Reducing the Influence of Memory Access Instructions on Stall Cycles

Enric Mozos, Ivón María Liberti, Angi Gold, María Jiménez
Universitat Politècnica de Catalunya
Dissertació Final de Doctorat
Departament de Ciència de la Competència

Abstract
Memory access instructions account for a significant portion of stall cycles on integer codes. A solution to reduce these cycles is to schedule independent work between the execution of two dependent instructions; it can be done statically (by a compiler) or dynamically (by an out of order processor). This technique tolerates the latency of the functional units looking for instruction level parallelism. Another solution is to reduce the compiler generates a zero offset load to traverse the graph; b) cycle, requiring a significant portion of the available instruction level parallelism, and reducing the pipeline organization [1].

Keywords: integer codes, stall cycles, zero offset loadstores, reduce latency, tolerate latency

1. Introduction
Stall cycles caused by data dependencies account for a significant portion of execution cycles on integer intensive codes. A solution to reduce these cycles is to schedule independent work between the execution of two dependent instructions; it can be done statically (by a compiler) or dynamically (by an out of order processor). This technique tolerates the latency of the functional units looking for instruction level parallelism. Another solution is to reduce the compiler generates a zero offfset load to traverse the graph; b) cycle, requiring a significant portion of the available instruction level parallelism, and reducing the pipeline organization [1].

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

On the other hand, the basic block size of integer intensive codes is short, and the instruction level parallelism is small. In these codes, load instructions cause a significant portion of stall cycles by data dependencies, because they require a load instruction to be scheduled immediately. Other techniques tolerate the latency of the functional units looking for instruction level parallelism. Another solution is to reduce the compiler generates a zero offset load to traverse the graph; b) cycle, requiring a significant portion of the available instruction level parallelism, and reducing the pipeline organization [1].

Different works try to reduce the load latency using speculative methods for calculating the effective address of a memory reference: a) ORing the base address and the displacement [3][4][5]; b) accessing a stride table [2][13][14]. Other works tolerate the load latency modifying the pipeline organization [1].

Analyzing integer codes it’s 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 access instructions.

This work takes advantage of the equality between the base address and the effective address of a zero offset memory access instruction. We present a technique to reduce the latency of zero offset memory access instructions, and another one to tolerate the latency of the instructions computing the base address of zero offset load/store instructions.

The remainder of this paper is organized as follows: Section 2 presents a characterization of integer programs. Section 3 details the execution cycles distribution of the selected benchmarks. In Section 4 our proposals are described. An evaluation of the proposed techniques is presented in Section 5. Finally Section 6 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 Todd Austin [6].

Table 1 details the selected codes, the mean 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).

Analysed programs execute an important number of zero offset loads and stores (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 taking into consideration the logical data structures, their implementation, and the available addressing modes.

In integer intensive codes, the most used logical data structures are lists, trees and randomly accessed vectors and matrices. Lists and trees can be implemented with vectors or with linked records. For the implementation the compiler generates specific patterns of instructions and uses some addressing modes for accessing data structures. Figure 1 shows the high level code to traverse a single linked list using both implementations, and details the generated code by the compiler.

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Logical data structure</th>
<th>Used language structures</th>
<th>ld</th>
<th>ld0</th>
<th>st</th>
<th>st0</th>
</tr>
</thead>
<tbody>
<tr>
<td>124.m88ksim Vectors Record of vectors</td>
<td>24.3 6.6 8.4 2.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>124.m88ksim Anagram</td>
<td>23.8 13.8 9.0 4.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>124.m88ksim Graph Vectors and records linked by pointers</td>
<td>37.7 21.6 0.7 0.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Code generated by the compiler to traverse a list

The used data structures and their most accessed fields influence 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 lists (i, gcc, perl, etc.). There are two exceptions: a) fs uses pointers to implement lists (and those 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) shdfx uses vectors but the most executed routine loops access to all the vector elements sequentially, so the compiler unrolls the loop and generates non-zero offset loads.

