Exploiting the Role of Hardware Prefetchers in Multicore Processors

Hasina Khatoon
Computer & Info. Sys. Engg. Dept.
NED Univ. of Engg. & Technology
Karachi, Pakistan

Shahid Hafeez Mirza
Usman Institute of Engg. & Tech.
Karachi, Pakistan

Talat Altaf
Electrical Engg. Dept.
NED Univ. of Engg. & Tech.
Karachi, Pakistan

***Abstract*—The processor-memory speed gap referred to as *memory wall,* has become much wider in multi core processors due to a number of cores sharing the processor-memory interface. In addition to other cache optimization techniques, the mechanism of prefetching instructions and data has been used effectively to close the processor-memory speed gap and lower the memory wall. A number of issues have emerged when prefetching is used aggressively in multicore processors. The results presented in this paper are an indicator of the problems that need to be taken into consideration while using prefetching as a default technique. This paper also quantifies the amount of degradation that applications face with the aggressive use of prefetching. Another aspect that is investigated is the performance of multicore processors using a multiprogram workload as compared to a single program workload while varying the configuration of the built-in hardware prefetchers. Parallel workloads are also investigated to estimate the speedup and the effect of hardware prefetchers.**

This paper is the outcome of work that forms a part of the PhD research project currently in progress at NED University of Engineering and Technology, Karachi.

Keywords—Multicore; prefetchers; prefetch-sensitive; memory wall; aggressive prefetching; multiprogram workload; parallel workload.

I. INTRODUCTION

Multicore processors are the mainstream processors of today with the number of cores increasing at a fast pace. A number of issues have emerged in these processors that are becoming more acute with the increasing number of cores. A large body of publications has accumulated in the last decade that has summarized these issues. Some of the challenges are presented in [1]. One of the main issues that directly impacts application performance is the large processor-memory speed gap referred to as memory wall by Wulf and Mckee [2] and elaborated by Weidendorfer [3]. Recent researches have sought solution to this problem through on-chip cache hierarchy [4] and novel architectural features like NUCA cache [5]. Other solutions include R-NUCA [6], Victim Replication [7], and Pressure-Aware Associative Block Placement [8]. A detailed summary of the publications related to the memory wall problem is presented in [9].

One of common solution to the memory wall problem is prefetching of instructions and data at every level of memory hierarchy. Prefetching is a latency hiding technique that access instructions and data from the next level of memory hierarchy before the demand for it is actually raised by the processor. Prefetching was almost always beneficial in single core processors, even though there were some useless prefetches. As a result, prefetchers now form an integral part of most of the current generation processors. In multicore processors, all cores share chip resources that include on-chip memory hierarchy and the processor-memory interface. If all cores generate prefetch requests in addition to demand requests, a large amount of interference takes place causing contention for resources. This prefetcher caused contention may result in performance degradation in multicore processors, especially if prefetchers are used aggressively. Therefore, there is a need to investigate the effectiveness of prefetchers in multicore processors under different conditions and for all types of applications. The contribution of this paper is the analysis and quantification of the behaviour of applications in the presence and absence of prefetchers. The derived results provide guidelines for applications to activate prefetchers only when they are useful.

Recent research has focused on improving data prefetching mechanisms, especially for big data analysis and other streaming applications. Though prefetching pose degradation problems in multicore processors, especially when used aggressively, they remain the most effective mechanism to avoid stalls that are caused due to long latency accesses and contention based delays. This necessitates enhancements in the prefetcher designs that adapt to congestion and dynamically adjust their aggressiveness. Chen et al. in their publications [10, 11] have proposed storage efficient data prefetching mechanisms and power efficient feedback controlled adaptive prefetchers that are accurate and efficient. Other recent enhancements are discussed in Section II.

The rest of the paper is organized as follows. Section II gives a brief overview of related work. Section III outlines the experimental setup including test programs and specifications of the experimental platforms. Section IV presents the results and a brief analysis of the results and Section V concludes the paper.

II. RELATED WORK

Since prefetching is considered to be an important latency hiding technique, it has been used effectively in both single core processors and single core multiprocessors. Prefetching is performed in hardware, in software or in both. Software prefetching is supported by prefetch instructions and requires effort by the programmer or the compiler writer. Nataranjan et al. [12] have explored the effectiveness of compiler directed

prefetching for large data accesses in in-order multicore systems. Since the focus of this work is hardware prefetching, this section shall briefly describe some of the recent publications related to hardware prefetchers in the context of multicore processors.

Prefetchers are beneficial due to the principle of locality, an attribute of software. This is true most of the time in single core architecture, but as pointed out in [13], aggressive prefetching in multicore processors result in a large amount of interference giving rise to performance degradation. Ebrahimi et al. [13] have proposed more accurate prefetching with the use of local as well as global feedback by Hierarchical Prefetch Access Control (HPAC) to partially alleviate the above problem. Using coordinated prefetching, the authors compare the results of aggressive prefetching in multicore processors with that of a single core processor. With dynamic control of the prefetch aggressiveness using feedback directed control, they have shown that their technique improves the system performance by 14%.

Lee et al. [14] have identified degradation in performance due to congestion in the interconnection network especially due to prefetch requests in multicore processors. They have proposed to prioritize demand traffic by using TPA (Traffic-Aware Prioritized Arbiter) and TPT (Traffic-Aware Prefetch Throttling) to counter the negative effects of prefetch requests. Fukumoto et al. [15] have proposed the use of cache-to-cache transfer to reduce the overall congestion on the memory-bus interface.

Kamruzzaman et al. [16] have proposed a different way of using prefetching especially for applications like the legacy software that are inherently sequential in nature and cannot use all cores of the CMP. They have suggested the use of prefetch threads as *helper threads* to run on unused cores and make use of the injected parallelism for prefetching code and data. Using thread migration techniques, an overall improvement of 31% to 63% is shown for legacy software. The authors have concluded in their final analysis that the technique can also be used to enhance the performance of applications that are parallel.

Wu et al. [17] have proposed an automatic prefetch manager that estimates the interference caused by prefetching and adjusts the aggressiveness while programs are running. They have shown that this dynamic management improves the application performance and makes it more predictable. Verma et al. [18] have evaluated the effectiveness of various hybrid schemes of prefetching and have proposed to adaptively reduce the number of prefetches to reduce the interference. Lee et al. [19] identify the lack of parallelism that exists in DRAM banks, especially in multicore processor-based systems. They have proposed mechanisms to maximize DRAM Bank Level Parallelism (BLP) using BLP-aware Prefetch Issue (BAPI) with BLP-Preserving Multi core Request Issue (BPMRI) that helps improve the application performance with parallel servicing of requests. Ebrahimi et al. [20] have proposed mechanisms to exploit prefetching for shared resource management in multicore systems.

