An unsophisticated cooperative approach to prefetching linked data structures.

Alexander Galazin and Murad Neiman-zade
Programming Software Division, JSC "MCST"
{galazin,muradnz}@mcst.ru

The performance of modern microprocessors significantly depends on memory latency. Besides caches which greatly help to hide the latency for programs with spatial and temporal locality, a large number of methods and techniques are proposed to enhance the prefetching of data with regular access patterns. However, there are a lot of applications which lack locality or work with sophisticated data structures based on pointers. Usually pointer-based applications do not exhibit enough regularity or it's impossible to predict the next address until the content of the current is read. In recent years many papers have emerged which introduce different enhancements to compiler and memory models. These novelties aim to analyze statistics gathered during the execution of pointer-based applications and then to issue prefetches when a traversal of linked data structures (LDS) is recognized. These papers mainly discuss a cooperative approach where a compiler marks LDS traversals and a special unit implemented in the architecture executes them speculatively.

Our work is also devoted to the problem of tolerating memory latency in pointer-based applications. We propose a novel technique of predicting addresses used in LDS traversal and we suppose that implementation of our technique is much easier and more efficient compared to existing methods as it doesn't require elaboration of new CPU hardware.

The background of the method we propose lies in our study of SPEC CPU2000. Taking into account applications that suffer greatly from cache misses not overcome by regular data prefetching (particularly 181.mcf and 254.gap) we discovered/ascertained/established the following: if one tries to examine the difference between addresses with which LDS traversal operates on i^{th} and $(i+k)^{th}$ iteration of a loop it will see that there are not more than 3 prevailing values of the difference ($\delta_i, i = 1,2,3$). These values remain during the whole execution of the program and do not depend on the conditions in which the loop is running. In practice this means that we should get these values once during program execution and it's then possible to use them for prefetching and if something unexpected happens with LDS and these values are changed, it is necessary to get them again.

Taking into account these considerations we have developed our cooperative method, which consists of two parts:

1. We add a new instruction which we call **IsOperandsNotReady(TargetInstruction)**. This instruction returns **TRUE** if any of the operands of **TargetInstruction** are not ready and otherwise **FALSE**. It is always scheduled together with **TargetInstruction** in the same wide instruction and requires 1 logical unit.
2. The compiler creates additional code (Fig. 1) in the program which processes result of **IsOperandsNotReady**. The code consists of the following parts:
    a. for each load instruction (**LD**) which is considered to be a part of LDS traversal kernel we create a global array for keeping 3 most popular $\delta_i, i = 1,2,3$ and their frequencies sorted by frequencies;
    b. we keep a history of addresses for the load for *D* iterations, where *D* is calculated as prefetching distance;
    c. in the preloop we load all elements of the array to registers (Δ_i – is the register with the value of δ_i) ;
    d. in the loop head we create prefetches for ($A + \delta_i$) where *A* is the address of the load instruction on the current iteration;
    e. after the load instruction we add **IsOperandsNotReady(LD)** and branch which transfer control to a compensating node;
    f. 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 Δ_i with an equal value and if there is, we increment the value of register which keeps its frequency. If there is no such Δ_i we initialize a new register with *S* and set a frequency register to one, but the number of these newly initialized registers are limited. If the frequency of *S* becomes greater than that of the previous register we swap them, thus doing a “lazy bubble sort”;

g. finally, in the postloop we save values of 3 top Δ_i and their frequencies in the array.

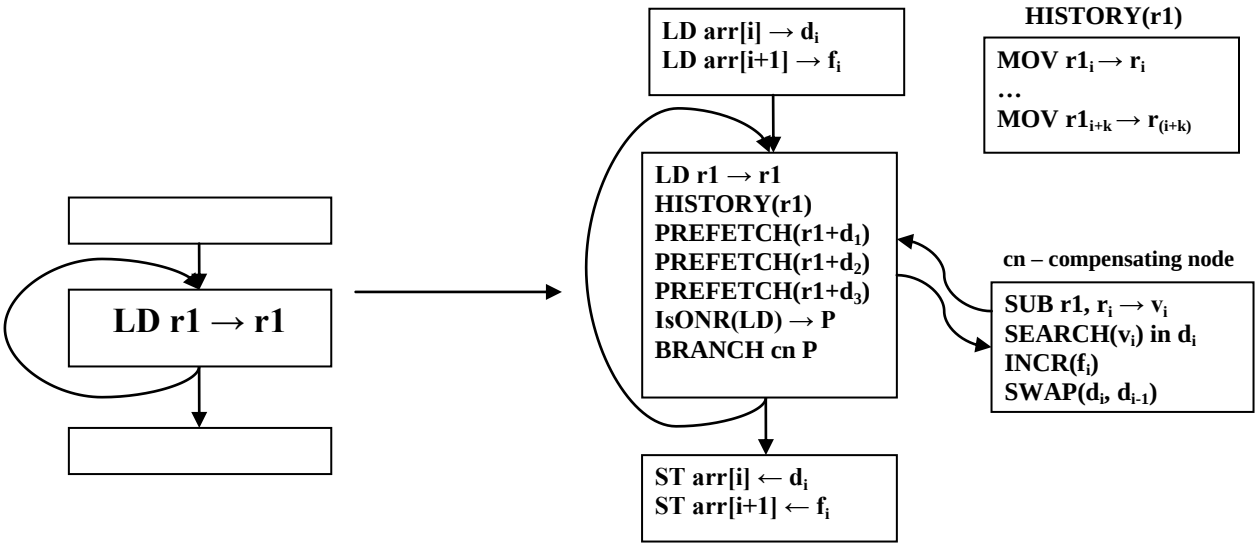

Figure 1. Code transformation

The method described above was evaluated on a computer with the Elbrus microprocessor developed by JSC "MCST". The microprocessor has EPIC architecture, 4-way associative L2 of 256 KB, 64-byte cache line, 4 load/store units. The area of applicability of the method was limited to loops with a high number of iteration to prevent augmentation of overheads over the whole program. Though the method did not apply in all hot spots due to different limitations, it reduced execution time of 181.mcf by 15% and 254.gap by 4%. These results are comparable to that shown by most recent techniques but the method itself is easier to implement due to a very slight modification of the architecture.