CSCI710 Project Report

Optimizing Shared Cache for Parallel Applications

Wenting Tan
The Colledge of William and Mary
wwtan@email.wm.edu

Abstract

Multicore processors have been widely used for various applications on different types of computing platforms, especially for scientific ones. Improving performance is always challenging while threaded programs are running on multiple cores. Since the last level cache is shared by multiple cores, the access contention happens frequently in the limited cache space. As for this problem, the existing approaches tend to involve heavy-weighted memory analysis and large overhead. In this project, we optimize shared the last level cache for applications in SPEC2006 benchmarks and reduced cache pollution by having dominant non-temporal memory accesses bypass the cache. We collect the data-centric data produced by a lightweight performance measurement tool – HPCToolkit. Based on the generated databases, we determine the data that has poor memory locality through differential analysis. We utilize the Extensible Micro-Architectural Optimizer (MAO) to have weak locality data bypass the last level cache. MAO is a tool which can be integrated into compiler to insert cache bypassing instructions automatically in any application program and generate the assembly code. We evaluate the results for benchmarks in SPEC2006. After a series of experiments and the data comparison, we see significant performance improvement through cache bypassing implementation. Some of the applications achieve up to 11% speedup.

Keywords Cache bypassing, data memory locality, performance, differential analysis

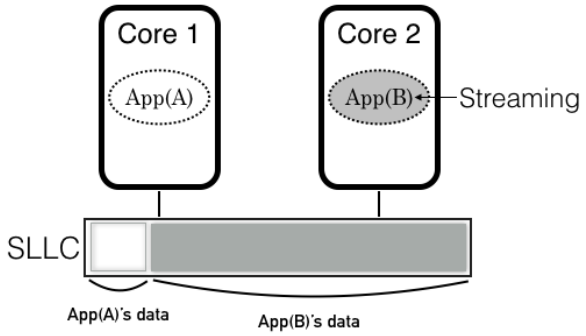

Figure 1. Cache structure of dual-core machine.

1. Introduction

Multicore processors[1] are being widely used in modern times. They are designed for running multiple instructions simultaneously on one machine. In this way, parallel computing is expected to increase the overall executing speed when multiple programs are running at the same time. Since programs running on multiple cores share the same last level cache and the cache space is limited, cache contention always exists. Figure 1 shows the structure of last level cache which is shared by two cores. Cache contention becomes severe when app(B) has streaming data in the program.

To achieve greater efficiency, a significant number of applications, especially scientific applications, have been using parallel programming techniques which are expected to give better computing performance. As a program becomes parallelized, multiple threads are running in parallel. In this case, the last level cache becomes the shared resource which affects performance dominantly. Besides, when different applications are running in parallel, the cache contention becomes even worse because the last level cache is shared by different programs. As a result, the access contention in the

shared last level cache (SLLC) is normally the main reason of computing performance degradation.

Cache management is a crucial approach that we used to deal with cache contention for the scenarios that we discussed above. In this project we are going to present a cache bypassing technique which is able to solve the problem at the software level. Optimization is achieved by having dominant non-temporal memory accesses in the program bypass cache. HPCToolkit [2] is a performance measurement tool we used to collect the performance data from computer hardware. We also used differential analysis method to determine the streaming data in certain type of applications that are considered as potential candidates for our cache optimization. An Extensible Micro-Architectural Optimizer (MAO), which is developed by a research program at Google [3], can be integrated into compiler to insert cache bypassing instructions semi-automatically on any application program and generate assembly code respectively. We tested the performance and ran the optimized benchmarks on x86 multicore machine, and from the optimization results we saw that the performance can be improved as high as 10%.

In this project, a series of experiments were conducted to evaluate the performance gain after the optimization. We sum up three main contributions from this research:

- We invented an instruction-centric cache optimization technique which can be implemented semi-automatically at the software level.
- We applied a light weight data-centric measurement tool to collect performance data from the program's profile. Additionally, we proposed a differential analysis method to determine the non-temporal memory accesses in applications' program.
- In the evaluation part, we presented results of a series of experiments which include applications co-run experiments. From the results we see that the increase in speed is obvious.