Nachiappan et al. [21] have suggested prefetch prioritization in the interconnection network on the basis of the potential utility of the requests in order to reduce the negative effects of prefetching. Wu et al. [22] characterize the performance of the LLC (Last Level Cache) management policies in the presence and absence of hardware prefetching. They propose Prefetch-Aware Cache Management (PACMan) for better and predictable performance. Lee et al. [23] have proposed prefetch-aware on-chip networks and network-aware prefetch designs that is sensitive to network congestion. Manikantan and Govindarajan [24] have proposed performance-oriented prefetching enhancements that include *focused prefetching* to avoid commit stalls. The authors claim that this enhancement also improved the accuracy of prefetching.

A number of recent publications have proposed complex prefetching mechanisms that take into account various factors while prefetching code and data [18, 25]. Grannaes et al. [25] have proposed Delta Correlating Prediction Table (DCPT), a pefetching heuristics based on the table-based design of Reference Prediction Tables (RPT) and the delta correlating design of Program Counter/ Delta Correlating Prefetching (PC/DC) with some improvements. These complex prefetching techniques have overheads that cannot be ignored as prefetchers incur a significant burden on system resources. Since simple prefetchers have low overheads, they are used mostly in current generation processors. For example, the prefetchers used in our experimental platform are simpler [26] as compared to the prefetchers discussed in [18, 25].

III. The Experimental Setup

This section gives an account of the test programs, the experimental platforms and the hardware prefetchers present in these experimental platforms.

Although prefetching code and data have been significantly effective in single core processors, some of the recent publications have pointed out an anomaly that takes place when prefetchers are used in multicore processors. Use of aggressive prefetching cause interference and results in overall degradation of performance [13], demanding an adjustment in the prefetch strategy. In many instances, it has been observed that applications perform better without all the prefetchers used by default as these are built-in in all current generation processors. The designers of most of these processors have therefore provided mechanisms where applications may use prefetch manipulating techniques to selectively enable/ disable the built-in hardware prefetchers, whenever desired. This involves manipulation of Machine Specific Registers (MSRs) related to hardware prefetchers. The decision to enable/ disable pefetchers is left to the application designer. The application areas that benefit most due to cache locality and being prefetch sensitive may continue using the prefetchers, but these applications should also investigate the benefits, before using it by default.

A. Test Programs

Three types of benchmark programs are used to measure and evaluate performance with enabled/ disabled configurations of prefetchers: SPEC 2006 [27], the parallel Parsec Benchmark suite [28] and the concurrent matrix multiplication program [29]. SPEC 2006 is a commonly used

benchmark for evaluation of single and multiprogram workloads; the Parsec Benchmark suite is used to evaluate the performance for parallel workloads and the concurrent matrix multiplication program is used to evaluate the performance of concurrent workloads. A brief description about the three sets of benchmarks/ programs is given in the following subsections.

1) SPEC 2006 Workload

The first set of experiments are conducted to evaluate the effect on the performance of SPEC 2006 [27] benchmark programs when run on multicore processor with various configurations of the built-in hardware prefetchers. Since SPEC 2006 benchmarks are not inherently parallel, multiple copies of each benchmark are run in parallel as a multiprogram workload to keep all cores busy and observe the results with and without prefetchers. Most of the runs are *reportable* runs [27] and the results of *reference* input set is used to perform the analysis. The experiments performed on the example platform took between 9 to 19 hours each and in some cases, it was even higher. In case of multiprogram workloads, the time taken was up to 31 hours for complete execution.

2) The Parallel Benchmarks Suite

The Princeton Application Repository for Shared Memory Computers (PARSEC) is a parallel benchmark suite that is suitable to evaluate multicore processors [28]. It consists of 13 benchmark programs taken from several different application domains including financial analysis, *animation*, data mining, computer vision, etc. These are diverse and emerging multithreaded workloads focusing on desktop and server applications that are expected to be the eventual workloads for multicore processors. The number of threads of each program can be adjusted depending on the number of cores and the application requirements. A detailed description of the design and implementation of this benchmark suite is given in [30]. However, a brief description is given below.

Both current and emerging workloads from recognition, mining and synthesis (RMS) application areas are represented in this benchmark suite. Each of the applications has been parallelized fulfilling the requirements of multithreaded applications that can be run on parallel architectures like multicore processors. Using parallelization models of Pthreads, OpenMP and Intel TBB, these programs provide portability for various types of platforms. Some of the programs present in the suite are *dedup, blackscholes, facesim, fuidanimate,* etc., taken from the application areas of computer vision, data mining, visualization, media processing, animation, financial analysis, etc. Six different input sets with different properties are defined for each workload that can be used with variable number of threads. Out of the input sets of *test, simdev, simsmall, simmedium, simlarge* and *native,* the *native* input set is the largest and is closest to the actual inputs.

All 13 benchmark programs are run with the *native* input set using single thread and *n* threads, where *n* is chosen to be the number of cores for each of the experiments. A more detailed description about the use of this benchmark suite can be found in [31].

3) Matrix multiplication program

The third test program is the parallel matrix multiplication program. This program has been parallelized to run on multicore processors using SPC^{3}PM (Serial, Parallel and Concurrent Core to Core Programming Model [29]), an algorithm developed at NED University by Ismail et al. for parallelization of programs. This programming model allows the user to specify any number of cores depending on the amount of parallelism and the available resources. More details about the model and algorithm can be found in [29].

B. The Experimental Platforms

Most of the experiments were conducted on a platform based on the 4-core Intel Core2 Quad processor running OpenSUSE 11.1 Linux 2.6.27.7 operating system. The main features of this machine are summarized in Table I as Platform No. 2. Both integer and floating point programs of SPECCPU2006 benchmark suite [27] and the Parsec Benchmark suite [28] were run on this platform using various combinations of the four built-in prefetchers per core in the multicore processor. A detailed description of the four prefetchers per core and a description of the Model Specific Register (MSR) to control them are given in the Intel Software Developers Manual [26].

Some experiments were also conducted on a 2-core and an 8-core machine to examine and validate some of the results obtained from the main platform. The salient features of these platforms are also listed in Table 1 as platform Nos. 1 and 3 respectively. The results of experiments conducted on platforms 1 and 3 are used as additional data for validation and testing of results and only a summary of the results are presented. The platform chosen to run the third test program is the dual-core Intel Xeon processor X5670 Series based SR1600UR server system having 24 cores. Other salient features of this platform are also listed in Table I as the specifications of platform No. 4.

C. Prefetchers in the main Experimental Platform

The example platforms 1 to 3 that are used to conduct most of the experiments in this study have four prefetchers per core, each performing the function of prefetching a specific set of information [26]. A brief description of the four hardware prefetchers in the experimental platforms is given in the following paragraph.

- The Instruction Prefetcher (IP), referred to as pf4 in this paper, prefetches instructions in the L1 instruction-cache based on branch prediction results.
- The Adjacent Cache Line (ACL) prefetcher, referred to as pf2, prefetches the next matching block in a cache block pair in to L2 cache.
- The Data Cache Unit (DCU) prefetcher, referred to as pf3, observes and detects the number of accesses to a specific cache block for a predetermined period of time and prefetches the subsequent block in the L1 D-cache.
- The Data Prefetch Logic (DPL) prefetcher, referred to as pf1, functions similar to the DCU prefetcher, except that the blocks are prefetched in L2 cache after it detects accesses to two successive cache blocks.

