ONE-CYCLE ZERO-OFFSET LOADS

E. MORANCHO, J.M. LLABERÍA, À. OLIVÉ, M. JIMÉNEZ
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
enricm@ac.upc.es
Fax: +34 - 93 401 70 55

Abstract

Memory instructions account for a significant portion of stall cycles on integer codes. Analysing several codes, we have noticed the existence of an important number of zero-offset memory instructions. This work proposes two techniques to reduce the stall cycles produced by zero-offset memory instructions: address-calculation collapsing and zero-offset load/store advancing. The evaluation of these proposals shows that the achieved improvements are not uniform; IPC improvement ranges from 2% to 13%. This work relates these improvements with the data structures used by the programs and their traverses.

Keywords: integer codes, stall cycles, zero-offset loads/stores, reduce latency, tolerate latency

1. INTRODUCTION

Stall cycles caused by data dependencies account for a significant portion of execution cycles on integer-intensive codes. These cycles can be decreased reducing the latency of the functional units or scheduling independent work between the execution of two dependent instructions. Instruction scheduling can be done by a compiler and/or by an out-of-order processor; in both cases, these techniques look for instruction-level parallelism, and their goal is to tolerate the latency of functional units.

Current trends on microprocessor design increase the number of instructions issued per cycle, requiring a significant portion of the available instruction-level parallelism, and reducing the remaining parallelism for tolerating latencies. Moreover, increasing the issuing width implies that more instructions are needed to tolerate the latency of the functional units. Consequently, techniques that reduce the latency of the functional units must be developed to improve the performance of future processors.

The basic-block size of integer-intensive codes is short, and the instruction-level parallelism is small [1]. In this codes, load instructions cause a significant portion of stall cycles produced by data dependencies because, usually (on a load hit), the latency of a load instruction is two cycles (the first one to evaluate the effective address, and the second one to access data cache), meanwhile, the latency of the arithmetic instructions is one cycle.

Different methods for reducing the effects of memory dependencies have been proposed. A technique for tolerating load instructions latency by modifying the pipeline organization is proposed in [2]. On the other hand, [3][4][5] reduce and/or tolerate load latency by predicting the effective address of the memory references early in the pipeline and accessing speculatively data cache. A more aggressive approach is proposed in [6], it executes speculatively instructions that depend on a predicted load.

Analysing integer codes it is appreciable that used data structures, their traverses, and the addressing modes used by the compiler determine the existence of an important number of zero-offset memory instructions. This work takes advantage of the equality between the base address and the effective address of a zero-offset memory instruction. We present a technique to reduce the latency of zero-offset load instructions, and another one to tolerate the latency of the instructions computing the base address of zero-offset loads/stores.

The remainder of this paper is organized as follows. Section 2 presents a characterization of integer programs. Section 3 details the execution-cycle distribution of the selected benchmarks. In Section 4 our proposes are described. An evaluation of the proposed techniques is shown in Section 5. Section 6 reviews related work. Finally, Section 7 presents our conclusions.

2. PROGRAM CHARACTERIZATION

To evaluate the proposals of this work, we have selected the integer-intensive codes from the SPEC-95 benchmark suite and some pointer-intensive codes used by T. Austin [7].

Table 1 details the selected codes, the main logical data structure (the one used by the programmer), the storage structures offered by the language used by the programmer, and reference counts of executed load instructions (ld), zero-offset loads (ld0), stores (st), and zero-offset stores (st0). Benchmarks were run in an Alpha processor.

Analysed programs execute an important number of zero-offset loads and stores; it ranges from 9% to 21% of the total number of executed instructions, and from 27% to 61% of the memory references. This fact is explainable considering the logical data structures, their implementation, and the available addressing modes.

Benchmark	Logical data structure	Used language structures	ld	ld0	st	st0
go	Lists	Vectors	27.7	16.2	7.6	3.7
mksim	Vectors	Record of vectors	24.3	6.6	8.4	2.4
gcc	Tree	Pointers to records	21.7	8.3	11.6	3.1
compress	Hash table	Vector	19.1	13.2	11.5	6.5
li	Tree	Pointers to records	22.9	6.3	13.4	5.3
ijpeg	Matrixes	Vectors	18.0	7.9	7.2	3.6
perl	Tree	Pointers to records	24.2	9.3	12.1	3.8
vortex	Lists	Pointers to records of pointers to vectors	24.4	9.3	15.9	3.1
anagram	Vector of records	Vector of pointers to records	23.8	13.8	9.0	4.8
bc	Tree	Pointers to records	19.7	7.8	10.2	4.0
ks	Graph	Vectors and records linked by pointers	37.7	21.6	0.7	0.2

Table 1: Benchmarks, logical data structures, used language structures, and reference counts of loads (ld), zero offset loads (ld0), stores (st) and zero offset stores (st0) . Reference counts represented as the percent of all executed instructions.

