Universidad de Alcalá

G.A.P. Departamento de Automática — Grupo de Arquitecturas Paralelas

Analysis of x86 Data Usage (16 bits subset)

Technical Report TR-UAH-AUT-GAP-2005-22-en

Rafael Rico

Department of Computer Engineering, Universidad de Alcalá, Spain

December 2005

Versión en español:

Análisis del uso de datos en el repertorio x86 (subconjunto de 16 bits)

Informe técnico TR-UAH-AUT-GAP-2005-22-es

Rafael Rico

Departamento de Automática, Universidad de Alcalá, España

Abstract:

To study the behavior of instruction sets in the superscalar setting to analyze the data usage is necessary because the main limiting factor to parallel execution is the data dependences among instructions.

This technical report shows the data usage distribution for x86 instruction set, 16 bits subset. The work has been done with a predefined test-bench.

The detailed study of data access has been organized as follows. First of all, the explicit register usage is analyzed, next the implicit usage and finally, the status flag usage is studied.

From usage distributions for each group, quantitative results about the most important sources of potential data dependences are obtained.

Index words: Evaluation of computer architectures, instruction level parallelism, instruction set architecture.

Resumen:

Para estudiar el comportamiento de los repertorios de instrucciones en el entorno de procesamiento superescalar es necesario analizar el uso que se hace de los datos ya que el factor limitante más importante de la ejecución paralela son las dependencias de datos.

El presente informe técnico muestra la distribución del uso de los datos para el repertorio x86, subconjunto de 16 bits. El trabajo se ha realizado a partir de un banco de pruebas predefinido.

El estudio detallado del acceso a datos se ha organizado como sigue. Primero se analiza el uso explícito de registros, a continuación el uso implícito y finalmente se estudia el uso de las banderas de estado.

A partir de las distribuciones de uso de cada grupo se obtienen resultados cualitativos acerca de las fuentes más importantes de potenciales dependencias de datos.

Palabras clave: Evaluación de arquitecturas de computadores, paralelismo a nivel de instrucción, arquitectura del repertorio de instrucciones.

1. Introduction

When it is intended to study the instruction sets behaviour in the superscalar processing setting to analyze the data usage is necessary.

The present technical report shows the data usage distribution for the x86 instruction set, 16 bits subset. The work has been made from a test-bench which in depth description can be known in previous report TR-UAH-AUT-GAP-21, titled "*Proposal of test-bench for the x86 instruction set (16 bits subset)*" [5].

2. Instruction frequency of use

Although it is not the aim of this report, a study has been performed on the frequency of use of the instructions, grouping them by their mnemonics. We are going to present our results comparing them with the obtained ones in 1989 by Adams and Zimmerman [1]. As these authors assure that, still nowadays, there are very few studies on dynamic traces of x86 instruction set. In this report, 190 million instructions have been traced in opposition to 18 million that they traced. Table 1 shows the top 25 more used instructions in average according to both studies.

At first sight, it is clear that both sets of instructions are similar. In the data of 1989, the 6 instructions that have left the table respect our study have been highlighted on grey. The leaving of the list of LOOP was awaited since, as explains Randall Hyde in the last version of their book on the assembly language [3], the compilers has yielded use this instruction due to the dedicated employ that imposes through counter register CX: when several loops are nested it is necessary to perform a continuous data movement with the stack to save the index coherence. Only 3 test-bench applications use LOOP and they do it with a weight around to 1.5%: RAR decompressing, SORT and TCC (see table 2 with the top 25 more used instructions for each one of the traces).

The instructions LES and LDS also leave the top list possibly because our test-bench does not count on great excessively programs nor on very large areas of data.

The rest of leaving instructions has more to do with the profile of the traced programs that with compilation criteria.

In our table, we see that instructions MOVS, JB/JNAE, AND, SCAS, STOS and CLC have entered. Among the three string instructions enclosed (MOVS, SCAS and STOS), the first is due to the SORT trace whereas the other two have a very irregular use in the rest of traces. Notice that we have counted all the occurrences of strings operations whereas the authors of cited previous work only counted them once (discarding the repetition prefix) measuring the string length with the aim of calculating the average string length. Logically, our counts are much greater when the lengths of the strings are large. This metric diversity affects something to the percentage of use of operations.

In many cases the percentage of use are similar (MOV, CMP, JNE/JNZ, SHL/SAL, SUB, XOR, DEC, OR) since they correspond to basic instructions of the instruction set. Otherwise, it is significant that PUSH and POP has diminished in percentage and simultaneously they have approximated their values. The explanation can be in an optimized compilation to diminish the transferences with memory through the stack and in a better use of the available registers[1].

It is necessary to emphasize that MOV is the most frequently used operation with great difference. Among all traces only two do not have it in first position: DEBUG, in favour of the conditional branches, and SORT, in favour of string movement.

The second operation in top 25 list is CMP, an arithmetical operation (subtraction) that does not write results. This behaviour has great importance because not writing in destination operand does not generate dependences by explicit data[2], although it does generate through implicit data[3]. The resulting information of the comparison is used to update the status register writing in the flags. It is there where the possible dependence settles down. Usually, it forms pair with a conditional branch, reason because between it and the bifurcation usually is not any other instruction that modifies the status register.

