Implementation of Effective Hardware-Based Data Prefetching for High-Performance Processors

Alireza Nazari, AmirSaber Sharifi,Bo Gao
Computer Department

New Mexico State University
Las Cruces, New Mexico

***Abstract*— Due to progress pace of processors performance which is higher than memory latency, need of functions which help to reduce this gap is increasing every day. Hardware-Based data prefetching is one of the approaches to make this gap smaller than before. We decided to have a closer look at basic and correlated prefetching and analyze how they will improve performance in memory latency. Basic prefetching method tries to provide prefetching by taking into account adjacent data access by storing the strides and previous address while the correlated approach tries to take advantage of not only the adjacent accesses but also of those correlated changes in the outer loop level.**

In this paper, implementation and analyze of simulation in GEMS–General Execution-driven Multiprocessor Simulator- are taken into consideration. We simulated a 16-core system and after modifying cache design in GEMS and adding Data Prefetching into it, we analyzed data by using PARSEC 2.0 benchmark.

By evaluation of the result we observed that basic prefetching scheme cannot be helpful on number of misses while correlated scheme can reduce it in most of the cases. But anyway on a multicore network neither have enhancement effect on speedup. There still huge amount of data which can be analyzed in future works.

Keywords—Hardware Prefetching, Multicore Simulator, High Performance Processor

I. INTRODUCTION

Memory latency and bandwidth have progressed over the last few years but at a much slower pace than processor performance. To bridge this gap using more efficiently of cache has been shown to be an effective way. Caches are used to reduce main memory access however it cannot decrease memory latency thus it is essential to find techniques to reduce memory access as much as possible.

There are several techniques that would help in reducing memory access and among these techniques data prefetching is one of the techniques that ideally can make memory access latency to zero [3]. There are two approaches for data prefetching, software and hardware data prefetching. The former should be done by compiler and needs a static program analysis to detect regular data access pattern while the latter detects accesses with regular patterns and issues prefetches at run time. Both of these approaches have advantages and drawbacks. To demonstrate how they really work in this paper we will concentrate on hardware-based data prefetching [1].

Hardware-Based prefetching can be categorized into spatial and temporal. The former is where access to the current block the basis for the prefetch decision and the latter is where lookahead decoding of the instruction stream is implied. There are three approach to hardware-based prefetching which are named, Basic, Lookahead and Correlated. The basis for the three designs is an RPT - Reference Predication Table - that holds data access patterns of load/store instructions and it is organized as an instruction cache. In this paper our concentration is on implementing basic and correlated design.

The basic idea of reference predication is to predict future references based on history for the same memory access instruction. When PC – program counter – detects a load/store, there would be a check in RPT to see if there is an entry corresponding to the instruction or not. If not, it would be entered in the table. If it is there and the reference for the next iteration is predictable a prefetch is issued. In the basic scheme only PC and RPT are involved. As shown in figure 1 RPT and it's states are illustrated [1].

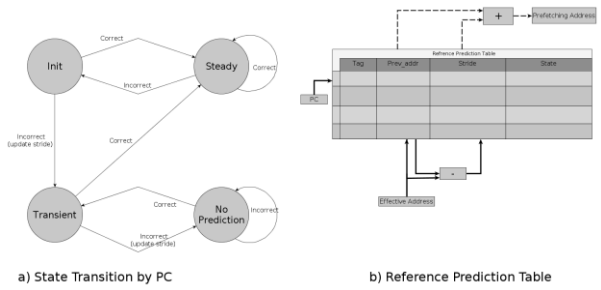

Fig. 1. Reference Predication

In basic design, reference predication is based on the regularity between adjacent data accesses. The key idea behind correlated reference prediction is to keep track not only of those adjacent accesses in inner loop, but also of those correlated by changes in the loop level. Since branches in the inner loop are taken until the last iteration, a not-taken branch will trigger the correlation to the next level up. By adding a shift register to record the outcome of the last branches and an extended RTP with separate fields for computing the strides of the various correlated accesses correlated design is

implemented. But to avoid prefetching too far in advance correlation in two-level nested loops is a limitation in this design. Figure 2 shows extended RPT which is used in correlated design.