In Section 2, we listed some outstanding research work about cache management. Different techniques were used in their projects to accomplish performance optimization. In Section 3, we introduced a basic cache contention problem. Basically, cache pollution is always the main factor of performance bottleneck. Additionally, we presented some solutions to reduce cache pollution, as well as the tools that are used in this project. Section 4 talks about the technical part of this research. In Section 5 , experimental results for all the benchmarks we evaluated are presented. Finally, Section 6 gives the potential further work for this research.

2. Related Work

Nowadays the SLLC management is a popular topic in which extensive amount of research work is conducted to accomplish performance improvement in different scenarios. The last level cache optimization on multicore was heavily researched, such as software based cache partitioning [4] by Lu. Ding etc. developed ULCC [5] in their research about cache partitioning techniques. This user level cache managing tool can help programmers explicitly on optimizing cache space in their code and programming cache-friendly applications. Andreas Sandberg etc. proposed a low over head software-only method to automatically implement instructions in order to improve the application's performance. [6] The technique is achieved by measuring the stack distance in cache modeling. Jacob Brock etc. presented *Pacman* in their paper [8]. They presented a program-assisted cache management by using profiling techniques to approximate optimal cache management. Each of those used unique methods in their independent research, and gained some performance improvement in certain cases. In our project we used a novel method to implement cache bypass instructions for non-temporal memory accesses, which is considered a new cache management policy.

3. Motivation and Background

3.1 What is the Problem (Cache Pollution)

Multicore processors are challenged by limited SLLC space, and cache space for each thread is not guaranteed. How do we manage the space of the last level cache that is shared among multiple threads? We will discuss this question in this section.

Cache contention in the last level cache is mainly caused by cache pollution. Cache pollution [10] happens while executing a computer program, strong locality data from one process that is stored in a cache line is evicted from the cache due to data from another process is mapped to the same cache line. The strong locality data is going to be frequently used in the program. However, if the strong locality data gets pushed back to the lower level structure such as main memory, then the next time when the program call the strong lo-

cality data, there will be cache misses in SLLC. This is an example from cache pollution wiki page [10]:

```
1 S[0] = S[0] +1;
2 for i in 0...SIZEOF(CACHE)
3   W[i] = foo(W[i]);
4 S[0] = S[0] + W[SIZEOF(CACHE)-1]
```

Where T[0] is the strong locality data, when the code is executed, S[0] will be fetched from main memory into cache first. In the loop, W[i] is the weak locality data and it will fill the entire cache space so that the S[0] has to be evicted from shared cache back to main memory. In the last line of code, when the program request S[0] to be updated from the cache line, then the cache miss happens. In this case the extra time would be spent by the cache controller on bringing the S[0] data block from main memory back to cache.

The performance is affected in this way when many cache misses happen, and almost all the application programs can not explicitly control and optimize the shared cache usage. The condition is more intensive for multi-threaded program than only one program running on single core. Since when cache pollution happens in the shared cache space, multiple threads are going to be badly affected. Another situation is that multiple applications are running on a multicore machine, in which all the cores share the same last level cache. Applications are usually written by different programmers, therefore strong locality data cannot be kept in cache at programmer's will. Cache pollution is main reason of performance degradation for those conditions we discussed above.

3.2 How to Solve the Problem

At user's side, programmers should give sufficient cache space to strong-locality data and minimize the cache space to weak-locality data. In the following we are going to talk about the cache bypassing technique utilized by this project. Since the non-temporal memory access, such as streaming data, is typically weak-locality and rarely used again. It is better to have those accesses not cached in SLLC, so that strong locality data kept in SLLC will not be evicted from cache space. The latency can be reduced as the strong locality data benefit from cache.