In integer-intensive codes, the most used logical data structures are lists, trees and randomly acceded vectors and matrixes. Lists and trees can be implemented with vectors or with linked records. For these implementations the compiler generates specific patterns of instructions and uses some addressing modes for accessing data structures. Figure 1 shows a fragment of high-level code that traverses a single-linked list using both implementations and details the code generated by an Alpha-AXP compiler.

Case a) shows a list implemented using two vectors: vector `data` stores the list elements, and vector `links` stores the index to the next list element for every one of them. To access the index of the next list element (`links[ptr]`), the compiler generates an arithmetic instruction to compute the associated address and a memory access instruction. The arithmetic instruction (`s4add`) uses the base address of vector `links` and the index value (register `a0` stores it) to compute the sum of the base address and the index shifted by the element size (four bytes). The memory access is a zero-offset load instruction; it obtains the index of the next list element. To get the list element (`data[ptr]`), the compiler generates analogous code. Equivalent code appears in every subscript reference. If the architectural instruction set does not have an instruction like `s4add`, the compiler will generate code to calculate the address using one shift and one sum instructions.

Case b) shows a list implemented using records linked by memory pointers. The compiler generates only one instruction to access the address of the next list element (`node->next`), a zero-offset load, because `next` is the first field of the record. To access the data (`node->data`) needs a nonzero-offset load, because this field is placed with a nonzero-offset displacement.

If the compiler generates code to traverse a tree where the nodes are linked with pointers, and every node has several pointers to his subtrees, at most one of these pointers will be stored with a zero-offset displacement. The remaining will be stored with a nonzero displacement.

a)

```
int links[MAX];
int data[MAX];

ptr = links[ptr];
element= data[ptr];
```

```
/* a0 stores ptr */

s4add a0, _links, a1
ld    a0, 0(a1)
s4add a0, _data, a2
ld    a3, 0(a2)
```

b)

```
struct{
  struct node *next;
  int data;
} *node;

node = node->next;
element = node->data;
```

```
/* a0 stores node */

ld a0, 0(a0)
ld a1, 8(a0)
```

Figure 1: **High-level code and assembly generated to traverse a list implemented using vectors (case a) and using memory pointers (case b).**

The used data structures and their most accessed fields influence on the number of executed zero-offset loads. Table 1 shows that, usually, benchmarks that use vectors (*go*, *compress*, *anagram*) have more zero-offset loads than programs that use pointers to implement trees (*li*, *gcc*, *perl*, *bc*). There are two exceptions: a) *ks* uses pointers to implement lists (and these lists to implement graphs), but the pointer field is placed with a zero-offset displacement, so the compiler generates a zero-offset load to traverse the graph; b) *m88ksim* uses vectors but the most executed routine loops to access all the vector elements sequentially, so the compiler unrolls the loop and generates nonzero-offset loads.

2.1. CHARACTERIZATION OF MEMORY-INSTRUCTION INTERLOCKS

Classifying stall cycles where memory instructions are involved will allow us a deeper knowledge of their cause, influence and the effectiveness of proposed solutions.

Memory instructions cause the following dependencies:

- The computation of the base register for a load/store determines that this memory instruction is destination of a dependence called AGD (Address-Generation Dependence).
- The use of the value retrieved by a load instruction produces that this load is source of a dependence called LUD (Load-Use Dependence).

One of these dependencies detected between instructions in execution produces a hazard. In an in-order processor these hazards can generate interlocks. Interlocks produced by AGD dependencies will be called AGI (Address-Generation Interlock), and interlocks produced by LUD dependencies will be called LUI (Load-Use Interlock). The number of stall cycles caused by a dependence is determined by several points: the distance between the source and the destination instruction, the number of instructions issued per cycle and the functional-unit latencies.