In this paper we will focus on these two designs and implement them. Implementation has been done by GEMS simulation by adding our customized file to a folder named "ruby/system". The most common file is CS573Prefetching.h which is a header file that contains common data structures. Then there are four files that implemented basic prefetching algorithm and correlated mechanism. For basic design BasicReferencePredictor.h, BasicReferencePredictor.C and for correlated design two files, CorrelatedReferencePredictor.h, CorrelatedReferencePredictor.C, are the files we implemented. In section II design and implementation of these files and related function are described more.

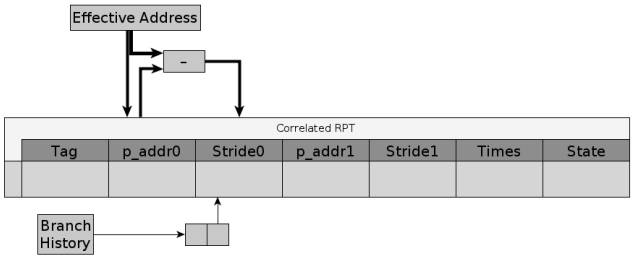

Fig. 2. Correlated RPT

The new cache design is simulated through SIMICS[4] and PARSEC 2.0 benchmark to get metrics and analyze them. In a configuration file which is used by SIMICS we configured chips, caches and memories. In GEMS configuration file we described number of cores, size of L1 cache, size and number of L2 caches and interconnection network. To analyze data the metrics we need number of issued prefetches and cache misses. To count these numbers in our simulation we implemented a customized profile under "ruby/profiler" with Profile.c and Profile.h files. These two files help us to print number of issued prefetches and cache misses in each core after of each benchmark program. In section III these analysis and graphs are shown and described more.

II. DESIGN

A. *GEMS Structure and Simulation Configurations*

As Figure 3 shows, the heart of GEMS is the Ruby memory system simulator. As illustrated in Figure 3, GEMS provides multiple drivers that can serve as a source of memory operation requests to Ruby [2].

Ruby is a simulator of a multiprocessor memory system that models: caches, cache controllers, system interconnect memory controllers, and banks of main memory. Ruby combines hard coded timing simulation for components such as interconnection network which are independent of the cache coherence protocol with the ability to specify the protocol-dependent components such as cache controller in a domain-specific language called SLICC (Specification Language for Implementing Cache Coherence). SLICC is syntactically similar to C or C++, but it is intentionally limited to constrain the specification to hardware-like structures. For example, no local variables or loops are allowed in the language.

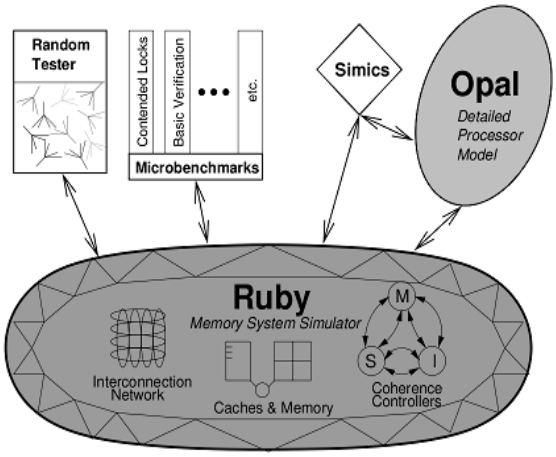

Fig. 3. Ruby's Structure and its drivers

One of the drivers on Ruby is SIMICS, which uses SIMICS' functional simulator to approximate a simple in-order processor with no pipeline stalls. SIMICS passes all load, store, and instruction fetch requests to Ruby, which performs the first level cache access to determine if the operation hits or misses in the primary cache. Ruby receives the requests and put them in a FIFO manner. On a hit, SIMICS continues executing instructions, switching between processors in a multiple processor setting. On a miss, Ruby stalls SIMICS' request from the issuing processor, and then simulates the cache miss. So while the SIMICS is executing the benchmark in a multi core manner, ruby simulates the memory behavior to send SIMICS the stalls due to the memory. Each processor can have only a single miss outstanding, but contention and other timing affects among the processors will determine when the request completes [3].