As far as the conditional branches are concern, we see that only there are four different ones that are complementary two to two. Consequently, we can say that among the 16 operation codes corresponding to conditional branches in this instruction set, many of them could not exist. Something similar happens with other instructions: BCD adjust operations, XCHG or the already commented case of LOOP.

In the end of list we find SHR, which do not leave the list thanks to the use that gives it program RAR, and CLC, which inclusion is due to FIND.

Below, we can glimpse a series of comparative graphs throughout all the test-bench about the following aspects: transferences with the stack, procedure calls, sequential run times and basic block sizes. In all the graphs the average value has been plotted.

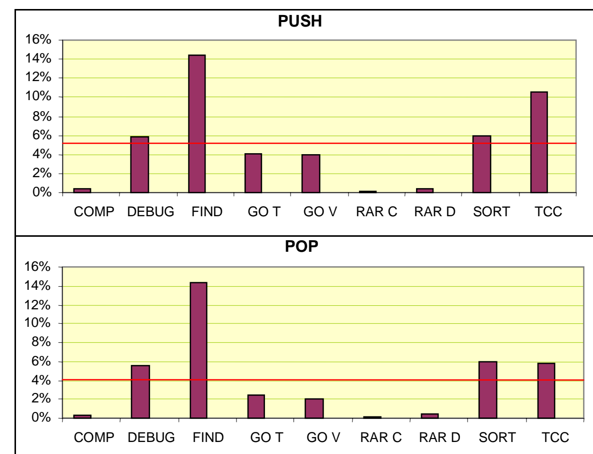

Fig. 1. Stack traffic comparative.

[1] In many occasions an optimal allocation of registers produces the same result that to count on a larger number of registers.

[2] Explicit data is considered which appears in the instruction.

[3] Implicit data is considered which does not appear in the instruction format since it is associated to operation code.

Table 1. Listing of the top 25 more frequently used operations in average: to the left according to the article of Adams and Zimmerman [1] and to the right according to the counts obtained in this report.

	MEDIA (trabajo previo)		
	operación	**%**	**acumulado**
1	MOV	29,95	29,95
2	PUSH	9,25	39,2
3	CMP	7,94	47,14
4	POP	6,22	53,36
5	JNE/JNZ	4,52	57,88
6	JE/JZ	3,63	61,51
7	ADD	3,47	64,98
8	CALL	3,29	68,27
9	RET	3,17	71,44
10	JMP	2,3	73,74
11	LOOP	1,96	75,7
12	INC	1,95	77,65
13	OR	1,84	79,49
14	SUB	1,74	81,23
15	SHL/SAL	1,38	82,61
16	XOR	1,17	83,78
17	DEC	1,17	84,95
18	LES	1,13	86,08
19	TEST	1,04	87,12
20	JNB/JAE	0,84	87,96
21	LDS	0,74	88,7
22	SHR	0,75	89,45
23	RCR	0,72	90,17
24	RETF	0,69	90,86
25	JNBE	0,66	91,52

	MEDIA (resultados actuales)		
	operación	**%**	**acumulado**
1	MOV	30,36	30,36
2	CMP	8,31	38,67
3	JE/JZ	5,85	44,52
4	MOVS	5,45	49,97
5	PUSH	5,10	55,07
6	ADD	4,71	59,77
7	INC	4,24	64,01
8	POP	4,12	68,13
9	JNE/JNZ	3,53	71,66
10	JMP	3,17	74,83
11	JNB/JAE	2,28	77,11
12	SHL/SAL	1,99	79,10
13	SUB	1,97	81,07
14	XOR	1,77	82,83
15	DEC	1,47	84,30
16	JB/JNAE	1,36	85,67
17	CALL	1,22	86,89
18	OR	1,19	88,08
19	AND	1,01	89,09
20	SCAS	1,01	90,09
21	STOS	1,00	91,09
22	TEST	0,98	92,07
23	RET	0,87	92,94
24	SHR	0,66	93,60
25	CLC	0,59	94,19

As far as the transferences with the stack are concern, we see how FIND stands up by amount and TCC by the disparity between PUSH and POP operations. The case of FIND denotes limited temporary storage resources which imply saving in the stack temporary data with the aim to change their use. This agrees with the dedicated use of the registers that in later sections is described. With respect to TCC, the difference of percentage between PUSH and POP is justified in attention to two diverse uses. The percentage of POP use and the equivalent one of PUSH make reference to a limited register file. The excess of use of PUSH with respect to POP points the arguments passing to procedures through the stack which does not have pair in returns with POP.

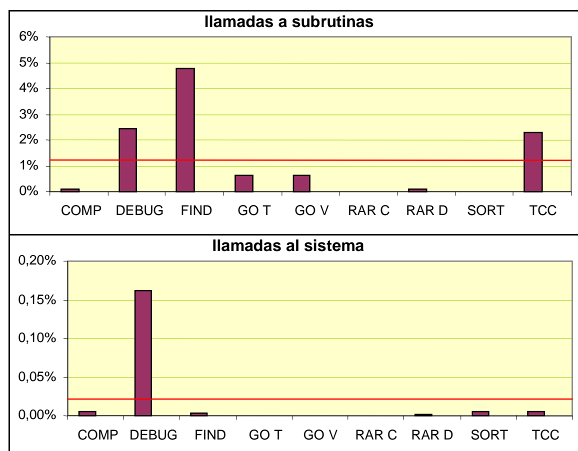

Fig. 2. Procedure calls comparative.