The influence of zero-offset loads in interlocks can be approached by the dynamic measure of the distance from the source instruction to a zero-offset load (AGD), and the distance from the load to the first instruction that uses the accessed data (LUD). The dynamic distance between two

instructions is defined as the number of groups of contiguous and aligned instructions between them. Measures on Alpha code with groups of two and four instructions are presented, where groups are aligned to addresses multiples of two and four instructions respectively. Table 2 shows the distance distribution for benchmarks *go* and *anagram*

Tables are trisected taking into consideration which instruction class generates the AGD: arithmetic (Arit→Ld0), nonzero-offset load (LdX→Ld0), and zero-offset load (Ld0→Ld0). For example, with groups of two instructions on benchmark *go* (Table 2.a), the 22.3% of zero-offset loads are at the same group of instructions where is located the arithmetic instruction that computes it base address (AGD=0), and one group before the first instruction that uses the value retrieved by the load (LUD=1).

Table 2 also shows the stall cycles produced in an in-order processor. If arithmetic-instruction latency is one cycle and load-instruction latency is two cycles, the bottom cell of each column indicates the amount of stall cycles (AGI Cycles) generated by the AGD related to this column. The rightmost cell of each row details the number of stall cycles (LUI Cycles) produced by the LUD related to this row. Every zero-offset load of the previous example (loads where AGD=0 and LUD=1) will cause two stall cycles: one produced by the AGD, and another one motivated by the LUD.

Distance-distribution tables (Table 2) reflect the data structures used by the programs (Table 1). In *go* benchmark, the source instruction of nearly every zero-offset load is an arithmetic one because *go* mainly uses vectors. In *anagram* code (Table 2.b), the source instruction of many zero-offset loads (about 30%) is another memory reference because the program uses pointers. Remaining codes have intermediate behaviours.

As can be expected, measures with groups of four instructions detail shorter distances than measures with groups of two instructions. For instance, using groups of two instructions in benchmark *go* (Table 2.a), the 23.3% of zero-offset loads are placed at the same group where is placed the arithmetic instruction that generates the AGD. Using groups of four instructions (Table 2.c) this percentage is increased to the 52.4%.

		AGD								
		Arit→Ld0		*LdX→Ld0*			*Ld0→Ld0*			
		0	*>0*	*0*	*1*	*>1*	*0*	*1*	*>1*	LUI Cycles
LUD	*0*		31.3		0.4	0.4				*2*
	1	22.3	31.4	0.2	0.1	0.8				*1*
	>1	1.0	10.8		0.1	0.5				*0*
AGI Cycles		*1*	*0*	*2*	*1*	*0*	*2*	*1*	*0*	

Table 2.a: Groups of 2 instructions in *go* code (13 millions of executed zero-offset loads).

		AGD								
		Arit→Ld0		*LdX→Ld0*			*Ld0→Ld0*			
		0	*>0*	*0*	*1*	*>1*	*0*	*1*	*>1*	LUI Cycles
LUD	*0*		14.9			6.0				*2*
	1	9.6	20.0		6.8	3.2		7.9		*1*
	>1	2.7	24.5			4.5				*0*
AGI Cycles		*1*	*0*	*2*	*1*	*0*	*2*	*1*	*0*	

Table 2.b: Groups of 2 instructions in *anagram* code (3 millions of executed zero-offset loads).

		AGD								
		Arit→Ld0		*LdX→Ld0*			*Ld0→Ld0*			
		0	*>0*	*0*	*1*	*>1*	*0*	*1*	*>1*	LUI Cycles
LUD	*0*	27.4	28.7	0.3	0.6	0.4				*2*
	1	23.4	11.9	0.1	0.3	0.6				*1*
	>1	1.6	3.7		0.1	0.3				*0*
AGI Cycles		*1*	*0*	*2*	*1*	*0*	*2*	*1*	*0*	

Table 2.c: Groups of 4 instructions in *go* code (13 millions of executed zero-offset loads)

		AGD								
		Arit→Ld0		*LdX→Ld0*			*Ld0→Ld0*			
		0	*>0*	*0*	*1*	*>1*	*0*	*1*	*>1*	LUI Cycles
LUD	*0*	8.3	30.7		6.8	6.1				*2*
	1	3.4	24.1		1.5	6.0	7.9			*1*
	>1	2.7	2.4							*0*
AGI Cycles		*1*	*0*	*2*	*1*	*0*	*2*	*1*	*0*	

Table 2.d: Groups of 4 instructions in *anagram* code (3 millions of executed zero-offset loads)

Table 2: Distance distribution. Cell at column *i*, row *j*, contents the percentage of zero-offset loads of all executed zero-offset loads where AGD=*i*, and LUD=*j* groups (empty cells details that the percentage is null or neglected). Columns and rows tagged >*i* compute all the distances bigger than *i* groups. Shaded cells represent impossible configurations.

