Investigating the Predictability of Linked Data Structures

BART GOEMAN, KOEN DE BOSSCHERE
Dept. of Electronics and Information Systems
Ghent University
Sint-Pietersnieuwstraat 41, B-9000 Ghent
BELGIUM
{bgoeman,kdb}@elis.rug.ac.be
http://www.elis.rug.ac.be/~{bgoeman,kdb}

MATEO VALERO
Dept. de Arquitectura de Computadores
Universidad Politècnica de Catalunya
c/ Jordi Girona 1-3, 08034 Barcelona
SPAIN
mateo@ac.upc.es
http://www.ac.upc.es/homes/mateo

Abstract: Linked data structures are not well suited for automatic parallelisation, and hence even a modern out-of-order architecture has problems to overcome the data dependencies that are being created by the links, especially during pointer chasing. In this paper, we show how value prediction can be used to effectively break up the data dependencies, and hence to unleash additional parallelism in the application. We show that the pointers that are used to chain the nodes of a linked data structure are amongst the best predictable values even with an FCM predictor with moderate size. This opens possibilities to speed up pointer based programs: their ILP is increased, and the impact on the performance of cache misses during pointer chasing is reduced.

Key Words: Value prediction, Speculation, FCM, Linked data structures

1 Introduction

Current microprocessor architectures use increasingly aggressive techniques to raise the average number of instructions executed per cycle (IPC). The upper bound on achievable IPC is generally imposed by true dependencies, either via registers, or via memory: instructions that need input from other instructions have to wait until the latter are finished.

Value prediction is a technique capable of pushing this upper bound by not waiting for the output of an instruction anymore, but by predicting it and by speculatively executing the dependent instructions using the predicted value. The instruction whose output was predicted must eventually be executed to verify the prediction, and to commit and retire the instructions that were executed speculatively. This is similar to branch prediction.

The concept of value prediction was introduced by Lipasti et al. [8] and by Gabbay et al. [5] They used respectively a last-value predictor and a stride predictor.

Several studies have shown that register values are indeed predictable [8, 11, 12, 13, 6] (in previous work we have reported accuracies up to 79% [6]), and that a significant speedup is achievable [3, 7, 10]. We however believe that the predictability strongly depends on the kind of code that is executing. In order to be predictable, there must be a kind of superstruc-
ture that creates a pattern that can be predicted (e.g., loops, array indexing, or counters in general). Only when we understand the behavior of a particular programming construct, we will be able to effectively predict it.

The only work in this field is by Lipasti et al. [8], which gives some examples of predictable programming constructs, and Calder [4] who uses the predictability to guide code specialization.

In this paper, we focus on one particular easy to recognize construct, namely the code needed to manipulate a Linked Data Structure (LDS). Linked Data Structures are very common in many applications, because of their flexibility. Unfortunately, the are very hard to parallelize (low ILP) because traversing a list (pointer chasing) is inherently serial. Furthermore, since dynamically allocated data structures have no built-in spatial locality (as arrays have), they benefit less from the caches.

Anything that can be done to improve the ILP or cache behavior is welcome. It turns out that if we could successfully predict the outcome of loads in a LDS, we would increase the ILP, and cache misses would have less impact on the performance.

This paper starts with clearly defining which instructions we want to predict. We then describe the benchmarks that are used, and present the results for a series of value predictors.

2 Methodology

In Figure 2, we show the most frequently executed loop of the health benchmark. The linked data structure that is used by this code depicted in Figure 2. The nodes are horizontally linked with the forward field, while the patient field keeps track of the data in the list. We have marked three instructions that need a load-instruction in their implementation.

```c
while (list != NULL) {
    i = village->hosp.free_personnel;
    p = list->patient;
    if (i > 0) {
        village->hosp.free_personnel--; 
        p->time_left = 3;
        p->time += p->time_left;
        removeList(&(village->hosp.waiting), p);
        addList(&(village->hosp.assess), p);
    } else {
        p->time++;
    }
    list = list->forward;
}
```

Fig. 1: The most frequently executed loop of the health benchmark.

![Fig. 2: The LDS of the health LDS.](image-url)
Table 1: The benchmarks used

<table>
<thead>
<tr>
<th>Name</th>
<th>Params</th>
<th>Instr</th>
<th>Description</th>
<th>Data Structures [1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>biort</td>
<td>35000 0</td>
<td>96M</td>
<td>sorting (merge)</td>
<td>dyn. binary tree</td>
</tr>
<tr>
<td>em3d</td>
<td>2000</td>
<td>21M</td>
<td>scientific code</td>
<td>static list</td>
</tr>
<tr>
<td>health</td>
<td>4 500 1</td>
<td>40M</td>
<td>process simulation</td>
<td>dyn. lists</td>
</tr>
<tr>
<td>mist</td>
<td>512 1</td>
<td>50M</td>
<td>graph optimization</td>
<td>static hash tables</td>
</tr>
<tr>
<td>perimeter</td>
<td>8 1</td>
<td>52M</td>
<td>graph utility</td>
<td>static quad tree</td>
</tr>
<tr>
<td>treeadd</td>
<td>15 1</td>
<td>59M</td>
<td>sum of binary tree</td>
<td>static binary tree</td>
</tr>
<tr>
<td>tsp</td>
<td>10000 1</td>
<td>40M</td>
<td>graph optimization</td>
<td>dyn. lists</td>
</tr>
<tr>
<td>li</td>
<td>5 queens</td>
<td>42M</td>
<td>lisp interpreter</td>
<td>dynamic lists</td>
</tr>
</tbody>
</table>