As far as the procedure calls, we observed an absolutely heterogeneous behaviour and the anomaly of DEBUG in the system calls.

The program that more procedure calls performs is FIND although that does not go with an important argument passing, whereas TCC passes an important amount of arguments and COMP even larger. Later, when we talk about the “explicit use of registers”, we will have occasion to describe this aspect detailed.

The amount of system calls that DEBUG makes justifies by the fact that it is the program which more data display in the screen.

Finally, two more graphs plot the results of sequential performance and basic block size. We can advance to a correlation between CPI and memory accesses (they represent a bottle-neck) so that the worse CPI, the one of COMP, corresponds with the greater percentage of memory accesses, especially over the average.

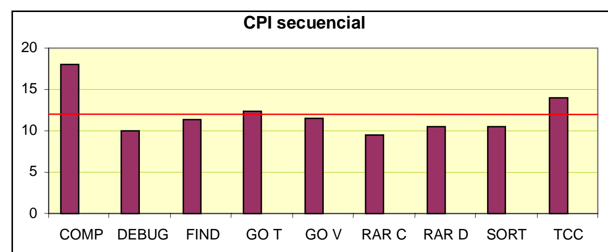

Fig. 3. Sequential CPI performance comparative.

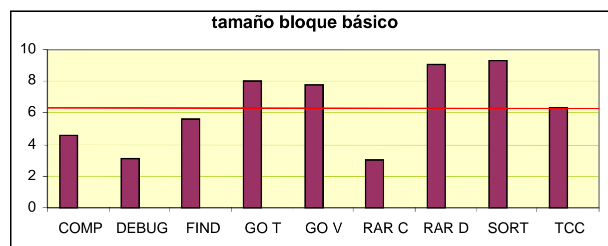

Fig. 4. Basic block comparative.

Table 2. Listing of the 25 operations more frequently used by program.

	COMP		**DEBUG**		**FIND**		**GO T**		**GO V**		**RAR C**		**RAR D**		**SORT**		**TCC**	
	operación	%	operación	%	operación	%	operación	%	operación	%	operación	%	operación	%	operación	%	operación	%
1	MOV	42,26	JE/JZ	12,33	MOV	16,16	MOV	49,66	MOV	49,38	MOV	21,71	MOV	39,01	MOVS	44,75	MOV	33,87
2	CMP	16,07	CMP	10,49	PUSH	14,39	ADD	12,20	ADD	12,56	JE/JZ	19,90	XOR	8,95	MOV	11,76	PUSH	10,64
3	INC	15,84	JNE/JNZ	10,38	POP	14,39	CMP	9,34	CMP	8,80	SHL/SAL	12,72	SUB	6,23	PUSH	6,00	CMP	7,29
4	JNB/JAE	10,24	MOV	9,45	CMP	4,84	PUSH	4,14	JNE/JNZ	4,77	CMP	10,23	CMP	4,56	POP	6,00	POP	5,78
5	JE/JZ	5,65	JMP	6,60	JMP	4,82	JNE/JNZ	3,87	PUSH	3,98	JB/JNAE	5,77	ADD	4,47	ADD	4,50	JE/JZ	3,99
6	JMP	5,31	PUSH	5,80	CALL	4,77	JE/JZ	2,88	JMP	2,97	XOR	3,73	SHR	4,38	XLAT	3,22	JNE/JNZ	3,79
7	SCAS	0,66	INC	5,72	RET	4,77	JMP	2,80	AND	2,49	DEC	3,49	STOS	3,72	CMP	3,15	INC	3,66
8	STOS	0,65	POP	5,62	INC	4,76	POP	2,50	JE/JZ	2,07	TEST	2,45	SHL/SAL	3,56	SUB	3,05	JMP	3,47
9	JNE/JNZ	0,60	OR	5,28	CLC	4,73	AND	2,46	POP	2,05	JNB/JAE	2,33	MOVS	3,43	JNBE/JA	2,93	CALL	2,30
10	OR	0,35	DEC	3,91	SCAS	3,42	INC	1,88	INC	1,94	ADD	2,23	JNB/JAE	1,94	SCAS	1,89	OR	2,10
11	PUSH	0,34	TEST	3,49	JNE/JNZ	3,33	LES	1,46	LES	1,77	JNE/JNZ	2,17	AND	1,83	INC	1,71	RETF	1,74
12	LODS	0,31	ADD	3,06	JE/JZ	3,22	JNLE/JG	0,91	JNLE/JG	0,90	CMPS	2,00	INC	1,82	LODS	1,61	ADD	1,63
13	SUB	0,25	STOS	2,50	SUB	3,19	SUB	0,87	OR	0,85	SUB	1,75	DEC	1,59	LOOPZ	1,61	DEC	1,59
14	POP	0,21	CALL	2,42	JNB/JAE	3,17	JL/JNGE	0,86	JL/JNGE	0,77	AND	1,56	JMP	1,56	JNE/JNZ	1,57	SCAS	1,57
15	DEC	0,21	RET	2,30	JCXZ	1,70	CALL	0,64	JNL/JGE	0,72	SHR	1,21	LOOP	1,50	JE/JZ	1,50	SUB	1,53
16	LES	0,16	XOR	1,84	DEC	1,70	RETF	0,64	CALL	0,65	JMP	0,86	XCHG	1,42	OR	1,48	LES	1,41
17	ADD	0,14	SCAS	1,46	TEST	1,62	JNL/JGE	0,58	RETF	0,65	INC	0,80	JNE/JNZ	1,33	JB/JNAE	1,47	STOS	1,37
18	MOVS	0,11	JB/JNAE	1,35	JB/JNAE	1,60	JLE/JNG	0,56	JLE/JNG	0,58	XCHG	0,67	JNBE/JA	1,25	JNB/JAE	1,47	JB/JNAE	1,31
19	LOOP	0,09	JNB/JAE	0,92	LODS	1,60	DEC	0,35	SUB	0,54	LOOP	0,62	JS	1,15	JMP	0,13	LOOP	1,18
20	JS	0,09	XCHG	0,56	ADD	1,57	LEA	0,25	DEC	0,35	STOS	0,60	JE/JZ	1,10	DEC	0,06	SHL/SAL	1,08
21	CALL	0,08	CLC	0,53	MOVS	0,10	SHL/SAL	0,24	SHL/SAL	0,24	JLE/JNG	0,33	SBB	1,04	STOS	0,05	XOR	0,99
22	RET	0,08	DIV	0,32	STOS	0,08	IMUL	0,22	IMUL	0,23	OR	0,32	JB/JNAE	0,72	JCXZ	0,04	TEST	0,90
23	XCHG	0,04	STC	0,30	STC	0,03	XOR	0,17	LEA	0,22	NOT	0,32	ADC	0,60	CLD	0,02	CBW	0,86
24	CBW	0,03	SUB	0,29	LOOP	0,01	TEST	0,16	XOR	0,21	LDS	0,32	JBE/JNA	0,46	SHR	0,02	LODS	0,86
25	SHL/SAL	0,02	JBE/JNA	0,28	XOR	0,00	OR	0,16	TEST	0,17	JNBE/JA	0,31	PUSH	0,40	STD	0,02	RET	0,55

