Northeastern University

Department of Electrical and Computer Engineering

Project Report

Instruction Prefetching Algorithms Comparison on PIN

Amanda Yilan Zhu

EECE 7352 Computer Architecture

Dr. David Kaeli

December 2016

Introduction

Instruction cache misses could be fatal for high performance processors and superscalar architectures, as this would result in the decrease of amount of instructions that could be fed to the processor within a cycle. Thus even with a high performance superscalar processor, without the ability to eliminate, or reduce significantly the number of cache misses, the processor would be highly inefficient.

In order to reduce cache misses in processors, there are numerous ways to accomplish this. The most commonly seen, and most effective, in this project, is increasing the cache set associativity. However, the downside of this method is taking up too much hardware space and would become less effective as the size of cache increases [1]. Another methodology is by using a victim cache, where we check the victim cache where there is a miss in L1 cache, if the block is there it would be brought up to L1 to swap out the first block that was put in the victim cache [2].

Cache prefetching is also another widely used method to promote cache performance. The two abilities that a good prefetching must possess are the following [1]: it would need to predict when exactly the cache lines would soon be brought into the cache, and also it would have the ability to start the prefetching just in time the instruction is needed. Ideally, this prefetching would have the ability to eliminate all cache misses by prefetching the instruction before they are needed. Three methods of prefetching would be experimented in this project, next-line prefetching, target-line prefetching, and wrong-path prefetching.

Tools

- **Pin**

The basic tool that would be used in this project is Pin 3.0, "a dynamic binary instrumental tool" [3], available for multiple platforms and integrated core architectures. In the downloaded pin, a lot of pintool are provided in C++ with their corresponding header files, which makes it easier for users to derive new tools using the provided ones as guidelines and example.

The specific pin tool that was modified for this project is the icache.cpp, available under pin/sources/tools/Memory.

- **Benchmarks**

Two c++ programs would be used as benchmarks to test the cache misses, FindPi.c, which is the program that computes the value of Pi by using the Monte-Carlo algorithm, and Barnes, an application that implements the Barnes-Hut method used by astronomers to simulate the interaction of a system of bodies, to approximate the forces between the corresponding points.[4] The former one is a simple, recursive and regular benchmark, while the latter is a more "irregular" benchmark that would better test the prefetching algorithms.

.

<u>Methodology</u>

The goal of this project is to successful reproduce the prefetching algorithms on Pin. For the cache simulator, we would have direct-mapping, two-way and four-way set associativity for more data result comparisons. The cache simulator, running on two benchmarks mentioned above are run on the Northeastern University COE Linux x86_64 (nano) architecture.

- ### Modifing Pin Tool

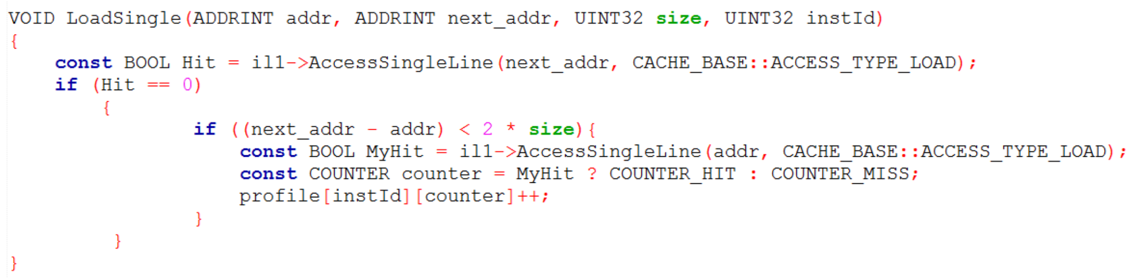
As mentioned in the previous section, Pin is a powerful instrumentation tool which evaluates and analyze a program at instruction level. As the provided tools are written in C++, a language that I am familiar with, the Intel manual [5] that was documented well, was informative and provided with the knowledge to help modify the pintool accurately.