TABLE I. SPECIFICATIONS OF EXPERIMENTAL PLATFORMS

	Platform 1	Platform 2 (Main Platform)	Platform 3	Platform 4
Processor	Intel Core™ 2 Duo CPU @ 2.2 GHz	Intel Core™ 2 Quad CPU @ 2.66 GHz	Intel Core™ i7-2600 CPU @ 3.4 GHz	4 x Intel Xeon X5670@ 2.93CHz
No .of cores	2	4	8	4 x 6
Cache and System Parameters				
L1 D-Cache (per core)	32KB, 64B, 8-way associative	32KB, 64B, 8-way associative	32KB, 64B, 8-way associative	6x32KB
L1 I-Cache (per core)	32Kb, 64B, 8-way associative	32KB, 64B, 8-way associative	32KB, 64B, 8-way associative	6 x 32KB
L2 Cache	2MB, 64B, 8-way associative	4MB 64B, 16-way associative	4x256KB, 64B, 8-way assoc.	6 x 256 KB
L3 Cache	NA	NA	8MB, 64B, 16-way assoc.	12 MB
Main mem.	1GB	4GB	8GB	24 GB
Operating System	OpenSUSE 11.1 Linux Kernel 2.6.27.7	OpenSUSE 11.1 Linux Kernel 2.6.27.7	OpenSUSE 11.1 Linux Kernel 2.6.27.7	Windows 2008 Server (64-bit)

Each of the four prefetchers can be selectively enabled/ disabled by putting On/Off individual bits in the Model Specific Register (MSR) number *0x1A0h* present in each core. This register can be accessed and the corresponding bits can be manipulated using assembly-level instructions. The tool used to manipulate the register bits for these experiments is *msr t*ools [32] available as free software. In addition to hardware prefetchers, prefetch instructions are also provided in all current generation processors that can be used to program prefetching of data through software prefetching. It may be noted that the experimental platform No. 4 does not allow selective enabling/ disabling of its hardware prefetchers. It only allows *all* prefetchers to be either enabled or disabled.

The following paragraphs summarize the results of the experiments performed after selective enabling/disabling of prefetchers and the effect it has on the performance of multicore processors.

IV. Results And Analysis

Table II gives a summary of the experiments conducted to deduce the following results on the main platform (platform 2). A number of experiments that were conducted on platforms 1, 3 and 4 are also discussed in this section (not given in Table II).

A. Benchmarks and Measurement Metrics

The experiments were conducted by running all 29 SPECCPU2006 programs comprising of 12 integer benchmarks and 17 floating-point benchmarks, all 13 programs of Parsec Benchmark suite Version 2.1 and the concurrent matrix multiplication program. The effect of the use of prefetch inhibiting techniques on the overall performance of benchmark programs is illustrated through column charts. In addition, collected data is also presented in the form of tables that give more accurate information. Two separate sections present the results of SPEC2006 benchmarks as single program and multiprogram workloads. A third section presents the results of parallel benchmarks.

Some of the terms that shall be used to explain the results in this paper have been taken from [13] and are discussed here briefly. An application is said to have *cache locality* if the number of L2 cache hits per 1000 instructions is greater than five. If the L2 cache miss is greater than 1 per 1000 instructions (MPKI – Miss Per Kilo Instructions), the application is referred to as *memory intensive*. If the improvement in performance when a prefetcher is used is greater than 10% compared to no prefetching, the application is said to be *prefetch sensitive*.

B. SPEC CPU 2006 Results

Results of experiments 1 to 16 that were conducted with various prefetcher options are presented in this section.

1) SPEC CPU 2006 as Single Program Workload

Experiments 1 to 4 and 9 to 12 were conducted by using all SPEC programs as single program workload with various prefetcher options.

2) SPECint as Single Program Workload

In experiments 1 to 4, use of prefetchers mostly proved to be beneficial, because all resources were utilized by only a single core as SPEC benchmarks are not inherently parallel. Fig. 1 shows the performance in terms of execution time for SPECint 2006 benchmarks executed with and without the built-in hardware prefetchers in each of the cores. A number of experiments were conducted using various configurations of on-chip hardware prefetchers. Four of these experiments are listed in Table II. The data collected from the experiments is presented in Table III. An overall average degradation of 14.4% is observed in 10 out of 12 integer benchmarks when the DPL (pf1) prefetcher is disabled. This is because prefetching in L2 is more beneficial for most of the applications. The highest degradation of 54% is observed in *libquantum* benchmark. Since this benchmark consists of a library of software that simulates a quantum computer, it is expected to be prefetch sensitive and benefits most from any kind of prefetching mechanism. The other benchmark programs that are prefetch sensitive are *mcf*, *sjeng* and *xalancbmk*. Two of the benchmarks, namely, *hmmer* and *omnetpp* give better performance when the prefetcher is off, with *hmmer* giving an improvement of 7.3%. This is because *hmmer* is database search software that searches for a gene sequence.

The experiments were again conducted by disabling two of the four prefetchers and a different set of results were obtained (experiment 3). When both DPL (pf1) and ACL (pf2) prefetchers are disabled, there is an average degradation of 13.5% in only 3 out of 12 integer benchmarks and 9 benchmark programs show an average improvement of 8.4%.

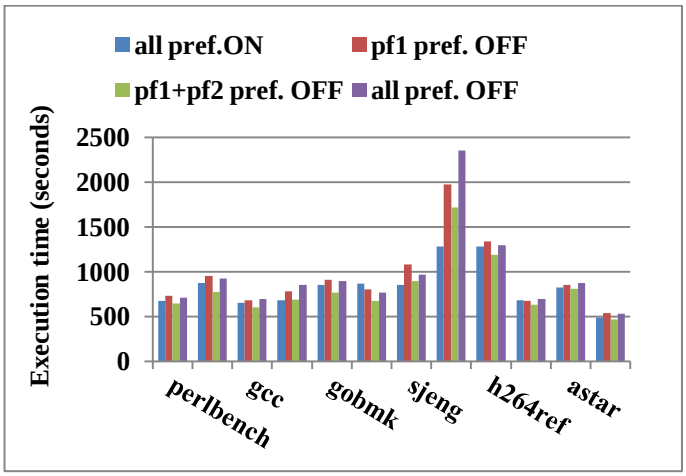

Fig. 1 Execution time of single copy of SPECint 2006 Benchmark programs with various prefetcher configurations

The highest degradation of 34.6% is observed in *libquantum* program because of the reasons mentioned before. The *hmmer* program shows the highest improvement of 22.6%.