In Table 1 the configuration of simulated system is proposed. This configuration contains number of cores, memory hierarchy arrangement and network protocols. The simulator provides CPU cores with memory hierarchy and a profiler which is implemented to give us access to statistical information of operations. To evaluate the system, we used PARSEC 2.0. In PARSEC there are about 13 programs, and among these programs, simulation is done under only 11 of them because of availability of checkpoints and analyze of data has been done through implementing a customized profile which help us to count number of issued prefetches and cache misses [4].

Core model	PowerPC Simple
Number of Cores	16
Private I/D L1$	32KB, 4-way, LRU
Shared L2 per bank	1024KB, 16-way, LRU
Cache block size	64 Bytes
Cache Coherence Protocol	MOESI
Network topology	MESH

Table 1. The Most important configurations of simulation.

B. *RPT data structure*

RPT is the most important part of data prefetching, so we designed an RPT data structure which holds data access pattern of load/store instructions. Each entry of this table contains a tag related to the instruction address, field to record the memory operand address and its stride and a state transition field. In code 1, the code related to our data structure which can be find in file "ruby/system/BasicReferencePredictor.h" is shown

```
01 RPT_Entry {
02      public:
03              Address PC;
04              Address prev_addr;
05              unsigned long stride;
06              EntryState state;
07 };
08 map< Address, RPT_Entry* > _rpt;
```

Code. 1. Basic RPT data Structure

In line 03 Address PC is defined which is responsible of storing tag of each instruction. In 04 we defined the last address that was referenced when the PC reach that instruction as prev_addr. Line 05 represent the difference between the last two addresses that were generated and 06 line is holding state and transitions to prevent from false prefetching. So when PC decodes a load/store instruction, a check is made to see if there is an entry corresponding to the instruction in RPT. If there is according to state of that entry prefetching would be issued and if not it would enter it in table with state of init. State of each entry is implemented by EntryState data structure which is shown below in Code 2.

```
01 enum EntryState {
02      INITIAL,
03      TRANSIENT,
04      STEADY,
05      NOPREDICTION
06 };
```

Code. 2. Data structure of EntryState

In correlated design there is a difference because it should keep track of two nested loop levels and for each loop RPT entry is using the basic RPT data structure. So for implementing an extended RPT for correlated design we used the previous data structure and made a new data structures which is shown in code 3. Correlated codes are located in "ruby/system/CorrelatedReferencePredictor.h".

```
01 class RPT_Entry_Ext {
02      public:
03              RPT_Entry outer_branch;
04              RPT_Entry inner_branch;
05              BranchRegState branch_reg;
06 };
```

Code. 3. Basic RPT data Structure

As it's shown for each outer branch and inner branch a RPT_Entry is defined and for each entry a two bit branch register is used which is called branch_reg to keep track of loops to see if inner or outer loop should be considered for issuing prefetching. This two-bit branch register is implemented as a data structure which is shown below in Code 4.

```
01 enum BranchRegState {
02      OUTER_TAKEN_INNER_NOTTAKEN, // 10
03      OUTER_TAKEN_INNER_TAKEN, // 11
04      OUTER_NOTTAKEN_INNER_NOTTAKEN, // 00
05      OUTER_NOTTAKEN_INNER_TAKEN // 01
06 };
```

Code. 4. Data Structure of two bit branch register named BranchRegState

C. *Detecting load/store instruction*

One thing that is important here is how to detect that an instruction is a load/store instruction. To lookup or add entry into RPT first of all we should detect load/store instructions. In "ruby/system/Sequencer.C" there is a function which is named tryCacheAccess and by this function we can capture the cache access operation. Thus, full address, cache line address and PC of the cache request message – CacheMsg – can be obtained. By making an object of CacheMsg class functions of getAddress(), getProgramCounter() and line_address() can be used to get more information about cache request. For both designs, basic and correlated, the way we detect load/store instruction is the same.

