An unsophisticated cooperative approach to prefetching linked data structures

Alexander Galazin
Murad Neiman-zade
JSC “MCST”, Moscow

Motivation

- Pointer-based applications significantly lack performance due to irregularity of memory access patterns
- There is no information on how linked data structures addresses evolve in major applications
- Existing approaches propose sophisticated cooperative techniques with great modifications in CPU

Background

App	Procedure	$\%T_{app}$	Data Misses
181.mcf	flow_cost	53.7%	94.2%
	update_tree	15.8%	95.1%
197.parser	xfree	7.0%	43.6%
	table_pointer	3.6%	59.4%
254.gap	CollectGarb	9.4%	82.4%
300.twolf	new_dbox_a	17.3%	71.0%

Studying LDS Traversal

- Discover LDS traversal
- Collect $\delta_k = addr_{i+k} - addr_i$, where $addr$ – address with which LDS traversal operates, i-loop iteration and k={1..16}

LDS Traversal Behavior

- 181.mcf
  - *flow_cost*: 2 addresses in LDS and only 1 δ if k is fixed
  - *update_tree*: 3 δ in 97%
- 197.parser
  - *xfree*: 1 δ in 90%
  - *table_pointer*: 3 δ in 49%
- 254.gap
  - *CollectGarb*: 2 δ in 96%
- 300.twolf
  - *new_dbox_a*: 3 δ in 98%

Our method

- Architectural support
  - New instruction **IsOperandsNotReady**
- Compiler support
  - Discover LDS traversal
  - Inject prefetching code
  - Create compensating nodes

Architectural support

- **IsOperandsNotReady(TI)**
  - returns **TRUE** if any of the operands of **TI** are not ready
  - otherwise **FALSE**
  - is always scheduled together with **TI** in the same wide instruction and requires 1 logical unit.