As documented in the pin tutorial, there are two components of the instrumentation Pintool code: *instrumentation* and *analysis*. Instrumentation decides where and what code is inserted at the location,it would be called to evaluate the properties of the code to be presented, while the analysis determines the code to be executed at the insertion point, collects information about the program to be analyzed[5]. Two of three of the prefetching algorithms, next-line prefetching and target-line prefetching, are implemented in both the analysis and instrumentation routine, while the wrong-path prefetching scheme was implemented using only instrumentation.

The instrumentation API offered by Pin gives pintool the ability to inspect and instrument one instruction at a time. The API was utilized in this project when accessing into the address of the next instruction in line, the target of the branch instruction, and determining if the instruction is a branch etc.

- ### Next-Line Prefetching

Next-line prefetching the simplest method of prefetching. If the following address line of the current line is not in cache, it would be fetched when the distance between the current address and the following one fall within a fetchahead distance. As seen in the code shown below, the `LoadSingle` function used to only contain the Boolean function where determines if the input address is a hit or not, and a profile counter. To accomplish the functions of the next-line prefetching, when a miss occurs to the next address, an if statement determines if the fetchahead distance is less than twice (just for an example) of the address size. If yes, the current address will be fetched.

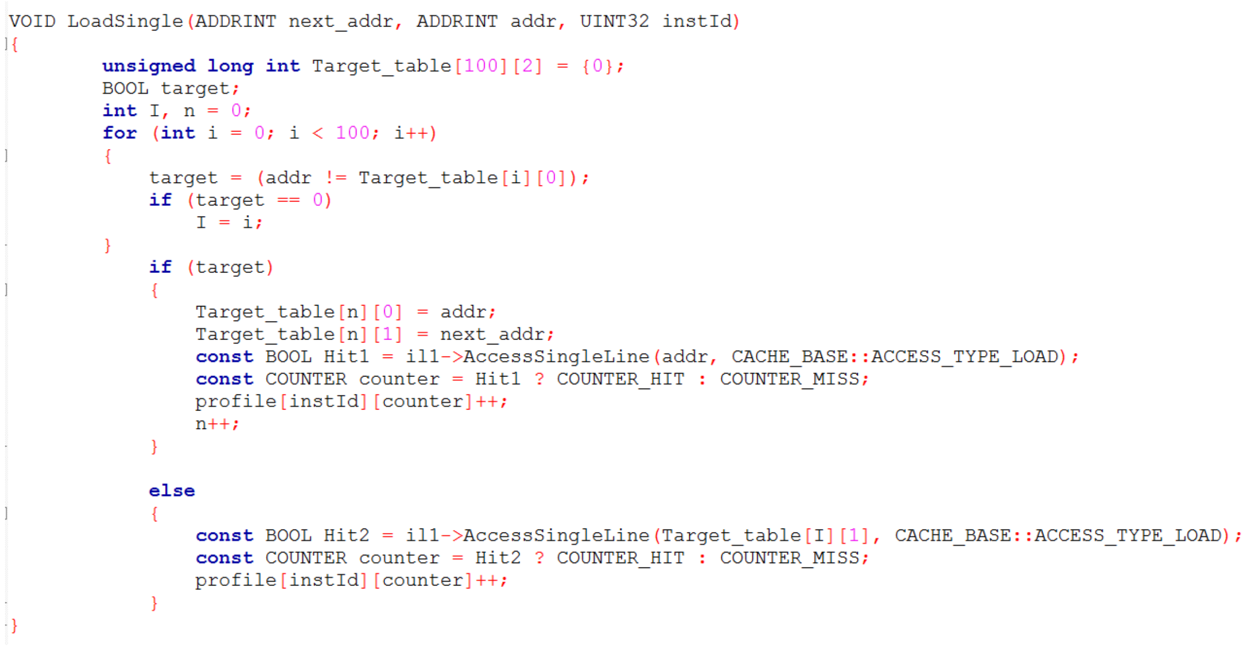


```
VOID LoadSingle(ADDRINT addr, ADDRINT next_addr, UINT32 size, UINT32 instId)
{
    const BOOL Hit = il1->AccessSingleLine(next_addr, CACHE_BASE::ACCESS_TYPE_LOAD);
    if (Hit == 0)
        {
                if ((next_addr - addr) < 2 * size){
                    const BOOL MyHit = il1->AccessSingleLine(addr, CACHE_BASE::ACCESS_TYPE_LOAD);
                    const COUNTER counter = MyHit ? COUNTER_HIT : COUNTER_MISS;
                    profile[instId][counter]++;
                }
         }
}
```