As we mentioned in Section 2, selective cache bypassing can be designed into computer hardware architecture. This project proposes a method at compiler level which can prevent non-temporal memory accesses from being cached in SLLC, and sequentially cache pollution is expected to be reduced.

Another challenge is that how to identify the non-temporal memory accesses in an application's source code, and how to implement the cache bypassing instructions to let them not cacheable in the execution. We need a profiler to analyse the performance for every line of code in the application and then determine the which are streaming data. Cache bypassing is achieved by inserting PrefetchNTA instructions in the assembly code, this step can be done semi-automatically by an optimizer called MAO. Those tools are presented in the following subsection.

3.3 Tools

In this project we use PrefetchNTA insertion function in MAO and HPCToolkit which can work as a profiler.

3.3.1 MAO

MAO is developed by Google Research, it is a micro-architectural optimizer for x86/64 processors. It works on assembly code and can insert a PrefetchNTA instruction in front of any memory load.

MAO achieves cache bypassing by two types of PrefetchNTA insertion options [3]:

- PREFNTA - *Simple PrefetchNTA Insertion*: This pass inserts a PrefetchNTA instruction right in front of every memory load.
- INSPREFNTA - *PrefetchNTA Insertion for Specified Instructions*: This pass inserts a PrefetchNTA instruction right before a list of specified instructions. In the input file, instructions can be specified by a list of (file name, function, offset) tuple.

```
1 for function quantum_sigma_x:
2 ......
3 174       reg->node[i]...;
4     401a30 <+80>: mov     \%rcx,\%rsi
5     401a33 <+83>: prefetchnta 0x10(\%rbx)
6     401a37 <+87>: add     0x10(\%rbx),\%rsi
7     401a42 <+98>: prefetchnta 0x8(\%rsi)
8     401a46 <+102>: xor    \%rax,0x8(\%rsi)
9 ......
```

Listing 1. Insert PrefetchNTA instruction using MAO

Code segment above shows a piece of assembler code of an application, from which we can see the PrefetchNTA instructions inserted right in front of the "add" and "xor" memory loads respectively after using MAO.

3.3.2 HPCToolkit

HPCToolkit is an open source software that includes tools for performance measurement and analysis of program at binary-level. It can record low-level events, such as cache misses and issue stall cycles. Through statistical sampling of timers and hardware performance counters, HPCToolkit works on an application's binary to collect and correlate accurate hardware performance metrics at very low overhead (1-5%).[2]

In the research [7], in order to support accurate measurement and attribution of performance metrics in out-of-order processors, instruction-based sampling (IBS) was proposed. HPCToolkits sampling based measurements rely on a performance monitoring unit (PMU) that periodically selects an instruction for monitoring while using IBS over time. The PMU records information about the occurrence of key events, latencies, and the effective address of a memory operand during the execution. When the number of samples collected is sufficiently large, their distribution is expected to approximate the true distribution of the performance metrics.[2]

While a program is running, HPCToolkits profiler triggers samples and captures full calling contexts for sample events. It also can track variables, and attribute samples to both code and variables. After executing steps *hpcrun*, *hpcstruct*, *hpcprof* consecutively with an application, a database that stores all the profiling information is generated. In order to present the data. *hpcviewer* is developed as a graphical user interface that interactively presents performance data from multiple perspectives. Figure 2 illustrates the *hpcviewer* user interface.

4. Identifying Non-Temporal Memory Accesses

4.1 Average Latency and Differential Analysis

Average latency is a metric used to analyse memory locality of data accesses in applications. It is derived from the statistical data of performance measured by HPCToolkit. For each memory access, average latency (AvgLatency) is calculated by:

$$AvgLatency = \frac{LATENCY}{CACHE\_MISSES}. \quad (1)$$

in which the LATENCY represents the total latency for a memory access in a performance profile, and the CACHE_MISSES is the sum of L3 cache misses collected by HPCToolkit.