C-code

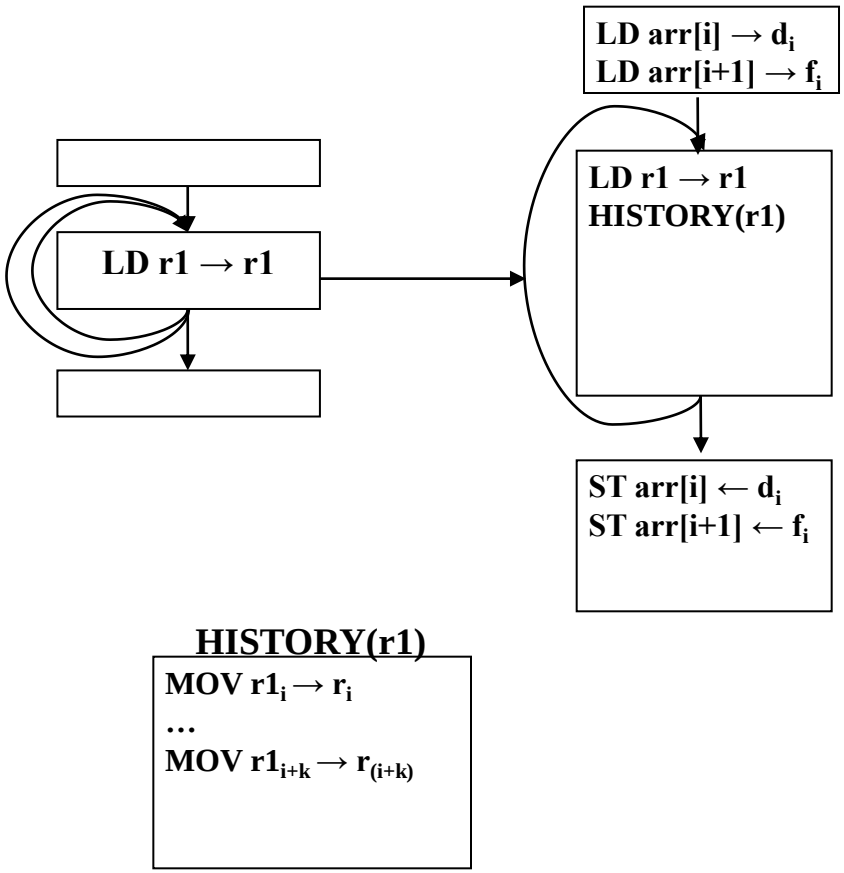
```
while(a)
{
    a=a->next;
}
```

ASM-code

```
{
cmpesb,1 %r0, 0, %pred1
pass %ionr1, %pred5
}
```

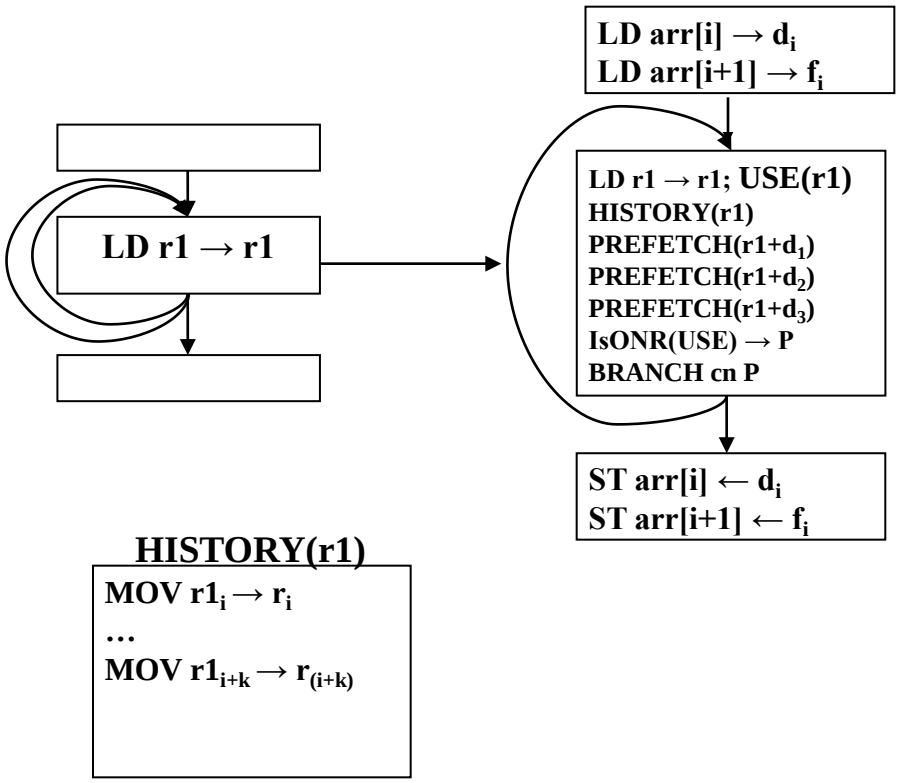
Compiler support. Preparation

- for each LD we create a global array for keeping 3 most popular δ and their frequencies;
- we keep a history of addresses for the load for *D* iterations;
- in the preloop we load all elements of the array to registers
- in the postloop we save values of 3 top δ and their frequencies in the array;



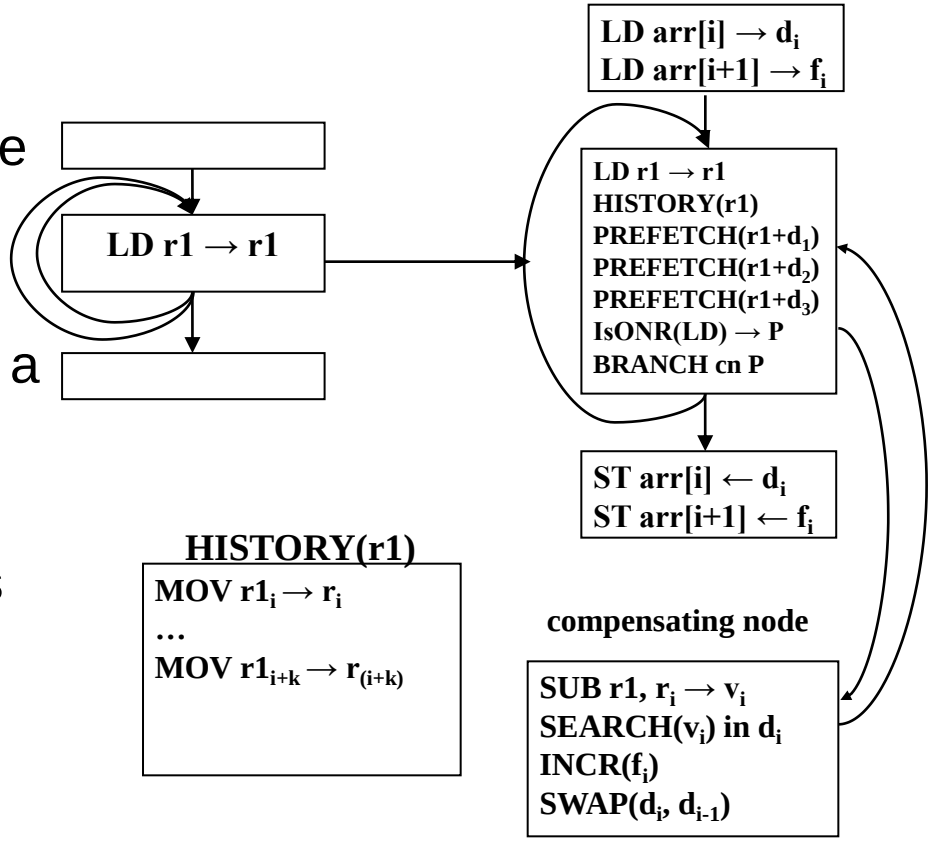
Compiler support. Prefetching

- in the loop head we create prefetches for $(A+\delta)$ where A is the address of the LD on the current iteration;
- after the USE of LD result we add IsOperandsNotReady and branch which transfer control to a compensating node;



Compiler support. Calculating δ

- in the compensating node we calculate *S* – the difference between current load address and its oldest retained address;
- then we search for whether there is such δ and if there is, we increment the value of register which keeps its frequency;
- if there is no such δ we initialize a new register with *S* and set a frequency register to one;
- if the frequency of *S* becomes greater than that of the previous register we swap them, thus doing a “lazy bubble sort”;



Experimental results

- The method was evaluated on a computer with the Elbrus microprocessor;
- The microprocessor has EPIC architecture, 4-way associative L2 of 256 KB, 4 load/store units.
- 181.mcf reduced by 15%
- 254.gap reduced by 4%
- The method is still in the phase of active development