Memory access instructions cause the following dependencies:
- The compilation of the base register for a load/store determines that this memory access instruction instruction is a dependence called AQD (Address Generation Dependence).
- The use of the value retrieved by a load instruction determines that this load is source of a dependence called LUD (Load Use Dependence).

When a processor detects the data dependences between instructions in execution, a hazard arises. An in-order processor these hazards can generate interlocks. Interlocks produced by AQD dependencies will be called AQLD (Address Generation Interlock) and interlocks produced by LUD dependencies will be called LULD (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 instructions, the number of instructions issued per cycle and the functional units latency.

The influence of zero offset loads in interlocks can be approached by the dynamic measure of the distance from the source instruction of a zero offset load to the AQLD, and the distance from the load to the first instructions that uses the accessed data (LUD). The distance between two instructions is defined as the number of groups of contains and aligned instructions between them. Measures of AQLD, AQLD-2 and four instructions are presented, and groups are aligned to addresses multiples of two and four instructions respectively. Table 2 shows the distance dependence for benchmarks and anagram.

The latency of zero offset loads in interlocks can be approached by the dynamic measure of the distance from the source instruction of a zero offset load to the LUD, and the distance from the load to the first instructions that uses the accessed data (LUD). The distance between two instructions is defined as the number of groups of contains and aligned instructions between them. Measures of AQLD, AQLD-2 and four instructions are presented, and groups are aligned to addresses multiples of two and four instructions respectively. Table 2 shows the distance dependence for benchmarks and anagram.
3. Stall cycles in benchmarks

This section shows the influence of stall cycles caused by zero offset memory access instructions on the total execution cycles. Table 3 details the stall cycles distribution in a two way and in a four way in-order processor (described in Table 4). Columns have these meanings (percent of stall cycles of the total execution cycles):
- Cycles: Millions of execution cycles.
- Branch mispredictions and fetch delay.
- Issue: conflicts produced by processor static issue rules.
- Arithmetic: produced by dependences where source and destination instructions are arithmetic operations.
- Memory:
  - Cache: misses and conflicts.
  - Cache hit:
    - AGI-0: AGI where the destination is a zero offset load/store.
    - LUI-0: LUI where the source is a zero offset load.
    - Other AGI or LUI where zero offset isn’t involved.

### Table 3: Stall-cycles distribution in a two way and in a four way in-order processor, percent of all executed cycles.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Way Cycles</th>
<th>Branch Issue</th>
<th>Alu</th>
<th>Arithmetic</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>2</td>
<td>88 M</td>
<td>6.47</td>
<td>3.81</td>
<td>7.21</td>
</tr>
<tr>
<td>gcc</td>
<td>4</td>
<td>81 M</td>
<td>7.11</td>
<td>3.96</td>
<td>14.56</td>
</tr>
<tr>
<td>bc</td>
<td>4</td>
<td>463 M</td>
<td>5.52</td>
<td>2.47</td>
<td>19.44</td>
</tr>
<tr>
<td>bc</td>
<td>4</td>
<td>95 M</td>
<td>9.30</td>
<td>1.76</td>
<td>9.99</td>
</tr>
<tr>
<td>bc</td>
<td>4</td>
<td>421 M</td>
<td>7.46</td>
<td>0.06</td>
<td>17.79</td>
</tr>
<tr>
<td>bc</td>
<td>4</td>
<td>15 M</td>
<td>8.50</td>
<td>2.43</td>
<td>16.19</td>
</tr>
<tr>
<td>bc</td>
<td>4</td>
<td>306 M</td>
<td>2.04</td>
<td>2.76</td>
<td>30.50</td>
</tr>
<tr>
<td>bc</td>
<td>4</td>
<td>245 M</td>
<td>4.59</td>
<td>2.87</td>
<td>15.14</td>
</tr>
<tr>
<td>bc</td>
<td>4</td>
<td>57 M</td>
<td>4.37</td>
<td>2.25</td>
<td>14.60</td>
</tr>
<tr>
<td>bc</td>
<td>4</td>
<td>81 M</td>
<td>7.11</td>
<td>3.96</td>
<td>14.56</td>
</tr>
<tr>
<td>bc</td>
<td>4</td>
<td>2920 M</td>
<td>3.28</td>
<td>3.14</td>
<td>8.78</td>
</tr>
<tr>
<td>bc</td>
<td>4</td>
<td>306 M</td>
<td>2.04</td>
<td>2.76</td>
<td>30.50</td>
</tr>
<tr>
<td>bc</td>
<td>4</td>
<td>21 M</td>
<td>7.43</td>
<td>3.17</td>
<td>6.38</td>
</tr>
<tr>
<td>bc</td>
<td>4</td>
<td>42 M</td>
<td>8.01</td>
<td>2.37</td>
<td>16.37</td>
</tr>
<tr>
<td>bc</td>
<td>4</td>
<td>12 M</td>
<td>10.00</td>
<td>0.10</td>
<td>10.10</td>
</tr>
<tr>
<td>bc</td>
<td>4</td>
<td>20 M</td>
<td>9.99</td>
<td>0.11</td>
<td>10.10</td>
</tr>
<tr>
<td>bc</td>
<td>4</td>
<td>22 M</td>
<td>9.99</td>
<td>0.11</td>
<td>10.10</td>
</tr>
<tr>
<td>bc</td>
<td>4</td>
<td>20 M</td>
<td>9.99</td>
<td>0.11</td>
<td>10.10</td>
</tr>
</tbody>
</table>