3. STALL CYCLES IN BENCHMARKS

This section analyses the influence of stall cycles caused by zero-offset memory instructions on the execution cycles. Table 3 details the stall cycles produced by several factors in a two-way and in a four-way in-order processor (described in Table 4). Columns have these meanings: issue width, millions of execution cycles and the stall cycle distribution (presented as the percent of the total execution cycles).

Stall cycles are classified into four types: 1) Branch (caused by mispredictions and fetch delay), 2) Issue (static issue rules of the processor), 3) Arithmetic (dependencies where source and destination instructions are arithmetic

operations) and 4) Memory (memory system).

Bench.	Way	Cycl. ($\times 10^6$)	Bran.	Issue	Arit.	Memory			
						Cache	Cache hit		
							Zero-offset		Other
							AGI0	LUI0	
go	2	100	8.78	2.40	5.70	19.34	3.39	13.85	4.00
	4	95	9.30	1.76	9.99	20.88	7.48	17.65	5.92
mksim	2	506	4.99	1.87	11.56	10.96	0.81	4.82	6.49
	4	463	5.52	2.47	19.44	12.06	1.63	6.25	15.00
gcc	2	88	6.47	3.81	7.21	18.54	2.03	5.35	6.21
	4	81	7.11	3.96	14.56	20.70	3.31	8.21	9.80
compress	2	65	3.76	3.48	6.57	13.04	2.24	9.81	2.57
	4	57	4.37	2.25	14.60	15.76	7.23	14.92	3.57
li	2	261	4.16	4.36	6.60	18.48	1.46	5.92	9.90
	4	245	4.59	2.87	15.14	22.41	2.26	8.43	14.96
ijpeg	2	340	1.80	1.12	12.32	8.45	0.20	4.94	4.46
	4	306	2.04	2.76	30.50	10.06	0.87	8.74	7.31
perl	2	11	3.63	3.74	9.17	15.33	2.54	7.44	7.28
	4	10	4.09	2.88	17.13	17.46	3.79	11.35	11.16
vortex	2	3227	2.86	4.24	2.70	20.27	1.66	11.02	6.08
	4	2920	3.28	3.14	8.78	25.52	3.11	15.29	8.86
anagram	2	20	2.71	1.16	7.10	14.25	3.86	11.44	4.93
	4	19	2.84	2.28	16.33	15.37	4.66	16.81	7.21
bc	2	17	7.43	3.17	6.38	11.76	1.21	6.34	5.58
	4	15	8.50	2.43	16.19	14.35	2.11	9.85	9.96
ks	2	22	7.24	0.12	1.03	16.93	0.33	12.15	17.76
	4	21	7.46	0.06	1.85	17.79	2.60	15.06	18.96

Table 3: Stall cycles distribution in a two-way (shaded rows) and in a four-way in-order processor; percents of all executed cycles.

Memory stall cycles are divided into two groups: a) Cache (stall cycles originated on misses) and b) Cache hit (assuming references are cache hits). Finally, Cache hit stall cycles are split into 1) AGI-0 (AGI where the destination is a zero-offset load/store), 2) LUI-0 (LUI where the source is a zero-offset load[1]) and 3) Other (AGI or LUI where zero-offset is not involved).

Stall cycles produced by zero-offset memory instructions (assuming hit) represent a high percentage of execution cycles (ranges from 5% to 17% in a two-way processor, and from 9% to 25% in the four way processor). This rate is larger than the stall-cycle percentage produced by branches (saturating counters) and, in most benchmarks, it is similar to the fraction generated by cache system (six-cycle latency on a cache miss), so, it is important to reduce these stall cycles.

The percentage of LUI-0 stall cycles is bigger than AGI-0 because load instructions have longer latencies than arithmetic instructions. Proportionally, comparing two and four way results, increment of AGI-0 stall cycles is bigger than increment of LUI-0 stall cycles. Moreover, the effect of zero-offset load/store instructions in the four-way processor is bigger than in the two-way processor because issuing four instructions per cycle reduces the parallelism for tolerating latencies.