D. *Checking entry of RPT*

In our design, we use program counter to index the RPT entry. As shown in Code. 1 line 08 there is a map between PC and RPT entry. Thus, to see if an entry exist or not, we use "RPT_ENTRY* an_entry= _rpt[PC];" if corresponding entry is there, we access the recorded referenced pattern. If not we initialize an entry by using code shown in code 5.

```
01 _rpt_entry = new RPT_Entry();
02 _rpt_entry->PC = PC;
03 _rpt_entry->prev_addr = full_addr;
04 _rpt_entry->stride = 0;
05 _rpt_entry->state = INITIAL;
06 _rpt.insert(makepair(PC,_rpt_entry));
```

Code 5. adding a new entry into prediction reference table

In correlated design there is a small difference that is related to branch shift register and data structure of extended rpt. The following Code 6 indicates the code that is used to initialize a RPT entry in correlated design.

```
01 RPT_Entry_Ext* m_rpt_entry = new RPT_Entry_Ext();
02 m_rpt_entry->outer_branch.PC = PC;
03 m_rpt_entry->inner_branch.PC = PC;
04 m_rpt_entry->outer_branch.prev_addr = full_addr;
05 m_rpt_entry->outer_branch.stride = 0;
06 m_rpt_entry->outer_branch.state = INITIAL;
07 m_rpt_entry->inner_branch.prev_addr = full_addr;
08 m_rpt_entry->inner_branch.stride = 0;
09 m_rpt_entry->inner_branch.state = INITIAL;
10 m_rpt_entry->branch_reg=OUTER_TAKEN_INNER_NOTTAKEN;
11 m_rpt.insert(make_pair(PC,m_rpt_entry));
```

Code 6. Adding a new entry into prediction reference table in correlated

In the next iteration when PC reach a load/store instruction which is already in RPT the thing that is important is to determine if the reference for the next iteration is predictable or not.

E. *Predicting reference for the next iteration*

After finding an entry in RPT we should be sure that the reference is predictable. This algorithm should work according to state transition of RPT table entires. First we should check the entry of PC by "_rpt_entry = _rpt[PC];" then we use code shown in code 7. This code is used in basic design.

```
01 if(!_isCorrect(PC,_rpt_entry) && (_rpt_entry->state == INITIAL)){
02      _rpt_entry->prev_addr = full_addr;
```

```
03 _rpt_entry->stride = full_addr.getAddress()-_rpt_entry->prev_addr.getAddress();
04 _rpt_entry->state = TRANSIENT;}
05 else if (_isCorrect(PC,_rpt_entry) && (_rpt_entry->state == INITIAL || _rpt_entry->state == TRANSIENT || _rpt_entry->state == STEADY )){
06 _rpt_entry->prev_addr = full_addr;
07 _rpt_entry->state = STEADY;}
08 else if(!_isCorrect(PC,_rpt_entry) && (_rpt_entry->state == STEADY) ){
09 _rpt_entry->prev_addr = full_addr;
10 _rpt_entry->state == INITIAL;}
11 else if(!_isCorrect(PC,_rpt_entry) && (_rpt_entry->state == TRANSIENT)){
12 _rpt_entry->prev_addr = full_addr;
13 _rpt_entry->stride = full_addr.getAddress() - _rpt_entry->prev_addr.getAddress();
14 _ rpt_entry->state = NOPREDICTION;}
15 else if(_isCorrect(PC,_rpt_entry) && (_rpt_entry->state == NOPREDICTION)){
16 _rpt_entry->prev_addr = full_addr;
17 _rpt_entry->state = TRANSIENT;}
18 else if(!_isCorrect(PC,_rpt_entry) && (_rpt_entry->state == NOPREDICTION)){
19 _rpt_entry->prev_addr = full_addr;}
```