Fig 1. C++ code of next-line prefetching in analysis routine

- **Target-Line Prefetching**

Target-line prefetching has the ability to prefetch non-sequential cache lines. This method utilizes a target table, with two entries on each row, address and successor address, to look up the previously entered cache lines and the next cache lines. The current address cache line is looked up in the target table, if it exists, the successor would be prefetched into the cache, if not, it would update the target table and would be available if the address is accessed once again. In order to reduce cache pollution and reduce cache space, for this experiment the size of the target table is set to be 100.



```
VOID LoadSingle(ADDRINT next_addr, ADDRINT addr, UINT32 instId)
{
        unsigned long int Target_table[100][2] = {0};
        BOOL target;
        int I, n = 0;
        for (int i = 0; i < 100; i++)
        {
            target = (addr != Target_table[i][0]);
            if (target == 0)
                I = i;
        }
            if (target)
            {
                Target_table[n][0] = addr;
                Target_table[n][1] = next_addr;
                const BOOL Hit1 = il1->AccessSingleLine(addr, CACHE_BASE::ACCESS_TYPE_LOAD);
                const COUNTER counter = Hit1 ? COUNTER_HIT : COUNTER_MISS;
                profile[instId][counter]++;
                n++;
            }

            else
            {
                const BOOL Hit2 = il1->AccessSingleLine(Target_table[I][1], CACHE_BASE::ACCESS_TYPE_LOAD);
                const COUNTER counter = Hit2 ? COUNTER_HIT : COUNTER_MISS;
                profile[instId][counter]++;
            }
}
```

Fig 2. C++ code of target-line prefetching in analysis routine

- **Wrong-path prefetching**

Wrong-path prefetching is a combination of next-line and target-line prefetching, while it focuses mainly on the wrong paths. The successor line would be prefetched when the instructions are within the fetchahead distance, as mentioned in the next-line prefetching scheme. However, the target line address are not accessed through the pre-maintained target table. Once a brunch instruction is recognized, the target address line is prefetched immediately. If the instruction line is not a branch, the next “natural” address would be prefetched using the next-line prefetching method.

The advantage of using the wrong-path prefetching is that compared to target-line prefetching it would require less hardware resources, while it would be more accurate and efficient compared to the next-line prefetching. All branches are fetched without considering the predicted direction.

In the C++ code demonstrated below, as the variables of the instruction itself, `ins`, could not be passed to the instrumental function `InsertCall` to be passed on to the analysis routine, thus the function could only be applied in the instrumental routine.

```
VOID Instruction(INS ins, void * v)
{
    // map sparse INS addresses to dense IDs
    const ADDRINT iaddr = INS_Address(ins);
    const UINT32 instId = profile.Map(iaddr);

    const UINT32 size   = INS_Size(ins);
    const BOOL   single = (size <= 4);

    if (KnobTrackInsts) {
        if (single)
        {
                if(INS_IsBranch(ins) && INS_HasFallThrough(ins)){ //if is taken branch
                    BOOL my_Hit1 = il1->AccessSingleLine(INS_DirectBranchOrCallTargetAddress(ins), CACHE_BASE::ACCESS_TYPE_LOAD);
                    const COUNTER counter = my_Hit1 ? COUNTER_HIT : COUNTER_MISS;
                    profile[instId][counter]++;
                }
                else{
                    BOOL my_Hit2 = il1->AccessSingleLine(INS_NextAddress(ins), CACHE_BASE::ACCESS_TYPE_LOAD);
                    const COUNTER counter = my_Hit2 ? COUNTER_HIT : COUNTER_MISS;
                    profile[instId][counter]++;
                }
        }
        else {
            INS_InsertPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR) LoadMulti,
                                     IARG_UINT32, iaddr,
                                     IARG_UINT32, size,
                                     IARG_UINT32, instId,
                                     IARG_END);
```