1. Stall cycles caused by a dependence between two zero offset loads are computed as AGI.

In some codes (for instance *li*), the percentage of stall cycles produced by static issue rules in the two-way processor is bigger than in the four-way processor, because the four-way processor can issue at the same cycle one store and another memory instruction (Table 4).

4. PROPOSED TECHNIQUES

The baseline processor uses the typical RISC two pipeline stages [8][9] to execute a load/store instruction: the first one for computing the effective address, and the second one for accessing memory (Section 5 details the pipeline). We propose two techniques to reduce or remove the influence of zero-offset load and store instructions on stall cycles. Which technique is applied rely upon the availability of the base-address value at issue stage:

- Base-address value not available: the base address and the effective address are equal, so it is possible to collapse the source instruction and the zero-offset load/store. This technique reduces the amount of AGI, by tolerating the latency of the instruction that produces the AGD. It will be called *Address-Calculation Collapsing*.
- Base-address value available: After issue stage the processor can access memory. This technique reduces the latency of zero-offset loads and the amount of LUI. It will be called *Zero-offset load advancing*.

4.1. ADDRESS-CALCULATION COLLAPSING (*ACC*)

This technique tries to tolerate the latency of an instruction that computes the base address of a memory reference. It takes advantage of the fact that the base and effective address of a zero-offset load/store are equal, then the computation of the first execution stage is useless.

At I stage, the memory-reference instruction is issued if, at the end of the next cycle, the base-address value will be available. Figure 2 shows the cases where this technique can be applied: the load/store can be issued at the same cycle that the source instruction (case a) or one cycle after (case b). In both cases, *ACC* reduces one interlock cycle produced by the AGD.

At I stage the processor must check: a) the instruction is a zero-offset memory reference, and b) at the next cycle the base-address value will be available. It is necessary to add bypasses from the functional units to the Memory Address Register.

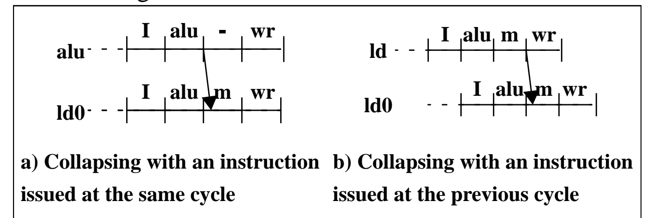

Figure 2: **Address-calculating collapsing.**

From distance distribution tables (Table 2), the reduction of execution cycles achieved by this technique in an in-order processor can be predicted. If the source instruction is an arithmetic one, no AGI interlock will appear. If the source instruction is a load and both

instructions are at the same group, only one AGI cycle will be generated, otherwise, no AGI cycles will arise.

For instance, using groups of two instructions in *go* benchmark (Table 2.a), the expected reduction of stall cycles is 3.1 millions (number of zero-offset loads close enough to its source instruction). On the other hand, Table 3 shows that the amount of AGI-0 stall cycles produced in the two-way processor is 3.39 millions of cycles (3.39 percent of 100M cycles). Cycles difference is explained due to AGI related to store instructions (not included in Table 2.a).

4.2. ZERO-OFFSET LOAD/STORE ADVANCING (*ZA*)

We propose to modify the usage of pipeline stages to reduce the latency of zero-offset loads. The generation of the effective address of a zero-offset load is an useless calculation because the base and the effective address are equal. To take advantage of this fact, we propose to access memory after issue stage. Consequently, the method advances one stage the memory access for a zero-offset load as regards the baseline pipeline usage (Figure 3). The result of advanced loads must be delayed one cycle in the pipeline to guarantee that every instruction longs the same number of cycles.

This technique reduces one cycle the latency of the advanced zero-offset loads, so, the number of LUI is decreased. Every advanced zero-offset load will reduce one stall cycle, if any.

The processor must check: a) conflicts in the memory system, and b) ordering between loads and stores. A conflict can be originated because there can be more memory access per cycle than supported by the memory system. Disordering loads is not a problem, but if an a zero-offset load is advanced and then overtakes a store instruction, a hazard can arise. In these cases, the processor executes the zero-offset load as an usual load, without stalling.

It is useful to apply this technique on zero-offset stores because it can solve some memory-access conflicts.