When all four prefetchers are disabled (experiment 4), the benchmarks show degradation in almost all SPECint programs with an average degradation of about 14%, with *libquantum* program suffering from the highest degradation of 84%. The reason for this behaviour has already been mentioned before. Only one of the programs namely, *hmmer* shows an improvement of 11% when all the four prefetchers are off, because it does not benefit from prefetching.

The results show that prefetchers are beneficial for single SPECint programs in multicore processors. Most of the applications benefit from prefetchers because the interference between demand and prefetch requests is not significant as only a single core generates prefetch and demand requests.

a) SPECfp as Single Program Workload

The results of SPECfp benchmarks are shown in Fig. 2 (experiments 9 to 12). Compared to the integer benchmarks, only 6 out of 17 floating-point benchmarks suffer from an average degradation of 6.1% in performance when the DPL (pf1) prefetcher is disabled. There is only one SPECfp benchmark that is prefetch sensitive, namely *bwaves* which suffers from the highest degradation of 19.8%. *bwaves* is a computational fluid dynamics software that simulates blast waves in three dimensions. Such software tends to benefit from prefetching. An average improvement of 6.4% is observed in 10 out of 17 floating point benchmarks with as high as 13.9% improvement observed in *povray* benchmark program. This is an image rendering software that uses ray tracing to visualize an object.

TABLE II. LIST OF EXPERIMENTS WITH DIFFERENT PREFETCHER OPTIONS ON PLATFORM 2 WITH EXECUTION TIME

Benchmark	Execution Mode	Experiment No.	Prefetcher option	Execution Time in seconds
SPECint	Single Program Workload	1	All Enabled	32400
		2	DPL=Disabled	35520
		3	DPL+ACL= Disabled	31200
		4	All Disabled	35280
	Multi-program workload (4-copies)	5	All Enabled	44400
		6	DPL= Disabled	43860
		7	DPL+ACL = Disabled	47040
		8	All Disabled	46380
SPECfp	Single Program Workload	9	All Enabled	72120
		10	DPL=Disabled	70140
		11	DPL+ACL = Disabled	69960
		12	All Disabled	70680
	Multi-program Workload (4-copies)	13	All Enabled	104400
		14	DPL=Disabled	104520
		15	DPL+ACL= Disabled	106320
		16	All Disabled	112440
PARSEC Benchmarks	Single Thread Workload	17	All Enabled	6840
		18	DPL=Disabled	6780
		19	ACL =Disabled	7020
		20	IP =Disabled	6600
		21	All Disabled	7560
	Multiple Thread Workload (4 threads)	22	All Enabled	3480
		23	DPL=Disabled	3720
		24	ACL=Disabled	3960
		25	IP=Disabled	3360
		26	All Disabled	3720

The ray tracing programs do not benefit from prefetching. As a result of experiment 11, the behaviour of floating point benchmarks remains almost the same as with one prefetcher disabled, with only a small change that can be observed from Fig. 2.

When all the four prefetchers are disabled, 8 out of 17 benchmark programs suffer from an average degradation of 17% with the highest degradation of 31% seen in *GemsFDTD* program. This program benefits from prefetching because it is a computational electromagnetic application that comprises mostly of loops. All the above results indicate that there is anomaly even when a single copy of benchmarks is run and different applications show different behaviour with or without the use of prefetchers. Moreover, floating point benchmarks mostly perform better when prefetchers are disabled selectively as compared to integer benchmarks. A closer examination reveals that most of the SPECfp programs are not prefetch sensitive.

TABLE III. EXECUTION TIME OF SPEC2006 PROGRAMS AS SINGLE PROGRAM WORKLOADS

Benchmarks	Execution Time (in seconds) of Benchmark programs with selective enable/ disable of on-chip Prefetchers			
	All PFs enabled	pf1 disabled	pf1+pf2 disabed	All PFs disabled
Perlbench	670	728	644	712
bzip2	874	951	776	923
Gcc	652	684	601	692
mcf	679	778	686	849
gobmk	852	907	767	896
hmmer	866	803	670	768
sjeng	855	1085	895	968
libquantum	1279	1975	1722	2357
h264ref	1285	1342	1188	1296
omnetpp	682	673	633	692
astar	822	854	810	877
xalancbmk	486	540	469	531
bwaves	1124	1347	1283	1334
gamess	1995	1845	1751	1448
milc	899	907	904	1060
zeusmp	1119	1076	1073	1150
gromacs	1297	1116	1192	900
cactusADM	2263	2185	2199	2016
leslie3d	2046	2190	2165	2348
namd	914	914	915	778
dealII	752	730	728	686
soplex	704	756	811	920
povray	508	437	430	317
calculix	1938	1808	1894	1759
GemsFDTD	1768	1770	1822	2319
tonto	1274	1183	1199	967
lbm	1204	1170	1140	1197
wrf	1520	1536	1586	1763
sphinx3	1720	1685	1576	1802

3) SPEC CPU 2006 as Multiprogram Workload

In this section, we present the results of experiments 5 to 8 and 13 to 16, which were performed by running SPEC programs as multiprogram workload to keep all cores busy. Although multicore processors are more useful and powerful for parallel workloads, most of the software that runs on these processors today is not parallel. For all such software, the main advantage that can be gained with multicore processors is higher throughput. Hence, studying the behaviour of multicore processors for multiprogrammed workload is also as important as for parallel workloads.

It was observed during these experiments, that there is a large increase in execution time with multiprogram workload. This is because a large number of memory requests, including demand and prefetch requests share the same limited bandwidth of the processor memory interface. The amount of memory required to run the programs also increases. In case of *mcf*, one of the integer benchmarks, the program becomes very slow and its progress almost stops on our experimental platforms because of the heavy usage of memory. This program is therefore not included in these measurements. For all other programs, the interference caused by multiple requests result in an overall degradation in performance as compared to the single program run on multiple cores. The execution time on the 4-core machine increases by an average of 50% for all integer benchmarks with the highest increase of 200% in the *libquantum* benchmark program. In case of floating-point benchmarks, multiprogram workload increases the execution time by almost 74% over the single program execution time, with the highest increase of 206% in the *lbm* benchmark program. In such a scenario it would not be fair to compare the performance of benchmarks when a single program is run with the performance of multiprogram workload with and without hardware prefetchers. The comparison is therefore made when a single program is run with and without prefetchers and when multiprogram workload is run with and without the hardware prefetchers.