a. Use of instructions outside the instruction set of microprocessor *Intel* 8086

As the used programs to generate the traces have not been compiled (the case of GO is an exception) because the source code was not available, a count of instructions not including in the instruction set of microprocessor *Intel* 8086 has been made.

The result is that none of the test-bench programs use later extensions excepting the compiler TCC which does it approximately in a 1% of occasions. This agrees with the same measurements made by Adams and Zimmerman [1] but remarking that our programs are more modern, as they correspond to later versions, and therefore more susceptible of having incorporated such extensions.

The conclusion is that the extensions added to the instruction ser of microprocessor 8086, specially to the 80386, are just applied to writing determined parts of the operating systems (virtual memory, memory protection, etc.) that are not in the ordinary applications.

b. Use of prefix codes

On the other hand, we are going to mention the use of prefix codes briefly. In the x86 instruction set we have prefix codes for string operation repetition, segment prefix codes of and LOCK prefix.

The repetition prefix codes are used to modify the string instructions in the way they are repeated as if they were inside a loop. It would be a loop with a single instruction, the string operation[4]. The traces have an average of 2.62% with the exception of SORT that makes an intensive use of these prefixes reaching 46.68%.

The segment prefix codes are used to modify the memory base register used by default to calculate the effective memory address. They concern, therefore, with the memory accesses. We notice that the base register establishes an additional dependence among instructions, which the prefix does not do more than to turn explicit. The segment prefixes are used with a frequency average of something more than 10% having distributed CS and ES the whole occurrences with a 4.25% and a 5.94% respectively.

Each memory access implies the use of a segment register by default. The appearance of a segment prefix in the code does not alter this fact, only turns explicit what of natural way is implicit.

The prefix LOCK, which function is to block the access to shared hardware resources by several processors, has not been found in any occasion.

[4] The repetition prefix codes hold the same problem as the LOOP instruction: the counter register CX has a dedicated use. By its functionality, to nest a string instruction in other loop is not going to be habitual and then to make transferences with the stack to save the counter is not necessary… Nevertheless, in many occasions, to enclose inside a loop several string sentences would be more effective than repeat one after other but it is not possible. The use of these strings operations imposes the distribution of the process in several implicit loops, each one of them with its own repetition prefix. Notice that the repetition prefix code for string operations gives a basic block size one.

3. Detailed study of data access

In superscalar processing, the final performance depends on several aspects: the programmed algorithm, the compiler behaviour, the architecture limitations and, finally, the instruction mix which the application has been implemented with since limited concurrent execution is due, obviously, to data dependences among instructions. In this sense, it becomes interesting to learn how data are acceded to since it is going to provide us information about the limitations that the instruction set architecture imposes.

Next, the addressing modes distribution in three different comparatives is shown graphically: data allocated in explicit registers, accesses to data located in memory and operations among registers.

We see as the COMP trace has the larger percentage of memory accesses and the minor of operations among registers. Evidently its CPI is the worse one as consequence, by far, of the memory accesses latency. The best one is the RAR compressing, with a similar use of operations among registers and memory accesses.