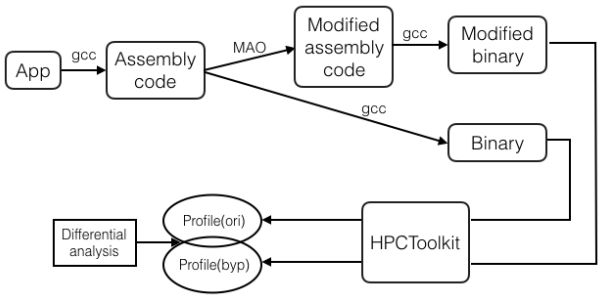

Figure 3. Differential analysis used to determine weak locality data in an application.

Therefore, we can use the average latency to analyse whether a memory access is non-temporal. Typically, if we have strong locality data bypass the last level cache, the average latency should be increased since strong locality data is frequently reused in the execution of applications. In the opposite, the cache bypassing instructions will not affect the streaming data, because weak locality memory access will not benefit from caching.

Differential analysis is a powerful technique to analyse application's performance. It is widely used in some research work before [11]. In this project the analysis is conducted through comparing a pair of profiles collected on different binary file from the same application. The work flow of differential analysis is illustrated in Figure 3. Average latency is the key metric that derived from HPCTookit databases. The comparison between AvgLatency_ori and AvgLatency_bypass is illustrated as:

$$\begin{aligned} AvgLatency\_ori &< AvgLatency\_bypass \quad (a) \\ AvgLatency\_ori &\simeq AvgLatency\_bypass \quad (b) \end{aligned} \quad (2)$$

In case (a), as for a memory access, if the average latency of the original version is less than the cache bypassing version, this memory access is determined as strong locality data. Since having strong locality data bypass cache will result in larger latency. In the opposite, as for a memory access in the application program, if the average latency does not have significant change (e.g. larger than 5%) after bypassing cache, as indicated in case (b), it can be determined as weak locality data.

Through differential analysis, weak locality memory access can be located in an application's binary. The mapping between instructions in application's assem-

