] D. Burger, S. W. Keckler *et al*. "Scaling to the End of Silicon with EDGE Architectures," *IEEE Computer*, vol. 37, 7, pages: 44 – 55, July 2004.

[8] D. Clark and H. Levy. "Measurement and analysis of instruction set use in the VAX-11/780," in *Proceedings of the 9th Symposium on Computer Architecture*, pages: 9 – 17, April 1982.

[9] A. Cristal, J. F. Martínez, J. Ll., and Mateo Valero. A Case for Resource-conscious Out-of-order Processors. Computer Architecture Letters, IEEE Computer Society, Vol. 2, no. 2, November 2003.
Also available as: Technical Report No. CSL-TR-2003-1034, May 2003.

[10] R. Durán and R. Rico, "On Applying Graph Theory to ILP Analysis," *Technical Note TN-UAH-AUT-GAP-2005-01*, March 2005. Available at: http://atc2.aut.uah.es/~gap/

[11] R. Durán and R. Rico, "Quantification of ISA Impact on Superscalar Processing," in *Proceeding of EUROCON2005*, pages: 701 – 704, November 2005.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture a Quantitative Approach. 2nd edition.* Morgan Kaufmann Publishers, 1996.

[13] 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, pages: 929 – 939, June 2002.

[14] 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.

[15] A. Lunde. "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," *Communications of the ACM*, vol. 20(3), pages: 143 – 153, March 1977.

[16] W. D. Maurer. "A theory of computer instructions," *Journal of the ACM*, 13(2), pages: 226 – 235, April 1966.

[17] 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)*, pp. 129–140, 2003.

[18] R. Rico, "Proposal of test-bench for the x86 instruction set (16 bits subset)," *Technical Report TR-UAH-AUT-GAP-2005-21*, November 2005. Available at: http://atc2.aut.uah.es/~gap/

[19] R. Rico, "Analysis of x86 Data Usage (16 bits subset)," *Technical Report TR-UAH-AUT-GAP-2005-22*, November 2005. Available at: http://atc2.aut.uah.es/~gap/

[20] 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.

[21] M. S. Schlansker and B. R. Rau. "EPIC: Explicitly Parallel Instruction Computing," *IEEE Computer*, pages 37-45, February 2000.

[22] 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.

[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*, pp. 57-66, 2000.

[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. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176-188, April 1991.

[27] Software tool (source code, configuration files and documentation): data dependence analyzer ADD; CVS repository (anonymous user): CVSROOOT:pserver:anoncvs@atc2.aut.uah.es:2401/home/cvsmgr/repositorio