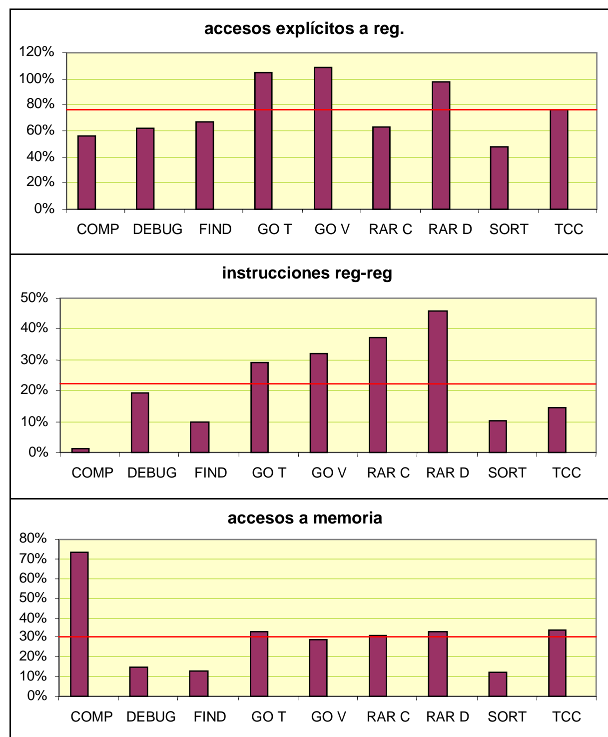

Fig. 5. Data allocation comparatives.

We want to know how the registers are used under the direct addressing mode to register, also we want to know how the memory is acceded and how the effective addresses are calculated and, finally, we cannot forget that there is an implicit use of data what we want to evaluate its impact on the final performance of the machine since also it is responsible for data dependences. We go, then, to look at the problem counting explicit use of registers both in direct access and involved in memory addresses and,

afterwards, we will study the counts of implicit use of registers.

This study is going to sketch the scene of the potential data dependences and, consequently, the disposition that we are going to have to take advantage of the concurrent processing.

a. Explicit use of registers

The register use has a double functionality: the data processing and the address computing. The address computing, both for data in memory or stack access, represents a computational load although it is not strictly associated to the programmed algorithm.

Figure 6 illustrate the number of register accesses to. In light gray we have the amount of accesses in direct addressing mode to register, that is, those used for the data processing. In dark gray the accesses to registers for computing memory addresses (addressing modes relative to register) have been accumulatively represented. The readings have been presented. As the instruction set format comprises just two directions, the destination operand (written) is also read. We try to describe the map of accesses more than if they are used in reading or writing access.