Fig 3. C++ code of wrong-path prefetching in analysis routine

Results and Analysis

- **Analysis routine modification prefetching tested on Barnes**

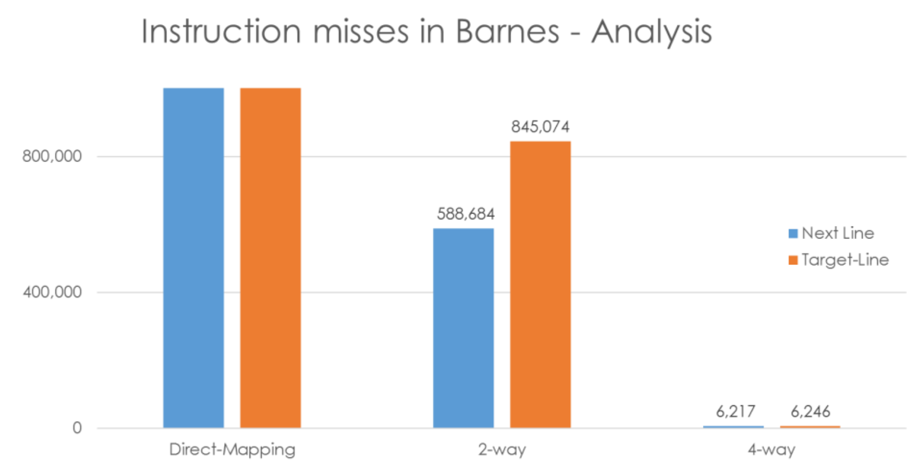


Fig 4. Plot of prefetching function implemented in the analysis routine for the Barnes Benchmark

It could be clearly observed above that when using direct-mapping, the hit rates are exponentially higher than when using two and four way set associative. The average miss rates when using direct-mapping for both prefetching algorithms are more than 20% (Next-line: 21.61%, target-line 24.23%). This is reasonable as each set has more blocks, there's a less chance of a conflict between two addresses.