The performance further degrades when multiprogram workload executes on an 8-core machine. The 8-core machine is an i7-based computer and the architecture of cores is similar to that of our main experimental platform. The gap between the execution time of single program and multiprogram is much wider than that of the 4-core machine. The average degradation in performance for 8-copies of integer benchmarks as compared to a single program run on the same machine is 150%, with *libquantum* suffering from the highest degradation of 490%. Fig. 3 shows the comparison of execution time of each SPECint benchmark program. When floating point benchmarks are run as multiprogram workload,

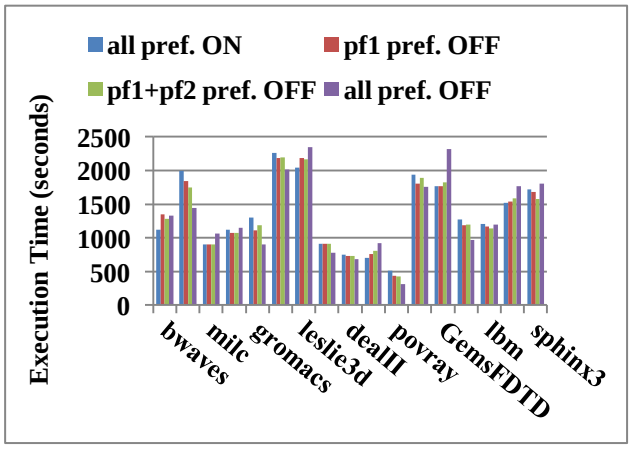

Fig. 2 Execution time of single SPECfp 2006 Benchmark programs with different prefetcher configurations

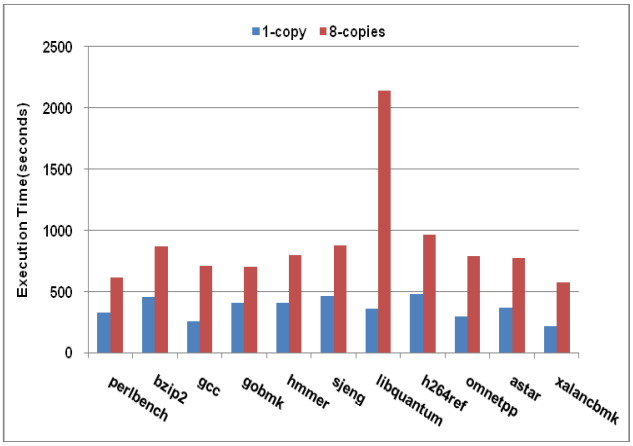

Fig. 3 Comparison of execution time of SPECint programs as single and multiprogram workloads on an 8–core machine (Platform No, 3)

the average degradation as compared to a single program is 173% with *lbm* suffering from the highest degradation of above 700%. The contention for resources is much higher in an 8-core machine because of a higher interference/ conflict between demand and prefetch requests giving rise to higher execution times. Similar results are obtained on a 2-core machine where two copies of benchmarks take much longer to execute as compared to a single program on the same machine.

b) SPECint as Multiprogram Workload

Keeping in view the objectives mentioned in the first part of section V.B.2 to compare multiprogram workloads separately, Fig. 4 summarizes the performance measurements of the benchmarks as a result of experiments 5 to 8. Table IV presents the data that were collected from these experiments. The increase in execution time is attributed to a number of factors including the interference that takes place between the demand and prefetch requests generated by all cores. The observations from Fig. 4 are summarized in the following paragraph.

Five out of 11 integer benchmarks suffer from an average degradation of 1.2% if the DPL (pf1) prefetcher is disabled (experiment 6), with the highest degradation of 2.2% observed in *h264ref* benchmark. This is video compression software that encodes video streams using two different parameter sets. 6 out of 11 integer benchmarks show an average improvement of 4.4% with the highest improvement of 12% observed in *omnetpp* benchmark. The *omnetpp* benchmark performs discrete event simulation by modelling a large Ethernet network on a campus. When both DPL (pf1) and ACL (pf2) prefetchers are disabled (experiment 7), 9 out of 11 integer benchmarks suffer from an average degradation of 10.7% with *sjeng* suffering from the highest degradation of 16.9%. Almost 25% of integer benchmarks perform better with an average improvement of 6.5%. When all the prefetchers are disabled, 9 out of 11 benchmark programs degrade in performance with an average degradation of 8% and the highest degradation of 15.2% is observed in *bzip2* program. *bzip2* is a compression software that benefits from pefetching. Two of the integer benchmarks improve in performance with an average improvement of 10.4%.

With multiprogram workload, disabling DPL prefetcher gives a better performance for most of SPECint benchmarks. This prefetcher belongs to L2 cache, which is interfaced to main memory.

a) SPECfp as Multiprogram Workload

The result of SPECfp programs (experiments 13 to 16) is illustrated in Fig. 5 with the data presented in Table IV. When the DPL(pf1) prefetcher is disabled, the SPECfp benchmarks show an average degradation of 3.3% in 10 out of 17 benchmarks with the highest degradation of 9% in *leslie3d* benchmark. This is a computational fluid dynamics program consisting of a large number of loops that benefit from prefetching. There is an average improvement of 3.3% in 7 out of 17 floating point benchmarks with the highest improvement of 16% in *milc* benchmark.

Almost 71% SPECfp benchmarks suffer from an average degradation of 4.7% when both DPL(pf1) and ACL(pf2) prefetchers are disabled (experiment 15), with the highest degradation of 10.5% in *leslie3d* benchmark. On the other hand, 5 out of 17 benchmark programs show an average improvement of 4.8% with the highest improvement of 17.2% observed in *milc* program. SPECfp gives the best performance improvement when the ACL(pf2) prefetcher is disabled with an average improvement of 8.2% in all programs. The highest improvement of 14.3% takes place in *povray* program, which is a computer visualization program that renders images through ray tracing.

TABLE IV. EXECUTION TIME OF SPEC2006 BENCHMARKS AS MULTIPROGRAM WORKLOADS

Benchmarks	Execution Time (in seconds) of 4-copies of Benchmark programs with selective enable/ disable of on-chip Prefetchers			
	All PFs enabled	pf1 disabled	pf1+pf2 disabled	All PFs disabled
perlbench	817	799	936	882
bzip2	1131	1144	1315	1303
gcc	948	908	953	977
gobmk	1023	1018	1159	1109
hmmer	950	957	1067	1037
sjeng	1139	1141	1331	1244
libquantum	3771	3837	4002	4003
h264ref	1566	1601	1758	1687
omnetpp	1247	1098	1094	1037
astar	1269	1212	1261	1220
xalancbmk	794	768	825	835
bwaves	2129	2244	2276	2370
gamess	2341	2359	2427	2161
milc	2304	1930	1907	1819
zeusmp	1416	1445	1516	1498
gromacs	1430	1437	1474	1303
cactusADM	2660	2658	2705	2542
leslie3d	2698	2939	2980	3654
namd	1131	1121	1137	1053
dealII	915	951	967	1011
soplex	1536	1487	1487	1631
povray	559	564	568	513
calculix	2232	2227	2221	2184
GemsFDTD	3042	2984	2971	3301
tonto	1584	1618	1633	1645
lbm	3678	3650	3653	3646
wrf	2001	2124	2120	2650
sphinx3	2713	2778	2884	3827

C. Parallel Benchmarks Results and Analysis

Three sets of experiments were conducted using the parallel benchmarks of 'Parsec Benchmark suite'. The first and second set was run on platform number 2 and the third set of experiment was run on platform 3. The results are presented in the following paragraphs.