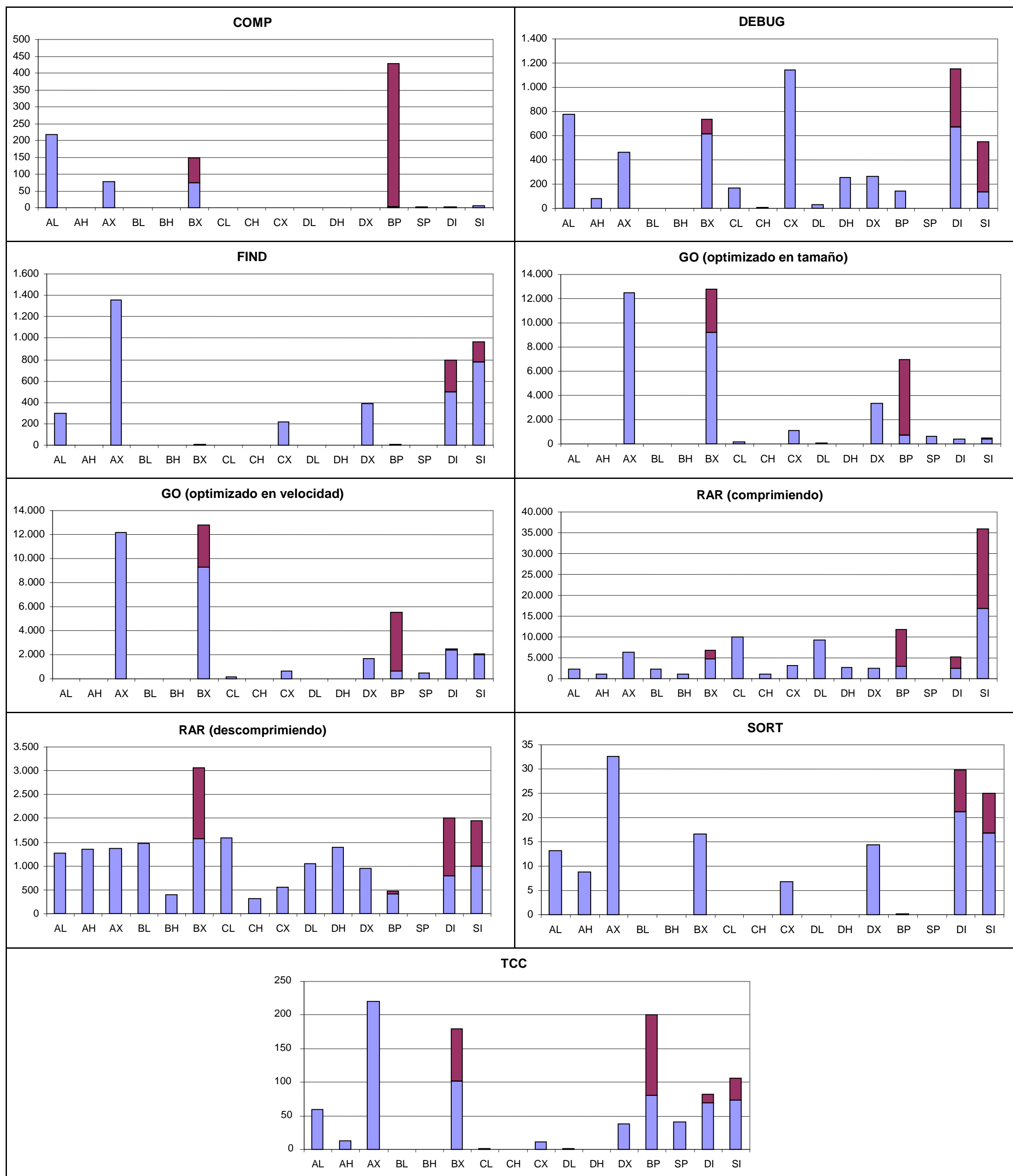

Fig. 6. Explicit access to registers in thousands of references. Dark gray for references used in memory addressing.

He is flashy to observe that the use of registers is concentrated in a few of then in the most cases, specially in COMP, FIND and the two versions of GO. This is more evident in the use of registers as memory pointers. We see that the variety of used registers to address computation is still smaller. This lack of versatility is clear in the grid of memory codification of addressing modes who we present next:

Table 3. Effective memory addresses computation in the instruction format.

r/m	mod = 00	mod = 01	mod = 10
000	[BX]+[SI]	[BX]+[SI]+D8	[BX]+[SI]+D16
001	[BX]+[DI]	[BX]+[DI]+D8	[BX]+[DI]+D16
010	[BP]+[SI]	[BP]+[SI]+D8	[BP]+[SI]+D16
011	[BP]+[DI]	[BP]+[DI]+D8	[BP]+[DI]+D16
100	[SI]	[SI]+D8	[SI]+D16
101	[DI]	[DI]+D8	[DI]+D16
110	dirección directa	[BP]+D8	[BP]+D16
111	[BX]	[BX]+D8	[BX]+D16

The fact that the previous table has three different columns with displacement 0, displacement of 8 bits and displacement of 16 bits is a consequence of instruction set design criterion: as it is tried to reduce the representation space and, therefore, the size of the formats is variable based on the amount of required information. In this case, the format codification must indicate the number of additional bytes that should be taken from the instruction flow to read the displacement. Thus we assured that small or null displacements do not occupy memory unnecessarily.

Really, the previous table could be reduced to Table 4 except the case of the direct address to memory. The segment register used as base has been included in each case.

Table 4. Registers evolved in memory addresses computation. The default segment register is enclosed.

DSx16+[BX]+[SI]
DSx16+ [BX]+[DI]
SSx16+ [BP]+[SI]
SSx16+ [BP]+[DI]
DSx16+ [SI]
DSx16+ [DI]
SSx16+ [BP]
DSx16+ [BX]

Only four registers are used to address computation: BX, BP as bases and SI, DI as indexes, according to the terminology of the manufacturer. To these registers it is necessary to add the implicit use (unless a segment prefix is specified explicitly) of a segment register by defect: DS or SS depending on the base and the index.

The high record number involved in the address computation is a potential source of data dependences.

In Fig. 7 is plotted the distribution of registers implied in the address computation over the total memory accesses for each test-bench program and the average on all of them in the rightmost column.

It is possible to think that the normal is to use a register of the four. Nevertheless, program RAR, as much in compression as in decompression, uses two registers (one of bases and one of the indexes) in an important percentage of the accesses. This implies to increase the potential data dependences that, in the end, limit the parallelism degree and therefore the performance. Let observe that the graph makes reference to the registers used as displacement. It is necessary to add, in each case, the segment register that has been used as base. Really, it is necessary to count one more register in each case.

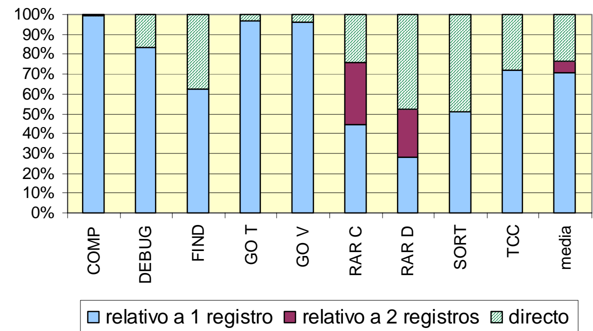

Fig. 7. Memory addressing modes distribution (considering just offset registers).

At the same time, we must indicate that the combined accesses to registers in word size and byte size (AX, AL, AH, etc.) also are potential sources of data dependences since they represent the same physical resource.

The trace of program COMP is the one that less registers uses: AX, BX and BP. The storage cell is used in direct address, register BP in addresses to memory (stack) and the BX is distributed for both functions. It is easy to conclude that the graph del the code is going to present/display many dependences through these three registers being limited the opportunities of concurrent execution.

The dedicated use of registers and the high reusability of them imply a greater degree of dependences among data. In fact, it sketches a scene of lack of physical resources, although in absolute terms we pruned to count with an appreciable amount of temporary storage elements. In addition, as Adams and Zimmerman indicate [1], this limitation elevates the number of occurrences of instructions MOV, PUSH and POP.