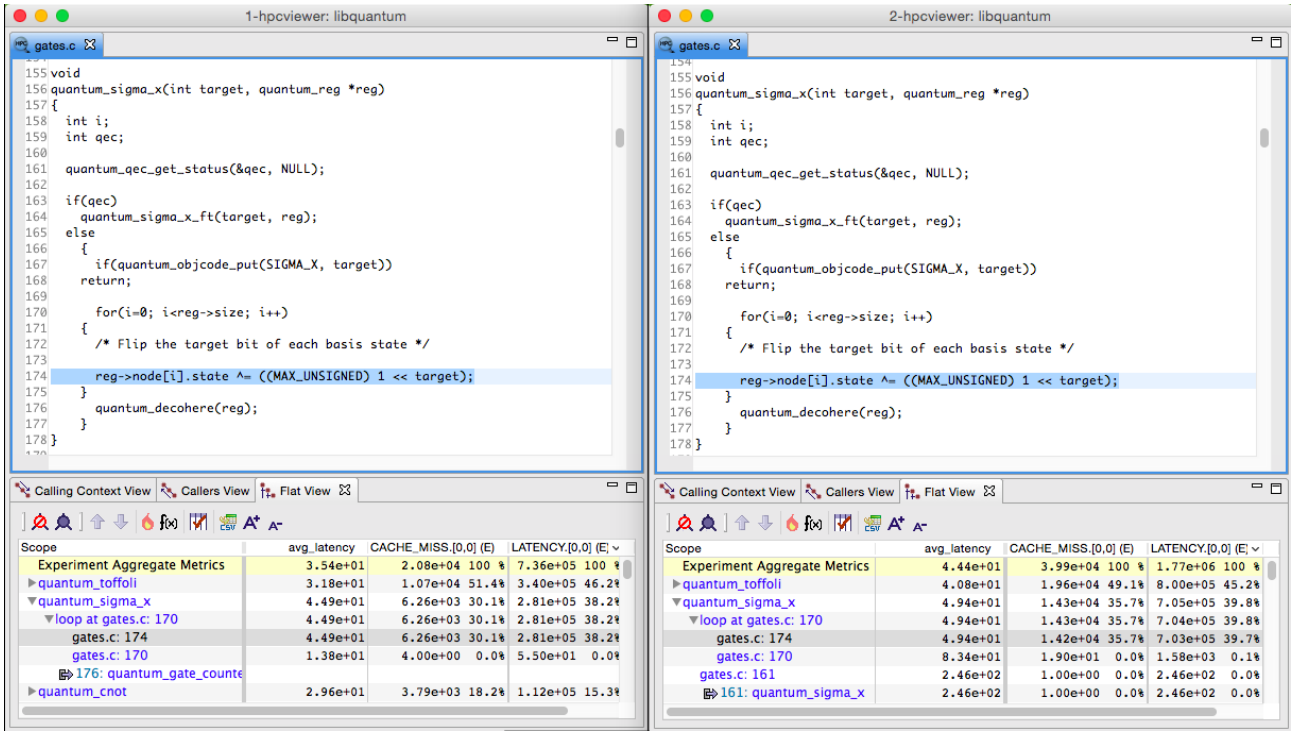

Figure 2. *libquantum* performance databases and *hpcviewer* user interface

bly code and the program should be conducted first in order to implement cache bypassing instructions.

4.2 Cache Optimization

As discussed in previous section, strong locality data should be cached in the SLLC and weak locality data need to bypass cache in order to prevent cache pollution. The INSPREFNTA option in MAO is designed for PrefetchNTA insertion for specified instructions.

As shown in the the application's profile database, the dominant memory accesses cause latency which occupy significant percentage of total latency. We only need to determine the streaming data among the dominant memory accesses. Through the mapping we can retrieve the function offset numbers which are needed as input information for INSPREFNTA function.

In this way, we implement cache bypass instructions for weak locality data accesses at application's binary level. The cache can be optimized without changing the original program or computer hardware architecture.

5. Experiments and Discussion

Experiments for this research are based upon SPEC2006 benchmarks. The experiments were conducted on a x86 hardware, 48 cores machine. In general, one processor has six cores sharing a last level cache (L3) with size 10MB. The hardware information is listed in Table 1.

LES Machine Information	
CPU Architecture	x86_64
CPU	AMD Opteron Processor 6168
Cores	48
L3 Cache size	10 MB

Table 1. Machine hardware information

Cache Management Candidates		
App	Original_time	AfterManage_time
libquantum	2m37.2s	2m37.1s
lbm	30.9s	30.4s

Table 2. Cache management application candidates.

5.1 Case Study: SPEC2006 Benchmarks

In SPEC2006, we analysed all the benchmarks and found there are two applications, *libquantum* and *lbm*, have the potential to be optimized using the technique we proposed.

Cache management application candidates are presented in Table 2.

Applications and libquantum co-run experiments				
App mix wt. libquantum	No. of instructions		No. of L3 cache misses	
	Before	After	Before	After
hmmer	3.40E+10	3.36E+10	4.65E+08	4.41E+08
bzip2	1.10E+11	1.09E+11	3.00E+09	2.61E+09
gcc	7.60E+10	7.60E+10	3.85E+09	3.33E+09
soplex	3.66E+11	3.65E+11	3.10E+10	2.80E+10
xalancbmk	3.01E+11	3.01E+11	2.40E+10	1.94E+10
astar	4.07E+11	4.07E+11	2.41E+10	2.10E+10
sjeng	4.67E+10	4.67E+10	1.11E+10	9.64E+09
gobmk	6.19E+11	6.19E+11	1.61E+10	1.37E+10

Table 3. No. of instructions and LLC misses in co- run experiments.