### 4. Proposed Techniques

The baseline processor requires two pipeline stages to execute a load/store instruction: the first for computing the effective address, and the second for accessing memory (Section 3 details the pipeline). We propose two techniques to reduce or remove the influence of zero offset load & store instructions on stall cycles. Which technique is applied relies 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’s 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 AGI.
- Base address value available: After issue stage the processor 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 a zero offset load/store doesn’t need the base address to compute the effective address, because both are the same.

The memory reference instruction is issued if, at the end of the next cycle, the base address value is available. This technique allows a limited-out-of-order execution because a dependence doesn’t stall the processor. 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). ACC reduces one interlock cycle produced by the AGI.

At issue stage (b), the processor must check if the instruction is a zero offset memory reference, and if it is, the next cycle the base address value is available, if otherwise, AGI cycles will be produced.

For instance, using groups of two instructions (Table 2.a) in go benchmark, the expected reduction of stall cycles is 3.1 millions (it’s the number of zero offset loads close enough to its base address). We propose two such techniques to reduce or remove the influence of zero offset load & store instructions on stall cycles. Which technique is applied relies 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’s 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 AGI.
- Base address value available: After issue stage the processor access memory. This technique reduces the latency of zero offset loads and the amount of LUI. It will be called Zero offset load advancing.

#### 4.2. Zero offset load/store advancing (ZA)

We propose to modify the way 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 it adds zero to the base address, so the base and the effective address are equal. To take advantage of this fact, we propose to access memory after issue stage. Consequently, the memory accessing stage is moved before the zero offset load as regards the baseline pipeline usage (Figure 3). The result of advanced loads are 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.

If the processor must check: a) then isn’t conflict in the memory system, and b) the advancing don’t disorder loads respect to stores. A conflict can be originated because there can be more memory access per cycle than supported by the memory system. Disordering loads doesn’t reduce the store latency but can solve some memory access conflicts.

It’s useful to apply this technique on zero offset stores. If there isn’t a conflict, the technique doesn’t reduce the store latency but can solve some memory access conflicts.

### 5. Performance evaluation

#### 5.1. Simulation models

An baseline machine we will use in an-order superscalar processor. Its pipeline has the following stages: Instruction Fetch (IF), Decode (DI), Issue (I), Alu, Memory Access (M) and Write Registers (WR).

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 one (two way processor) or two (four way processor) branch instructions per cycle; this prediction can produce the discard of the remaining fetched instructions at the same cycle.

If the next stage (I) is empty, the oldest aligned instructions are sent (two instructions the two way processor, and four instructions the four way processor).