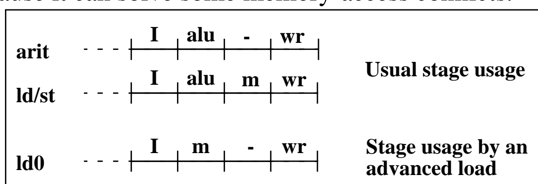

Figure 3: **Pipeline stage usages.**

From distance-distribution tables (Table 2) can be predicted the execution-cycle reduction achieved by this technique in an in-order processor. If the zero-offset load and the destination instruction are at the same group, one LUI cycle will be produced. Otherwise, no LUI cycles will be generated.

For instance, using groups of four instructions (Table 2.d) in *anagram* benchmark, the expected reduction of stall cycles is 2.6 millions (it is the number of zero-offset loads close enough to its first destination instruction). On the other hand, Table 3 details the amount of LUI-0 stall cycles produced in the four-way processor, it is 3.2 millions of cycles (16.81 percent of 19M cycles). Cycle difference is explained due to LUD's between load instructions placed at the same group.

5. PERFORMANCE EVALUATION

5.1. SIMULATION MODELS

As baseline machine we will use an in-order superscalar processor. Its pipeline has the following stages: Instruction Fetch (IF), Decode (D), Issue (I), Alu, Memory Access (M) and Write Registers (WR). It is similar to the integer-instructions pipeline of the Alpha-21164 processor [8].

Issue Width	2-way	4-way
Fetch Interface	half cache block/cycle (4 contiguous and aligned instructions)	
Instruction buffers	2 half-cache-block buffers (8 instructions)	4 half-cache-block buffers(16 instructions)
I-Cache	Perfect	
Branch Predictor	256 2-bit counters, Direct mapped, no tag checking	
	1 prediction/cycle	2 predictions/cycle
Issue Mechanism	In order issue. Any pair of aligned and independent instructions are issued in parallel except an store and any other memory instruction, or two branches	In order issue. Any quartet of aligned and independent instructions are issued in parallel except those with three or four memory instructions or three or four branches.
Functional Units	2 ALU	4 ALU
Functional Units Latency	Mem : 2 cycles ALU : 1 cycle	
D-Cache	8Kb direct mapped, 32-byte cache blocks, 6-cycle miss latency Write through and no allocate, Blocking cache, Nonblocking loads	
	2 read ports, 1 write port (stores have exclusive access on D-Cache)	2 read/write ports
Write Buffer	8 merging cache blocks	

Table 4: Processor models

IF stage gets four aligned instructions per cycle. These instructions will be stored on instruction buffers. Instruction Buffers are placed at Decode stage, they try to reduce the stall cycles produced by the fetch delay. If Instruction Buffers are full, fetched instructions will be discarded.

Decode stage predicts branch instructions; this prediction can produce the discarding of the remaining fetched instructions at the same cycle. If the next stage (I) is empty, the oldest aligned instructions are sent to.

At I stage, static and dynamic conflict checks are performed. Static conflicts are produced by a lack of functional units, for example, the four-way processor can not issue at the same time three memory instructions because the memory system offers two ports. Dynamic conflicts are produced by data dependencies. In both cases,

the instruction that produces the conflict, and logically subsequent instructions are stalled until the conflict disappears.

ALU stage computes arithmetic operations, calculates effective addresses, and evaluates branch conditions. In a branch misprediction, subsequent instructions to the branch instruction will be discarded, and, at next cycle, correct branch destination will be fetched.

The instruction set offers only one addressing mode for accessing memory: base address plus offset, where base address is stored in a register, and the offset is coded at the instruction.

To evaluate the proposed techniques two in-order issue widths configurations will be simulated. Table 4 details their instruction latencies, and memory hierarchy configuration.

ZA technique can break the issue rules imposed by the memory system of the two-way processor, for instance, it is possible to issue at the same cycle a load and a store instruction if one of them is advanced.

5.2. SIMULATION ENVIRONMENT

Binaries used in this work have been obtained compiling with O4 switch of the original machine compiler (an Alpha 21164 processor, with OSF1 V3.2 installed). Then, they have been instrumented with ATOM (this tool is able to instrument user level code, but does not instrument operating-system code). To analyse the proposals of this work, we have developed a cycle oriented simulator and the whole benchmarks have been simulated.