Some authors have pointed that the data dependences due to memory pointers are still more useless that the transformations of data resident in registers [4, 2]. The idea is that the second ones derive from the semantics of the program (what algorithm programmed performs) whereas first ones are artificial, they are due to the programming model, to the limitation in physical resources and, in addition, they generate double dependences: through the pointer registers and through the own memory considered as a solely resource. There is a load of programming, added to the own programmed algorithm task, that is in charge to execute code to properly update the memory pointers.

b. Implicit use of registers

The x86 instruction set architecture works with implicit operands associate to the operation code that do not appear, therefore, in the format. It is, also, a way to save representation space. However, that the programmer (the compiler) does not express them does not mean that they do not generate data dependences. The implicit operands involve, in addition, a dedicated use of registers that can aggravate the problem to find independent operations to execute in concurrent environment. Also, it increases the amount of stack swapping.

In Fig. 8 are presented the graphs that plot the distribution of the implicit use of registers for the different test-bench programs.

The referenced implicit registers are always the same: accumulator (AX), counter (CX), SP, DI, SI and very sporadic occurrences on BX and DX.

The number of implicit accesses surpasses in some occasions to explicit accesses. So it is the case of DEBUG, FIND and SORT.

The accesses to the stack pointer (SP) come from instructions of stack managing: PUSH, POP, CALL, RET. The work of Postiff [4] identifies this fact and analyzes its consequences.

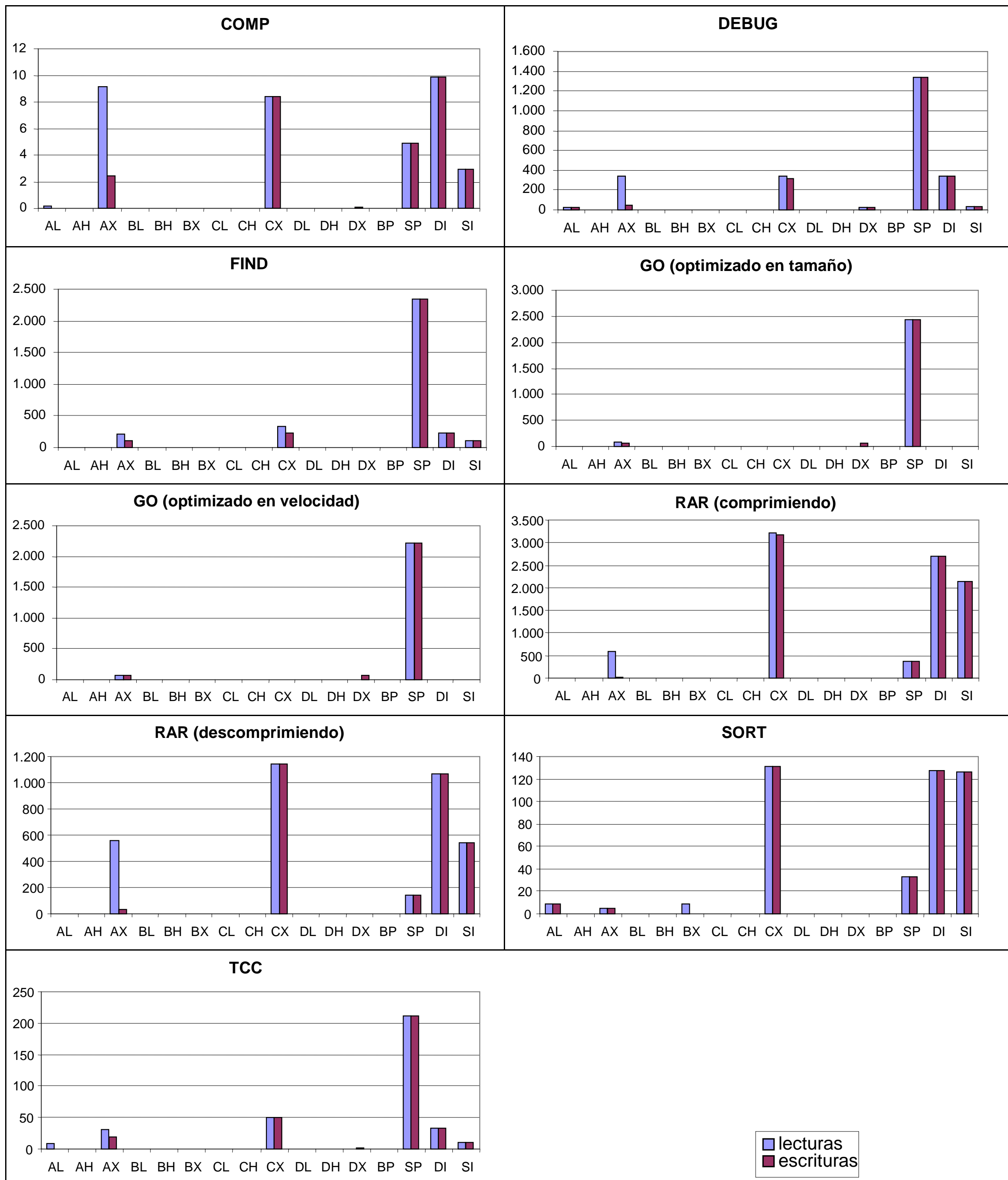

Fig. 8. Implicit use of registers expressed in thousands of references. Dark gray for writes and light gray for reads.