core

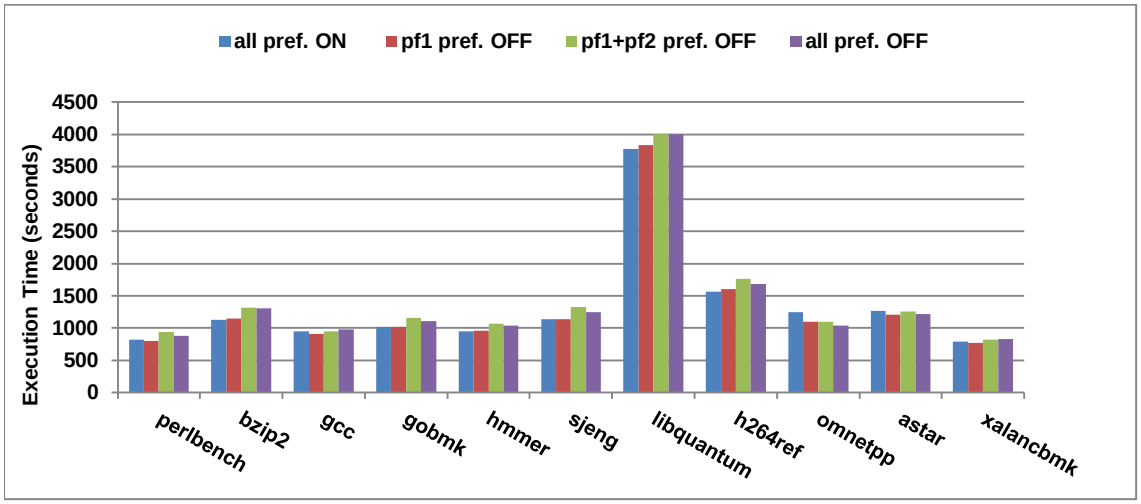

Fig. 4 Execution time of SPECint 2006 Benchmarks as multiprogram workloads (4-copies) with different prefetcher configurations

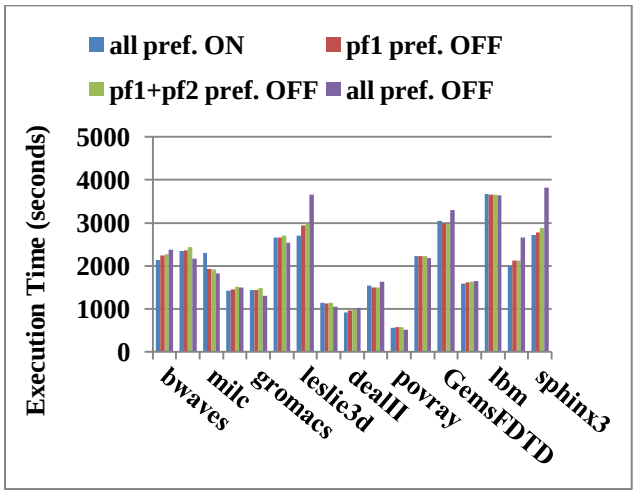

Fig. 5 Execution time of SPECfp 2006 Benchmarks as multiprogram workload (4-copies) with different prefetcher configurations

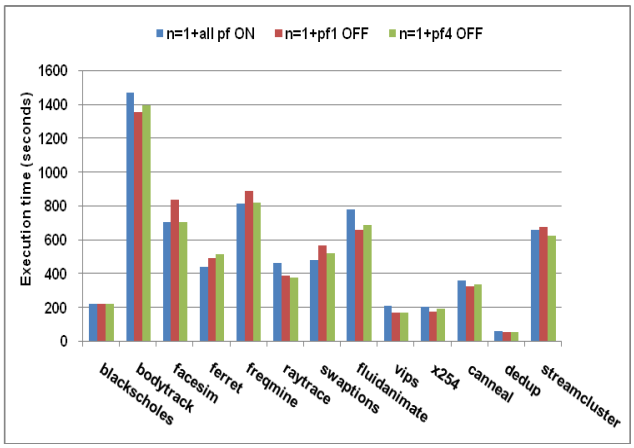

Fig. 6 Execution time of PARSEC Benchmarks with single thread and different prefetcher configurations

1) Parsec Benchmarks with Single Thread

Experiments 17 to 21 were performed with single thread and various configurations of hardware prefetchers. In experiment 18, 7 out of 13 benchmarks perform 9.5% better on the average, with the *vips* benchmark giving the highest improvement of 19.6%. The best results are obtained when the IP prefetcher is disabled (experiment 20), giving an average performance improvement of 5.3% in 9 out of 13 programs with the highest improvement of 20.6% in *vips* benchmark program. This is a media processing application that applies a series of transformations to an image. Other benchmarks that perform better with IP disabled are mostly image processing related applications. Fig. 6 gives the comparison of execution times when all prefetchers are enabled versus the DPL prefetcher disabled versus the IP prefetcher disabled respectively.

2) Parsec Benchmarks with n Threads

Experiments 22 to 26 were conducted using four parallel threads on a 4-core machine and eight parallel threads on an 8-machine. As expected, there is an overall improvement in execution time with an average speedup of 2.2 over the single thread execution time on the 4-core machine with the highest speedup of 2.8 in *vips* benchmark program. Similarly, there is an average speedup of 3.3 over a single thread execution time on an 8-core machine. Fig. 7 shows the comparison between the execution times of Benchmark programs using a single and 8-threads on the 8-core machine.

The overall performance improves when prefetchers are enabled/ disabled for each of the benchmark programs. The best performance is achieved when the IP Prefetcher is disabled (experiment 25), where 11 out of 13 benchmark programs give an average improvement of 6.4% with the highest improvement of 19.3% in *streamcluster* program. This is a machine learning application that performs optimal clustering for a stream of data points. It is a prefetch sensitive application that benefits from pefetching into L1 cache. Fig. 8 gives the comparison of execution time of the benchmark programs when all prefetchers are enabled versus the IP prefetcher disabled versus all prefetchers disabled.