5.3. SIMULATION RESULTS

In an in-order processor it is important to exploit the processor parallelism. Every stall cycle prevents the issue of some instructions (from one to the issue width). The goal of *ACC* is maintaining the parallelism by tolerating the latency of the source instruction of a zero-offset load/store. On the other hand, the objective of *ZA* is reducing the latency of a zero-offset load whenever the parallelism is not reduced (for instance issue-rules conflicts, and store/load disordering).

The improvement produced by the parallelism maintenance (*ACC*), and the gain originated by the latency reduction (*ZA*) are orthogonal. So, the reduction of execution cycles obtained applying at the same time both techniques is the addition of the reduction achieved applying every method individually.

Table 5 shows the influence of the proposed techniques on selected benchmarks. We present results in terms of instructions issued per cycle (IPC) of the baseline processor and percentage of IPC improvement for both the two-way and the four-way processor. Also details the branch-predictor behaviour and the hit rate of the data cache.

Proposed techniques exhibit better improvements in the four-way processor than in the two-way processor, because the issuing of more instructions per cycle produces more chances to apply them. Moreover, *ZA* technique produces more benefits than *ACC* (as can be expected from Table 2).

Benchmark	2 way			4 way			predict correct.	hit rate
	Base IPC	% ACC	%ZA +ACC	Base IPC	% ACC	%ZA +ACC		
go	0.793	3.35	9.22	0.838	7.00	11.06	73.87	86.39
mksim	1.033	0.70	2.68	1.129	1.20	3.02	87.88	92.80
gcc	0.921	1.66	4.05	0.997	2.32	5.20	80.50	88.60
compress	1.134	2.72	8.29	1.294	4.87	13.45	83.86	92.99
li	0.925	1.27	3.70	0.986	1.19	3.76	86.48	87.99
ijpeg	1.288	0.18	3.43	1.430	0.80	5.73	89.98	94.98
perl	0.931	1.87	5.41	1.009	2.80	7.54	88.49	89.51
vortex	0.976	1.38	5.40	1.078	2.09	7.93	90.83	88.28
anagram	1.029	4.14	8.76	1.089	4.69	11.93	93.77	90.63
bc	1.079	1.18	4.35	1.190	1.54	5.13	79.50	91.91
ks	0.634	0.28	11.84	0.652	2.58	13.49	87.61	88.03

Table 5: Simulation results: baseline-model IPC, percentage of IPC improvement applying ACC respect baseline model, and percentage of IPC improvement applying ACC and ZA respect baseline model (two-way and four-way processors used as baseline model), branch prediction and data-cache hit rates.

Codes using vectors (*go*, *compress*, *anagram*) show better improvements than codes using pointers (*li*, *gcc*, *perl*, *bc*). This fact is understandable looking at the code generated by the compiler (Table 1). The first group of codes presents a high percentage of zero-offset loads/stores (almost 20%), but the second group displays a lower one (11%).

ks uses pointers but presents a lot of zero-offset loads because the most accessed field is placed with a zero displacement respect the record base address. *Address-Calculating Collapsing* presents a poor improvement because the AGD distance is large, but *Zero-load advancing* exhibits a good gain because the LUD distance is short.

m88ksim and *ijpeg* use vectors but display a poor improvement because the compiler is able to unroll the loops that access vectors, so it generates nonzero-offset loads. Moreover, in *ijpeg*, there are vector accesses to fixed positions, so the compiler does not need to generate an arithmetic instruction to compute the reference address, it can use the offset field of the load instruction.

6. RELATED WORK

Several studies report the importance of untolerated load latencies on processor performance.

A load-latency tolerating technique is analysed in [2] comparing two pipeline organizations. The first one (LUIP) is the traditional RISC pipeline (Section 5.1 describes a LUIP pipeline). The second pipeline (AGIP) pushes the execution of ALU operations to the same stage as cache access, increasing the mispredicted-branch penalty by one cycle. AGIP reduces the amount of LUI interlocks, but increases the number of AGI interlocks. Which organization offers better performance depends on the branch misprediction rate, the amount of AGI and LUI

interlocks and the compiler support. Comparing [2] to our approach, we can notice two differences: i) related work does not reduce load latency and ii) it increments branch-misprediction latency.

A technique for reducing load latency is presented in [3]. It predicts the cache index of a memory reference (or'ing the base address and the offset) and access data cache speculatively during the effective-address calculation stage. If the prediction is correct, the cache access has been overlapped with nonspeculative effective-address calculation (and the load latency has been reduced), otherwise, the cache is accessed again using the correct effective address. To improve the prediction accuracy of the mechanism, some compiler and linker support is required.