Code 7. Predicting reference for an entry in RPT

In correlated design there is a two bit shift register which plays a key role in predicting the reference because it will decide that if outer loop or inner loop is going to be used. Code 8 shown implemented code in correlated for predicting reference.

```
01 void CorrelatedReferencePrediction::updateRPT(const Address& PC, const Address& full_addr, Branch brn){
02 if(brn == OUTER)
03 updateRPTBasic(&(m_rpt[PC]->outer_branch), full_addr);
04 else
05 updateRPTBasic(&(m_rpt[PC]->inner_branch), full_addr);}
06 void CorrelatedReferencePrediction::updateRPTBasic(RPT_Entry* m_rpt_entry, const Address& full_addr){
07 if(!_isCorrect(full_addr,m_rpt_entry) && (m_rpt_entry->state == INITIAL){
08 m_rpt_entry->prev_addr = full_addr;
09 m_rpt_entry->stride = full_addr.getAddress() - m_rpt_entry->prev_addr.getAddress();
10 m_rpt_entry->state = TRANSIENT;}
11 else if (_isCorrect(full_addr,m_rpt_entry) && (m_rpt_entry->state == INITIAL || m_rpt_entry->state == TRANSIENT || m_rpt_entry->state == STEADY )){
12 m_rpt_entry->prev_addr = full_addr;
13 m_rpt_entry->state = STEADY;}
14 else if(!_isCorrect(full_addr,m_rpt_entry) && (m_rpt_entry->state == STEADY) ){
15 m_rpt_entry->prev_addr = full_addr;
16 m_rpt_entry->state == INITIAL;}
17 else if(!_isCorrect(full_addr,m_rpt_entry) && (m_rpt_entry->state == TRANSIENT)){
18 m_rpt_entry->prev_addr = full_addr;
19 m_rpt_entry->stride = full_addr.getAddress() - m_rpt_entry->prev_addr.getAddress();
20 m_rpt_entry->state = NOPREDICTION;}
21 else if(_isCorrect(full_addr,m_rpt_entry) && (m_rpt_entry->state == NOPREDICTION)){
22 m_rpt_entry->prev_addr = full_addr;
23 m_rpt_entry->state = TRANSIENT;}
24 else if(!_isCorrect(full_addr,m_rpt_entry) && (m_rpt_entry->state == NOPREDICTION)){
25 m_rpt_entry->prev_addr = full_addr;}
26 }
```

Code 8. Predicting reference in Correlated design

As it shown in first function, the first decision is about outer loop and inner loop, and base of that the outer RPT or inner RPT in extended RPT should be updated. Then after choosing the outer or inner table, it is just the same is basic prediction.

F. Result from prefetching

The prefetching prediction scheme is reference pattern table. RPT is a small memory indexed by the PCor low-order bits of program counter of the memory access instruction. The table contains two bits that say whether the prefetching was recently taken or not. Here, in our design we view the size of this prefetching prediction table as a design parameter.

With a limited size of RPT, we don't know, in fact, if the prefetching prediction is correct – it may have been put there by another memory access instruction that has the same low order address bits, since with limited size of RPT, low-order bits of memory access instruction are used to index the an entry of RPT. But this doesn't matter. The prediction is a hint that is assumed to be correct, and the prefetching begins in the predicted direction. If the hint turns out to be wrong, the prediction status is transitioned and stored back.

The accuracy using basic algorithm for integer programs, which typically have higher branch frequencies, is lower than for the loop-intensive scientific programs. As we try to exploit more accuracy of prefetching, we can increase the size of the RPT and improve the predictor structure. However, a RPT table with size of 4K entries performs quite comparably to an infinite size of RPT. Therefore, simply increasing the number of entries without changing the predictor structure also has little impact.

III. RESULTS

A. Number of misses per thousand of instructions

Let's look at a single core where ideal hardware prefetching supposed to omit all the misses by prefetching data early enough or at least to decrease the average penalty due to the cache miss as much as it can, while it keeps the number of misses constant. But we know that implementation of prefetching in non-ideal world cannot keep the number of misses same as non-prefetching approach because obviously bringing a new block to the finite-capacity cache means evacuation of an existing block and this block might be still needed by program and soon we have to load it to the cache (prefetch it!) which force stalls to the memory. So in some cases prefetch might increase the number of misses by polluting the cache.

Our result of basic reference predictor (Figure 4) shows that while we have up to 14 percent decrease of cache misses (bodytrack), we have up to 7 percent incensement of miss per thousand instructions (facesim) due to cache pollution effect of prefetching. In general, the schemes work well for predicting references in inner loops and less effective for those inner loops with small bodies. We can conclude that effect of pollution due to prefetching plus this fact that BRP cannot help small-body loops at all, had more effect than prefetching's advantage on reduction of number of misses, consequently not only the number of misses didn't reduced, but also had a growth.