In the graphs, the segment registers, which always appear in the effective memory address computation, are not included. Its total amount is equal to the one of memory accesses and its distribution has to do with the default register given in Table 4 or specified by segment prefix, in its case, in a explicit way with the purpose of modifying the established one by default. However, in this work each occurrence has not been considered in detail.

In general, the number of reads and writes is practically equal over all the registers with the exception of accumulator that usually has less writes than reads. This fact contributes to the appearance of long chains of dependences.

The rest of registers used implicitly in our traces came mainly from strings operations. LOOP also generates implicit accesses to the counter register (CX) but the amount attributable to this instruction is very small since its use is very small and it is not among the top 25 more used in average.

The SORT trace highlights because the implicit accesses are far beyond the explicit ones (1,150,000 as opposed to approximately 130,000) due to the significant use of instructions of strings managing. Employing these operation codes, that in fact correspond to loop primitives, prevents to carry out optimizations with the purpose of taking advantage of potential parallel resources.

c. Implicit use of status flags

This instruction set is a clear representative of the architectures based on status register. In them, the conditional branches are evaluated based on the value that contains a special register constituted by a series of fields that store information on different situations. These fields are updated by some instructions in a implicit and unconditional way.

Figure 9 shows the distribution use of status flags by traces. The reads and writes of the bits corresponding to which *Intel* calls status flags have been indicated. The manufacturer distinguishes in this register between status flags, updated by process instructions, and control flags, governed by the programmer to settle down different operation modes. We are interested in the first ones as they straight depend on the execution of certain operation codes.

The write access to the status flags is specially important in the case of the process instructions whereas the read access usually correspond to the conditional branch instructions. Some process instructions read the flags as a more data input.

From the graphs plotted next, it is clear that the writes become in block whereas the reads are made over specific fields (bits). For example, the COMP trace only reads the ZF and CF flags whereas it writes in block in all the status flags (with the exception of CF); DEBUG, FIND, RAR and TCC behave of similar way. The two traces of GO also adjust to the saying: they write in block but they only read OF, SF and the ZF flags.

This way to operate is absolutely reasonable. The idea is that in each basic block we have a branch. This branch is mainly a conditional branch that evaluates a condition expressed by a status flag, sometimes a combination of two of them and very rare times three. Consequently, the execution of the bifurcation reads some specific flags, not the whole block. Nevertheless, the process instructions write the status modifying most of the status bits.

Let notice another fact. The amount of writes surpasses, in most cases, to the reads. The trace of SORT, as result of using repetition prefix over string instruction, is the only one which leaves the rule —these instructions again take out this program to the average norm—.

There is, therefore, an imbalance between the generated information and the required information both in extension and in amount. There is a disparity in extension because the written flags are more than necessary. In amount because the status writes surpass three or four times to the reads.

What is writes amount larger than read ones due to? Let us return to the basic block. In each one of them we have, in average, more than a process instruction (which write status in block) by a single conditional branch. The conditional branch only considers the last status update causing that the previous writes became absolutely useless. They not only are unproductive but generate data dependences. If the data dependences deal with each flag as an individual data symbol or resource, the resulting data dependence graph has many arcs (data dependences) and consequently is much coupled.

A very large number of writes on the same resource entails a potential increase of the output dependences. It is truth that the output dependences, as well as the antidependences, are not of the “true ones”, those that have computational sense, but the renaming technique to avoid then supposes to have an additional status register file.

We are going to examine our traces counting how many average process operations there are by basic block as a way to measure the degree of limitation to the parallelism imposed by the instruction set architecture and, therefore, no attributable to the program semantics. In the graph associate (Fig. 10) can be seen that the number of process operations and, therefore, the number of writes in the status register, by basic block surpasses to the read ones. This suggests us to think that many of them are superfluous. In that graph, SORT has not been plotted because, having a very high percentage of string instructions, cannot treat, with respect to the basic block, as to the rest of traces.

Fig. 9. Implicit use of status flags expressed in thousands of references. Dark gray for writes and light gray for reads.

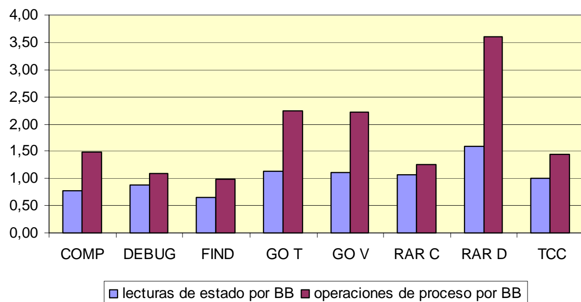

Fig. 10. Distribution of status reads and process operations (status writes) by basic block.

Let explain that all these dependences are not superfluous because they can be superposed to the real ones (semantic) generated by the data processing. Nevertheless, if we avoided them we can obtain the available parallelism upper bound. The degree of real parallelism will move between the present situation and the upper bound.

The description done till here explains the degree of data dependences in a direct way. However, the complete effect can be diminished by means of disambiguating techniques: register renaming and

memory fundamentally [7, 6, 2, 4]. The application of these techniques is simpler in the case of the processor registers than in the case of the memory. Also, it is simpler in the case of the explicit registers that in the case of the implicit ones and it becomes very complicated in the case of the status register.

4. 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] 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.

[3] R. Hyde. *The Art of Assembly Language Programming*. Versión borrador. 2001. Available at: http://webster.cs.ucr.edu

[4] 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.

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

[6] 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.

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