As shown in Table 2, the applications do not get longer running time after having the dominant memory load bypass cache. Therefore the cache space can be released and potentially used by other applications.

A series of co-run experiments were conducted and the results are presented in the Table 4 and Table 5. Take Table 4 as an example, before the cache management, we run each SPEC2006 application together with original *libquantum*. By using "-taskset" Linux command, the parallel applications can be set on two cores that share the same LLC. After the cache management for *libquantum*, we repeated the experiments and recorded the execution time for each run. We can see many applications' performance got improved, and the highest speed up was achieved up to 11.74%. Table 3 shows the number of instructions and number of LLC cache misses before and after the cache optimization. All of the applications have smaller L3 cache misses after the optimization in the co-run experiment. The results are considered as reasonable because cache pollution is reduced and therefore the applications can run faster.

Here are several more points about the co-run experiments.

- Applications presented in the table are a subset of SPEC2006 benchmarks.
- For each co-run experiment, the application and cache management application candidate (*libquantum* or *lbm*) are running on two different cores that share the same last level cache.
- Because the running time for applications are different, we let the applications continuously run several times to make sure the execution time window covers at least one entire run. The data in the table are the statistically-averaged running time.

Applications and *libquantum* co-run experiments data			
App	Before_manage	After_manage	Speed_up
hmmer	10.56s	9.32s	11.74%
bzip2	56.18s	53.77s	4.29%
gcc	1m7.13s	1m3.83s	4.92%
soplex	7m35.87s	7m15.77s	4.41%
xalan	5m52.58s	5m13.24s	11.16%
astar	7m25.27s	7m10.54s	3.31%
sjeng	4m0.57s	3m57.63s	1.22%
deall	17m25.69s	17m22.62s	0.29%
gobmk	5m18.90s	5m15.35s	1.11%

Table 4. Applications execution time before and after cache management for *libquantum*.

Applications and *lbm* co-run experiments data			
App	Before_manage	After_manage	Speed_up
hmmer	9.24s	9.29s	-0.54%
bzip2	55.40s	54.66s	1.34%
gcc	1m6.82s	1m5.23s	2.38%
soplex	7m37.01s	7m35.42s	0.35%
xalan	5m49.67s	5m35.66s	4.01%
astar	7m24.34s	7m17.72s	1.49%
sjeng	3m56.46s	3m55.45s	0.42%
deall	17m25.69s	17m22.62s	0.29%
gobmk	5m16.69s	5m15.29s	0.44%

Table 5. Applications execution time before and after cache management for *lbm*.

5.2 More Discussion

From the comparison between Table 4 and Table 5, we can see the performance improvement after optimization on *libquantum* is more obvious than *lbm*. It is because the size of streaming data determined through differential analysis in the former is more than that in latter. After the cache bypassing implementation, the former one released more cache space for the other application that is running in parallel.

In the source code in Figure 2, the blue color highlighted line is determined as non-temporal memory access. The left hand side bottom window shows the original *libquantum* performance metrics, while after implemented PREFNTA (insert PrefetchNTA in front of every memory load) on *libquantum*, we got another set

of metrics data, which are presented in the right hand side bottom window. According to the differential analysis, we can see the average latency basically remains the same for line 174 in the source code. Since this line does not benefit from cache, it is determined as the streaming data which should bypass cache. In the original code we also observed that this line is an operation of referencing "node[i]" in a loop.

6. Future Works

In this project we accomplished cache optimization for some of the applications in SPEC2006 benchmarks at the software level. Performance improved for the parallel running situation. Several potentially directions in which further research can develop in the future. The case study presented in this project is about running different applications on different cores in parallel, while the successful cache management for parallel applications (multi-threaded applications) can be a more supportive evidence for this research.