In Figure 5, correlated prefetching data has an absolutely better performance since it reduced the number of misses per thousand instructions in all of the benchmarks up to 73 percent (stream cluster). This result shows absolute win of CRP against BRP method since CRP gains of not only track of adjacent accesses in inner loops but also of those correlated changes in the loop level. Since branches in the inner loop are taken until the last iteration to the next level up.

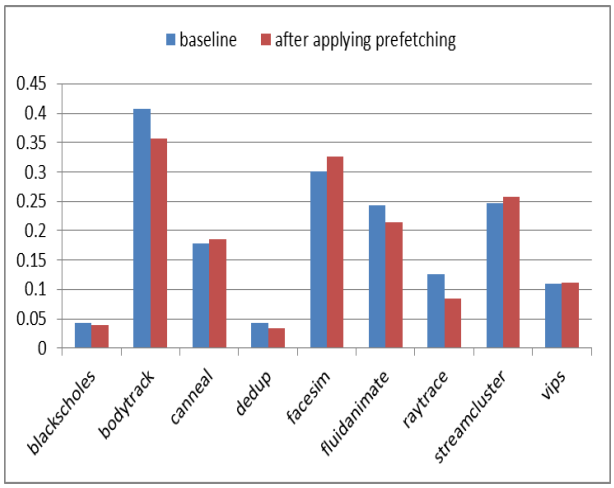

Fig. 4- Number of misses per thousand of instructions in BRP method

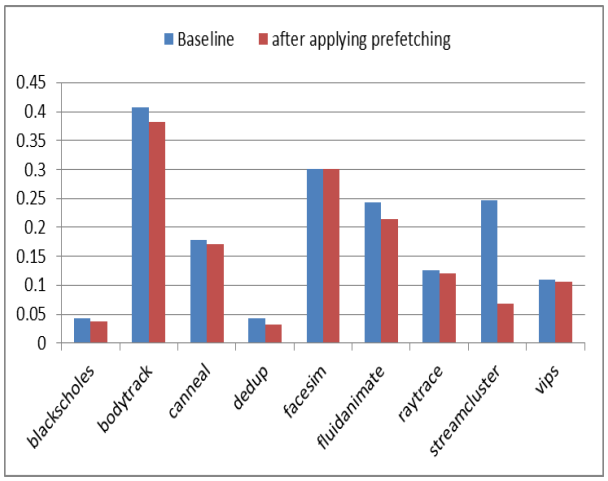

Fig. 5- Number of misses per thousand of instructions in CRP

B. *Execution time*

As we see in Figure 6, the execution time of BRP is increased. It can be easily interpreted due to increase of cache misses. Not only it didn't help, but also made the execution time worse. But the interesting result can be seen in Figure 7, where while we had reduction of cache misses, still execution time increased and prefetching didn't help us.

When we want to interpret the execution time data we should consider an essential point, the simulated CPU is multicore. The total goal of prefetching is reducing the amount of time, wasted in CPU, due to memory stalls by loading the target cache block enough cycles before reaching the program counter to associated load/store instruction. Obviously in a uni-core system, with certain amount of miss rate, reduction of memory stalls improves the speedup of system. But in a multicore system there is no guarantee to have a reduction of execution time by applying prefetching since adding prefetching instructions means adding more memory instructions and more memory instructions means more communication between the cores in network due to data request and data response of cache coherency protocols. Traffic increase of a system reduces the whole network efficiency and we know this fact that relation of average load rate and latency is exponential. So no wander if we have too much higher latency by increase of a small amount of traffic.

Streamcluster benchmark is a good example for this claim that even CRP prefetching didn't lead to speed up. We can easily see that while huge amount of miss number reduction is performed, still the execution time increased and this is because of grows of traffic profound descent of network throughput.