An aggressive hardware mechanism is added to the previous technique in [4]. It tries to produce the result of load instructions prior to reaching the execute stage of the pipeline, reducing the effective load latency to zero cycles. To check the prediction correctness, this technique looks for address generation dependencies.

Another load-latency tolerating technique is proposed in [5]. A load unit detects load instructions in the instruction buffer, predicts its effective address and access data cache before load instruction reaches address calculation stage. If the predicted address is correct, effective load latency is reduced to one cycle. This load unit implements two predictors: the first uses the current values of the registers (note that AGD's can produce wrong predictions), and the second uses a stride table and a two-delta algorithm. AGD detection and table tagging are used to filter predictions and the load unit uses history information to select one prediction.

Previous works ([3][4][5]) predict load effective addresses and access data cache speculatively, but never execute subsequent dependent instructions before checking the correctness of the prediction. A different approach is proposed in [6], this technique issues the dependent instructions of a load without waiting for the checking of the prediction, if the prediction is incorrect a recovery mechanism must restore the processor state.

Comparing [3][4][5][6] to our approach, four main differences can be found: i) *ACC* and *ZA* never access memory speculatively, ii) we propose a technique (*ACC*) to tolerate the latency of the source instruction of a zero-offset memory reference, iii) our proposes can only by applied to zero-offset memory references (but they represent a significant amount of memory instructions) and iv) the complexity of our approach is simpler than the complexity of related works. Moreover, we propose two state-less techniques but the stride table predictor proposed in [5] and [6] can only be applied after a learning phase.

7. CONCLUSIONS

Program analyses show that zero-offset memory instructions are executed frequently by integer-intensive codes. Data structures used by the programs influence in the number of executed zero-offset memory instructions. Codes using vectors or linked records where the pointer field is placed with a zero displacement respect the record address present more zero-offset load/stores than other programs. Moreover, these instructions are involved on a significant number of stall cycles.

This work proposes two techniques for reducing the amount of stall cycles where zero-offset memory instructions are concerned: *Address-Calculating Collapsing* and *Zero-offset load advancing*.

The first one helps to exploit more parallelism because it allows the parallel issue of two dependent instructions; it tolerates the latency of the source instruction of a zero-offset load/store. The second one reduces the latency of zero-offset memory access instructions, so it decreases the amount of interlocks.

The distance from the source instruction to the memory-reference instruction, and from it to the destination instruction, and the amount of zero-offset load/store determine the improvement achieved by the techniques. In the analysed programs, codes with a high percentage of executed zero-offset loads/stores obtain better improvements than the remaining codes. Issuing more instructions per cycle benefits the proposed techniques because it increments the number of chances to apply them.

ACKNOWLEDGEMENTS

This work was supported by the Ministry of Education and Science of Spain (CICYT TIC-0429/95)

REFERENCES

[1] J. L. Hennessy and D. Patterson, *Computer Architect a Quantitative Approach*. (Morgan Kaufnab Publishers, Inc., 1995)

[2] M. Golden and T. Mudge, A Comparison of Two Pipeline Organizations, *Proceedings of the 27th International Symposium on Microarchitecture*, 1994, 153-161.

[3] T.M. Austin, D.N. Pnevmatikos and G.S. Sohi, Streamlining Data Cache Access with Fast Address Calculation, *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, 369-380.

[4] T.M. Austin and G.S. Sohi, Zero-Cycle Load: Microarchitecture Support for Reducing Load Latency, *Proceedings of the 28th International Symposium on Microarchitecture*, 1995, 82-92

[5] R.J. Eickemeyer and S. Vassiliadis, A load-instructions unit for pipelined processors, *IBM Journal of Research and Development,* 37 (4), 1993, 547-564

[6] Y. Sazeides and J.E. Smith, The Predictability of Data Values, *Proceedings of the 29th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, 1996, 238-247

[7] T.M. Austin, S.E. Breach and G.S. Sohi, Efficient Detection of All Pointer and Array Access Errors, *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, 1994.

[8] D. P. Bhandarkar, *Alpha, implementations and architecture*. (Digital Press, 1996).

[9] K. C. Yeager, The MIPS R10000 SuperScalar Microprocessor. *Microprocessor Report*, 10 (14), 1996, 28-40