References

[1] Multi-core processor wikipage `https://en.wikipedia.org/wiki/Multi-core_processor`

[2] Xu Liu and John Mellor-Crummey, "A Data-centric Profiler for Parallel Programs" (2013) `http://hpctoolkit.org/`

[3] Robert Hundt and Easwaran Raman and Martin Thuresson and Neil Vachharajani, "MAO - an Extensible Micro-Architectural Optimizer" (2011)

[4] Qingda Lu etc. "Soft-OLP: Improving Hardware Cache Performance Through Software-Controlled Object-Level Cache Partitioning." (PACT'09)

[5] Xiaoning Ding, Kaibo Wang, and Xiaodong Zhang. "ULCC: a user-level facility for optimizing shared cache performance on multicores" (2011)

[6] Andreas Sandberg, David EkloV and Erik Hagersten, "Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses" (2010)

[7] J. Dean et al. "ProfileMe: Hardware support for instruction-level profiling on out-of-order processors." In Proc. of the 30th annual ACM/IEEE Intl.

[8] Jacob Brock, Xiaoming Gu, Bin Bao and Chen Ding, "Pacman: Program-Assisted Cache Management" (2013)

[9] Xu Liu, John Mellor-Crummey, "A tool to analyze the performance of multithreaded programs on NUMA architectures" (2014)

[10] Cache pollution wikipage. `http://en.wikipedia.org/wiki/Cache_pollution`

[11] Cristian Coarfa, John Mellor-Crummey, Nathan Froyd, and Yuri Dotsenko. "Scalability analysis of SPMD codes using expectations." In Proceedings of the 21st Annual international Conference on Supercomputing (Seattle, Washington, June 17 - 21, 2007). ICS '07. ACM Press, New York, NY, 13-22

A. Appendix: Scripting Code

A.1 Create Input Files for INSPREFNTA Instructions

Python script to generate input files automatically for INSPREFNTA of MAO. To specify offset numbers in the list with "offsets", and function name with the string "fun_name".

```
1
2  #!/usr/bin/python
3
4  import sys
5
6  def gene(f,offsets,fun_name):
7    for offset in offsets:
8      text = 'f'+'\t'+fun_name+\
9      '+'+str(offset)+'\n'
10     f.write(text)
11
12 def main():
13
14   f = open('INSP.txt', "w")
15
16   offsets = []
17   fun_name = ''
18   gene(f,offsets,fun_name)
19
20   f.close()
21
22 if __name__ == '__main__':
23   main()
```

Listing 2. Insert PrefetchNTA instruction using MAO

A.2 Shell Script to Compile files with MAO

The script to compile files with PREFNTA of MAO is listed below:

```
1  #!/bin/bash
2  for i in *.c
3  do
4          gcc -g -S "$i"
5  done
6
7  for j in *.s
8  do
9  user_path/MAO/bin/mao-x86_64-linux\
10 --mao=--plugin=user_path/MAO/bin/\
11 MaoPrefetchNta-x86_64-linux.so \
12 --mao=PREFNTA=ptype[0] \
13 --mao=ASM=o["$j".new.s] "$j"
14 done
```

Listing 3. Insert PrefetchNTA instruction using MAO

The script to compile files with INSPREFNTA of MAO is listed below:

```
1  #!/bin/bash
2  for i in *.c
3  do
4          gcc -DSPEC_CPU -DNDEBUG
5          -O2 -fno-strict-aliasing
6          -DSPEC_CPU_LP64
7          -DSPEC_CPU_LINUX
8          -g -S "$i"
9  done
10
11 for j in *.s
12 do
13 user_path/MAO/bin/mao-x86_64-linux\
14 --mao=--plugin=user_path/MAO/bin/\
15 MaoInsertPrefNta-x86_64-linux.so \
16 --mao=INSPREFNTA=
17 instn_list["INSP.txt"]\
18 --mao=ASM=o["$j".new.s] "$j"
19 done
```

Listing 4. Insert PrefetchNTA instruction using MAO