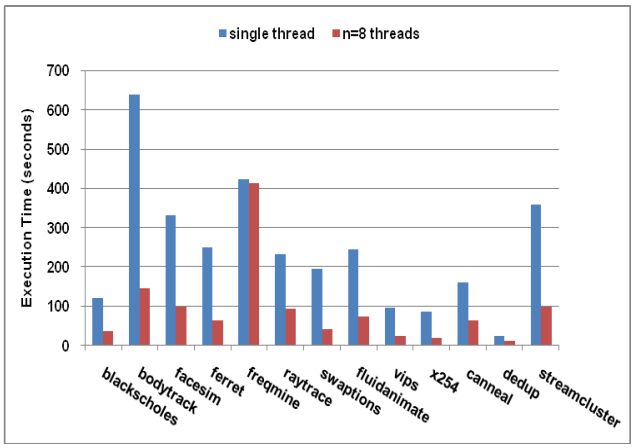

Fig. 7 Execution time of PARSEC Benchmarks with single and 8-thread on an 8-core machine (Platform No, 3)

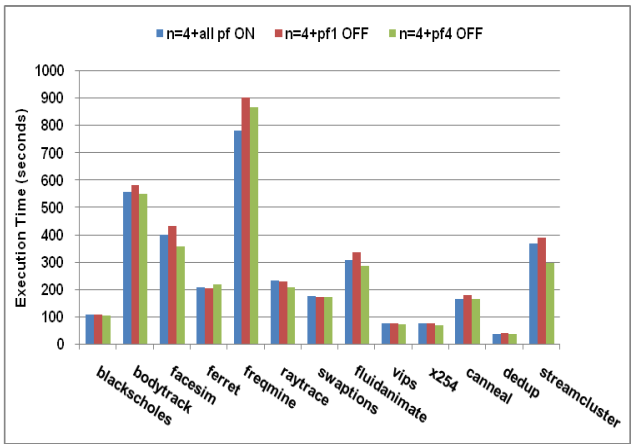

Fig. 8 Execution time of PARSEC Benchmarks with 4-threads on the 4-core machine (Platform No. 2)

D. Concurrent Matrix Multiplication

The platform used to run this program did not allow select-ive enabling and disabling of hardware prefetchers. This platform only allows all the prefetchers to be enabled/ disabled. The experiments were conducted by varying the number of matrix elements from 100x100, 1000x1000, 2000x2000 to 10000x10000 for both integer and floating point operands and by varying the number of cores from 4, 8, 12 to 24. Some results of Matrix multiplication program for integer and floating point operands on the 24-core platform is given in Fig. 9 (a) and (b) respectively. There is a degradation observed in all cases when the prefetchers are disabled with an average degradation of 6.7% for integer operands and 5.95% for floating point operands, indicating that the use of prefetchers is beneficial for concurrent matrix multiplication program.

E. Observations and Analysis

The results presented in this paper give an insight into the effectiveness of hardware prefetching, one of the most commonly used cache optimization technique in multicore processors. We have carried out a detailed set of experiments to estimate the performance with and without the built-in hardware prefetchers in multicore processors on a number of

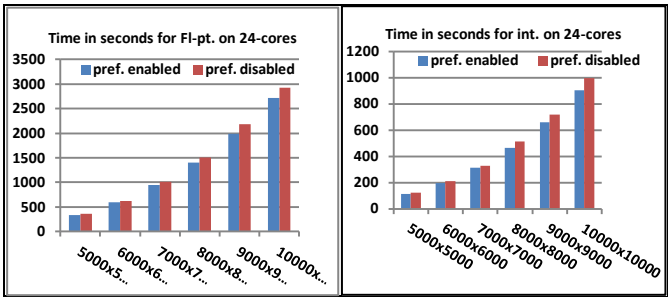

Fig. 9 Execution time of Conc. Matrix Mult. program for (a) Fl. Pt. (b) Integer

platforms. Both multiprogrammed and parallel workloads were used to study the effect. Two separate subsections briefly outline the observations and analysis of multiprogrammed and parallel workloads with various combinations of hardware prefetchers.

1) Multiprogrammed Workload

The results indicate that the effect of prefetching varies when an application is run as single program on a multicore processor compared to the case when the application is run as multiprogram workload. This is mainly because single programs suffer from less contention and interference. In general, most of the integer benchmarks benefit from prefetchers, whereas most of the floating-point benchmarks perform better without prefetchers.

Prefetching may be beneficial for applications that are prefetch sensitive. A larger number of integer applications are prefetch sensitive as compared to floating-point benchmarks. Even among integer applications, very few are prefetch sensitive, especially when run as multirpogram workloads. This is because the benefits of prefetching are overshadowed by the problems caused due to contention for resources and the interference between demand and prefetch requests generated by all cores. Some of the prefetches may also be useless. Prefetch to L1 cache by all cores cause redundant prefetches as multiple copies of the same block reside in multiple L1 caches. This also results in waste of cache space. In addition, all applications do not benefit from prefetching and do not exhibit the same behaviour for all types of prefetching. Database applications, image rendering through ray tracing, data mining applications and some image processing applications are some of the example areas that perform better with selective disabling of prefetchers.

The floating point benchmarks demonstrate a different behaviour pattern as compared to integer benchmarks. Most of these benchmarks perform better without prefetchers, especially when ACL prefetcher is disabled.

Another aspect that was explored was the performance of multicore processors for multiprogram workload. There is a significant increase in execution time as compared to single program workload. The average increase is 50% for a 4-core machine (4-copies) and 150% for an 8-core machine (8-copies) for SPECint benchmarks. Similar results are obtained for SPECfp benchmarks. The main reason attributed to this behaviour is the large amount of contention for resources,

which increases with the increasing number of cores. The proposed solution to this problem is that there should be a proportional increase in resources with the increase in the number of cores. This includes memory capacity, bandwidth of the interface between processor and memory and other components of the computer, as in case of conventional mutiprocessors. This is not what is observed from the architecture of multicore-based computers. Even though it may be possible to write fully parallel software that concurrently uses all cores of a multicore processor, the performance may not be as good as expected because of the above-mentioned reasons.

2) Parallel Workload

Since most of emerging applications for multicore processors are parallel workloads, the results obtained from these experiments are significant. When all prefetchers are enabled, average speedup of 4-threads execution is 2.2 over single thread execution (experiments 17 and 22). The speedup improves for most of the applications when the hardware prefetchers are manipulated. For example, the highest speedup of 3.1 is obtained when all prefetchers are disabled and *vips* program is run with four threads on the four-core machine (experiment 26). The main reason for this improvement is that there is less contention and interference among threads when prefetchers are disabled. The prefetch sensitive parallel benchmarks degrade in performance when hardware preftchers are disabled. For example, *freqmine* degrades in performance when prefetchers are disabled. This is a data mining application that identifies frequently occurring patterns in transaction databases. Fig. 8 gives an insight about other programs in this benchmark suite.

The use of prefetchers is beneficial for matrix multiplication program. The performance is better with prefetchers enabled because this is a data intensive application where the data access pattern is regular and predictable. Prefetching is considered to be suitable for such applications. The performance improves proportionately with the increase in the number of threads/ cores.

V. CONCLUSIONS

The role of hardware prefetchers have been exploited to examine their effectiveness in multicore processors with the goal of improving the overall system performance. Due to heavy sharing of on-chip resources including cache memory, there is degradation in performance when prefetchers are used aggressively, especially with multiprogram and parallel workloads.

The prefetchers need to be selectively enabled/ disabled depending upon the nature of the application and the type of prefetching. The selective use of prefetchers can control the interference of prefetch requests which interfere with demand requests due to extensive sharing of bandwidth at all levels of memory hierarchy and to the cache pollution caused due to useless prefetches. This results in better overall performance, thus effectively reducing the processor memory speed gap and lowering the memory wall.