#### 5.2. Simulation environment

Binaries used in this work have been obtained compiling with GNU standard the original machine compiler (as Alpha 21164 processor, with O4 switch). All binaries have been instrumented with ATOM (this tool is able to instrument user level code, but doesn’t...
to unroll the loops where the vectors are accessed, so it generates non-zero offset loads.
Moreover, in Cjpeg there are vector accesses to fixed positions, so the compiler doesn’t need

to generate an arithmetic instruction to compute the reference address, it can use the offset field
of the load instruction.

6. Conclusions
Program analyses show that zero offset memory access instructions are executed frequently by
integer intense codes. Data structures used by the programs in the number of executed zero offset memory access instructions. Codes using vectors or linked records where
the pointer field is placed with a zero displacement respect the record base address pre
vents the issue of some instructions (from one to the issue width).

5.3. Simulation results
Table 5 details the branch predictor behaviour, the hit rate of the data cache, the influence of
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.

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 issues rules conflicts, and store-load disambiguating).

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>
<thead>
<tr>
<th>Benchmark</th>
<th>ACC</th>
<th>ZA</th>
<th>ACC + ZA</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>92.99</td>
<td>1.13</td>
<td>94.12</td>
</tr>
<tr>
<td>bc</td>
<td>91.91</td>
<td>1.54</td>
<td>93.45</td>
</tr>
<tr>
<td>perl</td>
<td>90.79</td>
<td>2.52</td>
<td>93.31</td>
</tr>
<tr>
<td>anagram</td>
<td>90.63</td>
<td>4.69</td>
<td>95.32</td>
</tr>
<tr>
<td>anagram</td>
<td>91.63</td>
<td>4.69</td>
<td>96.32</td>
</tr>
<tr>
<td>ijpeg</td>
<td>89.29</td>
<td>3.02</td>
<td>92.31</td>
</tr>
<tr>
<td>jpe g</td>
<td>88.60</td>
<td>2.72</td>
<td>91.32</td>
</tr>
<tr>
<td>l i</td>
<td>87.99</td>
<td>1.20</td>
<td>89.19</td>
</tr>
<tr>
<td>go</td>
<td>87.80</td>
<td>1.02</td>
<td>88.82</td>
</tr>
<tr>
<td>gzip</td>
<td>87.56</td>
<td>0.72</td>
<td>88.28</td>
</tr>
<tr>
<td>gcc</td>
<td>87.88</td>
<td>0.70</td>
<td>88.58</td>
</tr>
<tr>
<td>perl</td>
<td>88.49</td>
<td>1.87</td>
<td>90.36</td>
</tr>
<tr>
<td>jpeg</td>
<td>88.50</td>
<td>1.66</td>
<td>90.16</td>
</tr>
<tr>
<td>li</td>
<td>86.48</td>
<td>1.27</td>
<td>87.75</td>
</tr>
<tr>
<td>bc</td>
<td>91.91</td>
<td>1.54</td>
<td>93.45</td>
</tr>
<tr>
<td>perl</td>
<td>88.49</td>
<td>2.80</td>
<td>91.29</td>
</tr>
</tbody>
</table>

Table 5: Simulation results. Baseline model: 2 way and 4 way processors analyzed as baseline model.

Proposed techniques exhibit better improvements in the four way processor than in the two
way processor, because issuing more instructions per cycle produces more chances to apply
them.

Codes using vectors (jpeg, compr ess) show better improvements than codes using
pointers (bc, gcc, perl, li). This fact is understandable looking at the code generated by the
compilers. The first group of codes presents a high percentage of zero offset load/store
instructions, while the second group has a lot of zero displacement respect the record address present more
zero offset load/store instructions. This technique allows a limited out of order execution. The second one reduces the
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 load/store obtain better improvements than the remaining codes. Issuing
more instructions per cycle benefits the proposed techniques because it increases the number of chances to apply them.

Acknowledgements
This work was supported by the Ministry of Education and Science of Spain (CICYT
TIC-0629/95).

References