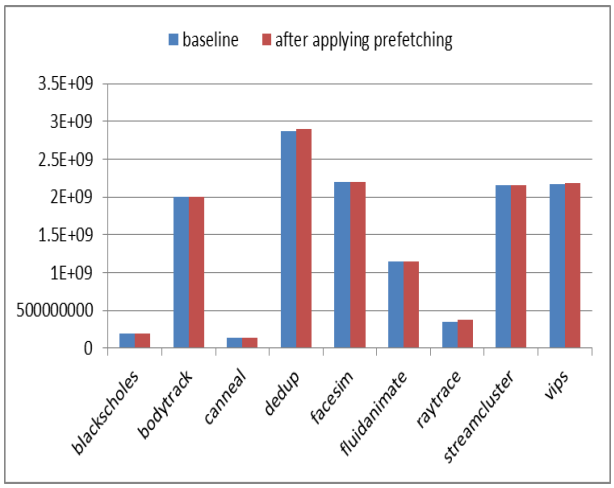

Fig. 6-CRP Execution time

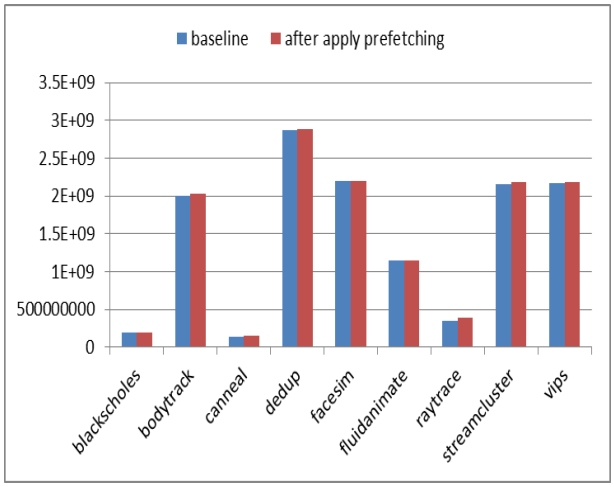

Fig. 7-CRP Execution time

During the implementation we found that in some cases we can prevent from ineffective prefetching instructions. Assume cases that we issue the address that we need to prefetch and that address is in the same block as what we are right now, it means that the current load/store instruction is hit and we issued the request for next memory instruction. We should check the stride and current address and see whether we go further than existing block or not, and if not, we don't need to issue a new prefetching request. Although this prefetch instruction doesn't have significant effect on single core performance but in multicore we expect more effect by reducing the amount of communication.

IV. Conclusion

In this project we implemented and evaluated the design for hardware based prefetching scheme, which is presented by "Effective Hardware-Based Data Prefetching for High-Performance Processor". The purpose of this prefetching mechanism is to reduce cache misses which would lead the increase of CPI. The basic idea behind this mechanism is to predict next data address that will be referenced by next load/store instruction. In order to do this, this design uses a Reference Prediction Table to keep track of past data access pattern. We have applied this design into a cycle-accurate simulator- GEMS. We run the simulation and the results show that basic prefetching schema increases the number of misses in the program by polluting the cache and this fact that it cannot help small-body loops. Also results show that the CRP schema can decrease the number of misses by taking to account the correlation between inner and outer loops.

In the other hand, observation showed that not only BRP cannot enhance efficiency, but also it make the execution time longer by increasing the number of misses. This observation showed that even CRP which decreases the number of misses cannot be helpful when we are using it in a multicore network, because prefetching force a communication overhead to network which decreases the throughput of network exponentially.

References

[1] Tien-Fu Chen; Jean-Loup Baer; , "Effective hardware-based data prefetching for high-performance processors," *Computers, IEEE Transactions on* , vol.44, no.5, pp.609-623, May 1995

[2] Milo M. K; Martin, Daniel J.; Sorin, Bradford M.; Beckmann, Michael R.; Marty, Min Xu,; Alaa R.; Alameldeen, Kevin E.; Moor, Mark D.; Hill, and David A. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset", Computer Architecture News Volume 33 Issue 4, pp 92-99, November 2005

[3] Magnusson, P.S.; Christensson, M.; Eskilson, J.; Forsgren, D.; Hallberg, G.; Hogberg, J.; Larsson, F.; Moestedt, A.; Werner, B.; , "Simics: A full system simulation platform," *Computer* , vol.35, no.2, pp.50-58, Feb 2002

[4] Bienia, C.; Kai Li; , "Fidelity and scaling of the PARSEC benchmark inputs," *Workload Characterization (IISWC), 2010 IEEE International Symposium on* , vol., no., pp.1-10, 2-4 Dec. 2010