- **Instrumentation routine modification prefetching tested on Barnes**

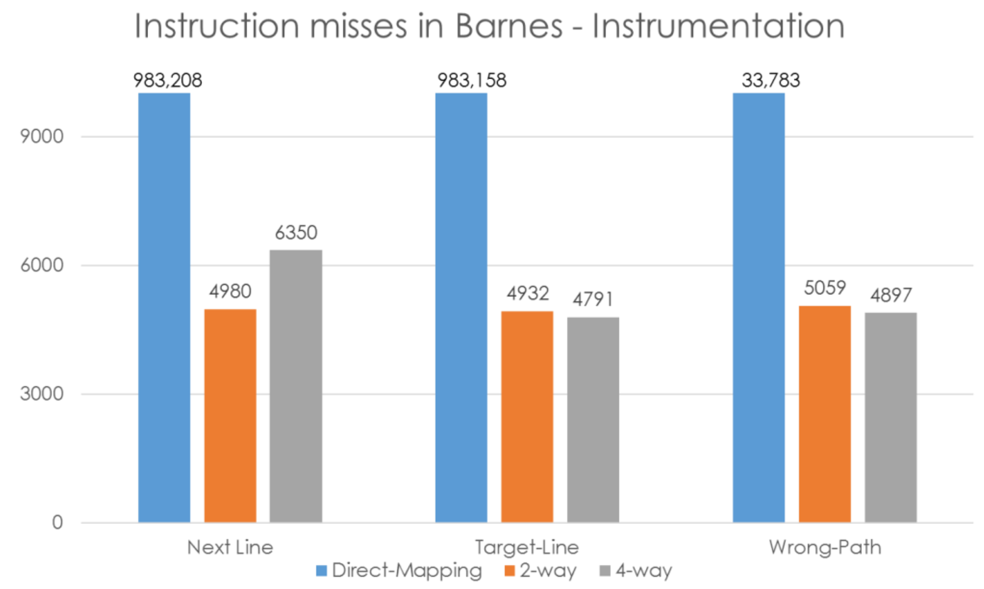


Fig 5. Plot of prefetching function implemented in the instrumentation routine for the Barnes Benchmark

As seen from the plot above, it could be concluded that still, direct mapping has a much higher rate of cache misses. However, as we could put wrong-path prefetching into comparison with the latter two prefetching methods, it wrong path does in fact has a lower and stable number of cache misses, especially in direct mapping. It could be clear as well, that the prefetching methodology would be more accurate if implemented in the instrumentation routine, where the routine would be executed to decide if and where to add in calls to analysis functions.

- **Analysis and Instrumentation routine modification prefetching tested on FindPi**

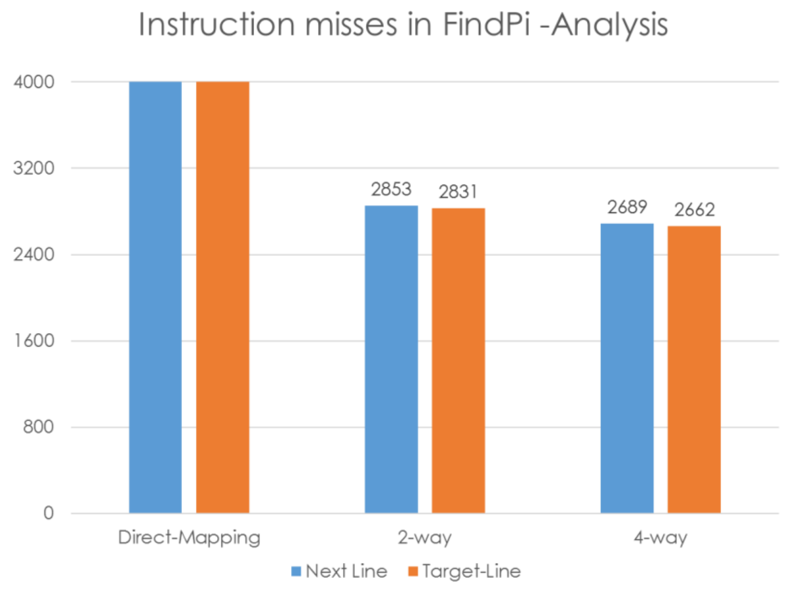


Fig 6. Plot of prefetching function implemented in the analysis routine for the FindPi Benchmark