Test results based on single program workload, concurrently running multiprogram workloads and parallel workloads confirm that appropriate enabling/ disabling of prefetchers can be used by application programmers to improve the execution time of programs. Experimental results indicate that database applications, image rendering applications, animation and some data mining applications perform better when prefetchers are disabled selectively.

REFERENCES

[1] J. Parkhurst, J. Darringer, B. Grundmann, “From Single Core to Multi-Core: Preparing for a new exponential”, Proc. of ICCAD, 2006, p. 67-72

[2] W. A. Wulf, S.A. McKee, “Hitting the Memory Wall – Implications of the Obvious”, ACM SIGARCH Computer Architecture News, 1995. p. 20-24

[3] J. Weidendorfer, “Understanding Memory Access Bottlenecks on Multicore”, Mini Symposium – Scalability and Usability of HPC Programming Tools, ParCo2007, FZ Julich

[4] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, D. Newell, “Exploring the Cache Design Space for Large Scale CMPs”, ACM SIGARCH Computer Architecture News, 2007, Volume 33, Issue 4: 24-33

[5] C. Kim, D. Burger, S. W. Keckler, “NUCA: A Non-Uniform Cache Access Architecture for Wire-Delay Dominated On-Chip Caches”, IEEE Micro, November/December 2003. p. 99-107

[6] N. Hardavellas, M. Ferdman, B. Falsafi, A. Ailamaki, “Near-Optimal Cache Block Placement with Reactive Non-uniform Cache Architecture”, IEEE Micro, January/February (2010), p. 20-28

[7] M. Zhang, K. Asanović, “Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors”, Proceedings of the 32nd International Symposium on Computer Architecture (ICSA-32), 2005. p. 336-345

[8] M. Hammoud, S. Cho, R. G. Melhem, “A Dynamic Pressure-Aware Associative Placement Strategy for Large Scale Chip Multiprocessors”, IEEE Computer Architecture Letters, Volume 9, No.1: January-June, 2010: 29-32

[9] H. Khatoon, S. H. Mirza, “Improving Memory Performance Using Cache Optimizations in Chip Multiprocessors”, Sindh University Research Journal (SURJ), Volume 43, Number 1A, June 2011: 43-50

[10] Y. Chen, H. Zhu, H. Jin, X. Sun, “Algorithm-level Feedback-controlled Adaptive data Prefetcher: Accelerating data access for high-performance processors”, Parallel Computing 38(2012) 533-551

[11] Y. Chen, H. Zhu, H. Jin, X. Sun, “Storage-Efficient Data Prefetching for High Performance Computing”, adfa, p.1, Springer-Verlag Berlin, 2012

[12] R. Nataranjan, V. Mekkat, W. C. Hsu. A. Zhai,” Effectiveness of Compiler Directed Prefetching on Data Mining benchmarks”, Journal of Circuits, Systems and Computers, Vol. 21, No.2, 2012, 23 pages.

[13] E. Ebrahimi, O. Mutlu, C. J. Lee, Y. N. Patt, “Coordinated Control of Multiple Prefetchers in Multi-Core Systems”, Proceedings of the 42nd International Symposium on Micro-architecture (MICRO), Dec.2009, New York. p. 327-336

[14] J. Lee, M. Shin, H. Kim, J. Kim, J. Huh, “Exploiting Mutual Awareness between Prefetchers and On-chip Networks in Multi-cores”, 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT 2011). p. 177-178

[15] N. Fukumoto, T. Mihara, K. Inoue, K. Murakami, “Analyzing the Impact of Data Prefetching on Chip MultiProcessors”, Proceedings of 13th Asia-Pacific Computer Systems Architecture Conference, 2008. p. 1-8

[16] M. Kamruzzaman, S. Swanson, D. M. Tullsen, “Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads”, Proceedings of ASPLOS 2011, ACM. p. 393-404

[17] C. J. Wu, M. Martonosi, “Characterization and Dynamic Mitigation of Intra-Application Cache Interference”, Proceedings of IEEE International Symposium on Performance Analysis of System & Software (ISPASS 2011). p. 2-11

[18] S. Verma, D. M. Koppelman, L. Peng, “Efficient Prefetching with Hybrid Schemes and Use of Program Feedback to Adjust Prefetcher Aggressiveness”, Journal of Instruction-Level Parallelism 13 (2011): 1-14

[19] C. J. Lee, V. Narasiman, O. Mutlu, Y. N. Patt, "Improving Memory Bank-Level Parallelism in the Presence of Prefetching", Proceedings of 42nd IEEE/ACM International Symposium on Micro-architecture (MICRO 2009), p. 327-336

[20] E. Ebrahimi, C. J. Lee, O. Mutlu, Y. N. Patt, "Prefetch-Aware Shared-Resource Management for Multi-Core Systems", Proc.of ISCA,2011

[21] N. C. Nachiappan, A. K. Mishra, M. Kandemir, A. Sivasubramanium, O. Mutlu, C. R. Das, "Application-aware Prefetch Prioritization in On-Chip Networks", Proceedings of PACT, 2012

[22] C. J. Wu, A Jaleel, M. Martonosi, S.C. Steely Jr.,J. Emer, "PACMan: Prefetch-aware Cache Management for High Performance Computing", MICRO 2011

[23] J. Lee, H. Kim, M. Shin, J. Kim. J. Huh, "Mutually Aware Prefetch and On-chip Network Designs for Multi-cores", IEEE Transactions on Computers, Preprint, 26 April 2013.

[24] R. Manikantan, R. Govindarajan, " Performance-oriented Prefetch Enhancements Using Commit Stalls", Journal of Instruction-level Parallelism 13(2011) 1-28

[25] M. Grannaes, M. Jahre, L. Natvig, "Storage Efficient Hardware Prefetching using Delta-Correlating Prediction Tables", Journal of Instruction-Level Parallelism 13 (2011)

[26] Order Number 325462-040US. Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C. October 2011

[27] SPECCPU2006 Standard Performance Evaluation Corporation. Details can be found at the web site http://www.spec.org/

[28] PARSEC (Princeton Application Repository for Shared-Memory Computers) website: address follows. Parsec v 2.1 Benchmark suite from the following website:

[29] M. A. Ismail, S. H. Mirza, T. Altaf, "Concurrent Matrix Multiplication on Multi-Core Processors", International Journal of Computer Science and Security (IJCSS), Volume (5): Issue (2): 2011, p. 208-220

[30] C. Bienia, "Benchmarking Modern Multiprocessors", PhD Thesis, Department of Computer Science, Princeton University, January 2011

[31] M. Bhadauria, V. Weaver, S. Mckee, "Understanding PARSEC Performance on Contemporary CMPs", Proceedings of 2009 IEEE International Symposium on Workload Characterization, p. 98-107

[32] msr tools and documentation from any of the following web sites
sourceforge.net/projects/msr
www.kernel.org/pub/linux/utils/cpu/msr_tools
Downloaded in April 2012
http://parsec.cs.princeton.edu downloaded in April 2012