Table 2: Value Predictor Configuration for FCM

<table>
<thead>
<tr>
<th># entries L2</th>
<th>order</th>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^{12}</td>
<td>3</td>
<td>fcml_12</td>
</tr>
<tr>
<td>2^{14}</td>
<td>3</td>
<td>fcml_14</td>
</tr>
<tr>
<td>2^{16}</td>
<td>4</td>
<td>fcml_16</td>
</tr>
<tr>
<td>2^{18}</td>
<td>4</td>
<td>fcml_18</td>
</tr>
<tr>
<td>2^{20}</td>
<td>4</td>
<td>fcml_20</td>
</tr>
</tbody>
</table>

We now present the results for the value prediction for the three classes of load instructions presented in the previous section. The benchmarks were compiled using gcc 2.7.2.3 for the simplescalar PISA architecture [2].

3 Results

We use seven programs from the Olden [9] benchmarks and the SPECint95 li program. We use the Olden benchmarks because these are small and very pointer intensive. For each of these, we identified the most frequently executed LDS code, classified all load instructions in that code and investigated the predictability using several value predictors.

The benchmarks and parameters used are described in table 1. The predictor mechanisms considered are last value, stride and FCM [11]. Each load instruction was investigated separately (no aliasing), and for FCM several orders and sizes of the second level table (containing the values, the first level contains the history) were investigated (see table 2).

3.1 Recurrent Loads

Figure 3 shows the predictability of recurrent loads. The two entries for treeadd correspond to the left and right pointers, when the tree is traversed depth-first from left to right.

In general, a last value predictor performs very bad, except for trees. The bad behavior for lists is not surprising as a node pointing to itself is not very common. The better perfor-
mance for trees is caused by the high probability of NULL pointers in trees ($P_{NULL} \approx \frac{n-1}{n}$ for $n =$ branch factor of the tree). A stride predictor only performs well when the nodes happen to be allocated sequentially in memory. In the order cases, a stride predictor is worse due to its longer learning period. The FCM predictor is better suited for recurrent loads and can achieve a high prediction accuracy. In general, its accuracy is best for linked list, less for trees and the worst for the hash table ($mst$). Well constructed hash tables contain lists consisting of only a few elements, so when a history based predictor has collected enough information about the current lookup to be able to successfully predict, the lookup is already finished. For FCM, the size of the prediction table turns out not to be critical.

### 3.2 Traversal Loads

Figure 4 shows the predictability of traversal loads. Not all the benchmarks contain this type of loads, as the LDS can contain the data itself instead of a pointer. The benchmark $li$ needs to traverse two pointers.

The traversal loads of the $li$ benchmark are quite predictable by a stride predictor. The reason for this is again that the nodes are allocated sequentially in memory, and the structure (the global symbol space of the lisp program) is rather static. However, the predictability by a stride predictor for the other two benchmarks is non-existent. The same applies for last value prediction. FCM predictors are again accurate, but we can observe the size of the prediction table is important here, while for recurrent loads it was not. Thus, in practice a traversal load will be less predictable.

### 3.3 Data Loads

Figure 5 shows the predictability of data loads. Not all benchmarks contain this type of loads
in the inner loop, the inner function of \textit{li} works completely on pointers and \textit{tsp} works on \textit{doubles}, which we cannot predict since the predictor uses 32-bit tables.

Some benchmarks show a very high predictability, but this is partly due to the artificial nature of the benchmarks: the toy benchmark \textit{treadd} has always the same value in the data field and is thus perfectly predictable, while all the hash tables in \textit{mst} contain (almost) the same information. The benchmark \textit{perimeter} benefits from the fact that there are only four possible data values (four wind directions). When we look at the three remaining benchmarks, \textit{bisort}, \textit{en3d} and \textit{health}, the picture is less convincing; only \textit{bisort} is quite predictable. This is an effect of the sorting algorithm (merge sort) and the data being sorted (integers): at the end of the sorting process, the lists being merged contain already long sequences of identical values. This effect would not occur when sorting e.g. strings. The difference between FCM and the other prediction techniques is less pronounced.

4 Conclusions

In this paper, we have investigated the predictability of linked data structures.

Our first observation is that the predictability of load instructions is sometimes due to not-so-obvious program characteristics. Even in the manipulation of linked lists, it is worth making a distinction between recurrent loads, traversal loads and data loads.

In general, we found that data loads are rather unpredictable, that traversal loads are predictable using large FCM tables, and that recurrent loads are often predictable by FCM even using small tables. On the other hand, last value prediction and stride prediction are not suited for pointer predictions in the context of linked data structures.

We believe the higher predictability of recurrent loads is due to the fact that the LDS is only partially transformed during each traversal, thus leaving a lot of links unchanged, and that this information can be used for the next traversal.

Since the recurrent loads can be predicted with moderately sized prediction tables, and since the recurrent loads are generally considered the cause of low ILP, value prediction on recurrent loads can actually improve the performance of pointer based programs: their ILP is increased and the impact on performance of cache misses is reduced.

Acknowledgements

This work was supported by the Fund for Scientific Research – Flanders, project 3G003699.

References