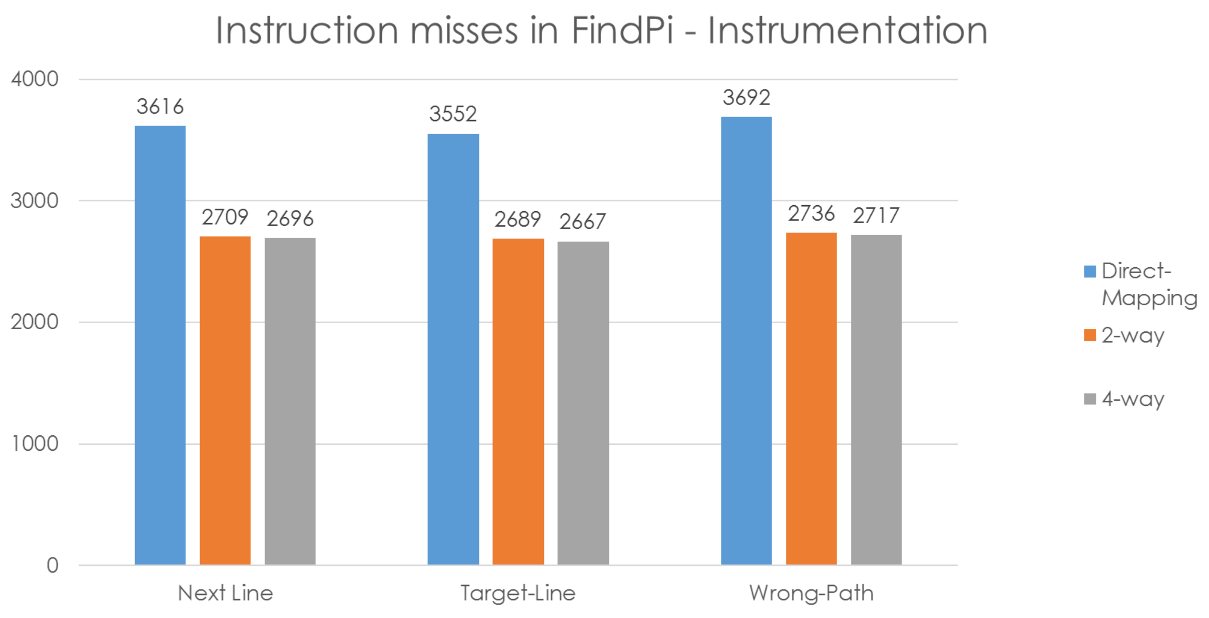


Fig 7. Plot of prefetching function implemented in the instrumentation routine for the FindPi Benchmark

The trends for the FindPi benchmark was similar of that of the Barnes benchmark: very high miss rates for direct-mapping, and not much difference between the next-line, target-line and wrong-path prefetching method. The reason for the limited difference that was obtained is that the wrong-path method works better only when there exists a large disparity between the CPU cycle time and the memory speed. The reason for this would be that the memory might not be fast enough for the prefetching to start, which is designed to prefetch target addresses that would be executed almost immediately.

For this experiment, only instructions that are single (size ≤ 4) are implemented with the prefetching schemes. In other words, in other multi-instructions operations, there might be a considerable room for improvement that could be done to increase hit rates. The `INS_InsertPredicatedCall` function that is called in the instrumental routine for every function "avoid generating references to instructions that are predicated when the predicate is false" would greatly decrease the number of misses in multi-instructions.

Conclusion

In this project, three prefetching algorithms are successfully implemented using Intel's instrumental tool, Pin, by modifying an instruction cache simulator, icache and the results using 2 benchmarks, Barnes and FindPi, are used to testify the performance of the algorithm.

Most of the time spent on this project was on how to understand the icache pintool that was provided as a resource, without too much information other than the tutorial that was provided by Intel. I went through the variables in the header files and other pintool examples that was on the tutorial to help me better understand the program well. One lesson I learnt is that one could never be impatient when it comes to interpreting someone else's code. The key is to be familiar with the relevant information first, then think like how the author is thinking.

Reference

1. “Wrong-Path Instruction Prefetching”, Jim Pierce, Trevor Mudge, Department of Electrical Engineering and Computer Science, University of Michigan, Arbor, 1996

2. “Miss Caches, Victim Caches and Stream Buffers - Three optimizations to improve the performance of L1 caches”, slides for course CSE240, University of California San Diego.

3. *PIN*, Intel developer zone, https://softwhttps://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-toolare.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
4. “Barnes-Hut-SNE”, Laurens van der Maaten, Pattern Recognition and Bioinformatics Group, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands

5. Pin 3.0 User Guide(Kit 76991 Pin Manual), Intel, https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool