Sapna Prabhu, Dr. R.D. Daruwala / International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622 www.ijera.com
Vol. 3, Issue 2, March -April 2013, pp.098-102

Measuring Performance Degradation in Multi-core Processors due to Shared resources

Sapna Prabhu, Dr. R.D. Daruwala
*Research Scholar VJTI, Mumbai
**Professor VJTI, Mumbai

Abstract

The effect of resource sharing in multi-core processors can lead to many more effects most of which are undesirable. This effect of Cross-core interference is a major performance bottleneck. It is important that Chip multiprocessors (CMPs) incorporate methods that minimise this interference. To do so, some accurate measure of Cross Core Interference needs to be devised. This paper studies the relation between Instructions per cycle (IPC) of a core and the cache miss rate across various workloads of the SPECCPU 2006 benchmark suite by conducting experimentation on a Full System simulator and makes some important observations that need to be taken into account while allocating resources to a core in multi-core processors.

Keywords*:* Chip Multiprocessors (CMPs), Cross-Core Interference, Pre-fetching, Instructions Per Cycle (IPC), LLC (Last Level cache)Miss rate

I.INTRODUCTION

The need for better performance from computing systems has always been on the rise and is predicted to continue. This has necessitated the advent of high performance microprocessors with sophisticated architectures. The increased hardware has however led to power levels above acceptable limits. Hence, leading microprocessor manufacturers like INTEL and AMD have adopted a shift in computing paradigm by introducing the concept of Multi-core processing. Multi-core processing can be achieved by multiple cores on a single processor die which run at lower clock frequencies than their single-core counterparts. Since, these cores work in parallel, the performance is bound to improve. Also, power consumption can be kept within limits.

The cores generally share a number of resources to avoid hardware duplication like the caches, pre-fetch buffers, Front-Side Bus controllers to name a few. For eg. Consider the Intel Xeon Quad core processor where the last level cache is shared between a pair of cores. (Core 0 and Core 1 share a L2 cache and Core 2 and Core 3 share another L2) . Resource Sharing can lead to various effects , some which may enhance the performance while others that can lead to performance degradation. For the rest of the paper, Shared resources will be restricted to sharing of the last level cache.

The Shared cache can be beneficial for Inter -Core Communication . Also, if the two cores need the same data, sharing will improve performance. The bigger issues related to sharing of resources however are due to the slowdown of applications that run in parallel on neighbouring cores. Section II will explain the phenomenon of Cross-core Interference.

II. CROSS-CORE INTERFERENCE

To bridge the speed disparity between the cores and main memory, all the cores are normally attached to two or more levels of caches. The cache architecture varies across processors with some having private caches and others having a combination of private and shared cache. Generally, most cores have private caches and share the last level cache (maybe L2 or L3). Figure 1 shows one such dual-core dual processor system.

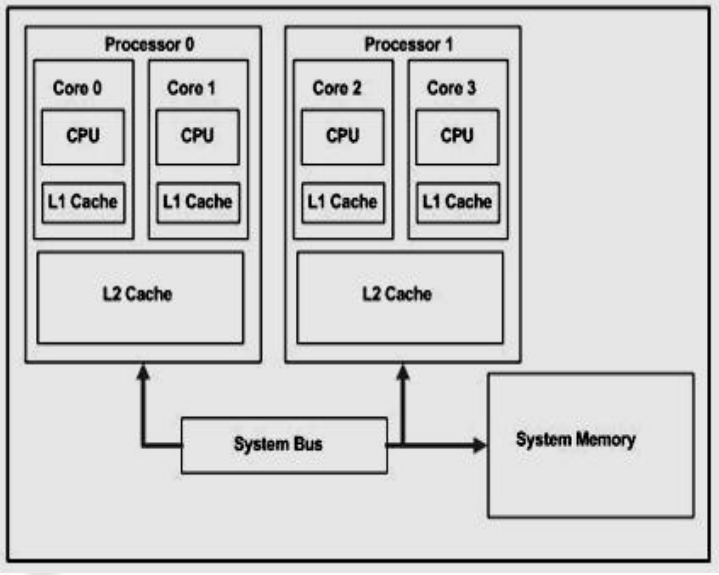


Figure 1: A Dual-core Dual processor system

We shall assume that the last level is L2 throughout the remaining part of the paper. In this case, Core 0 and Core 1 in a dual core processor have a private L1 per core and both the cores share an L2. In such a case, if Core 0 brought in a line of cache into L2 due to a cache miss, it is possible that Core 1 may evict that valid line to accommodate a line required by it. To summarise , Core 0 may

experience cache misses due to Core 1 and vice versa. This effect is called Cross-core interference and is highly undesirable. Due to sharing of L2 , performance of an application running on Core 0 will depend on the number of additional cache misses that it experiences due to the application running on Core 1.

One of the main objectives in Multi-core processing is Performance isolation where-in the performance of all the cores must be independent of each other. Cross-Core Interference leads to poor performance isolation and may result in higher order system-level effects like priority inversion and thread starvation as it brings in indeterminism in the system. The execution time of a thread can vary greatly depending on its co-running thread. Note that the performance of a core is therefore dependent on the nature of its co-running thread on the neighbouring core. This is termed as Performance variability and is a result of poor Performance Isolation between cores [1] .

Thus, Cross-Core interference can cause a major performance degradation in a multi-core system. It is interesting to note that this interference is workload specific . Thus, measuring the impact of cache interference on performance is a multi variate problem that is considerably challenging [2] . The nature of cross-core interference experienced by an application due to its co-running application depends on the nature of both the applications.

A. Factors affecting Cross Core Interference

Some application workloads are compute intensive and others may be memory intensive. Applications that contain a large number of memory accesses are generally prone to more cache misses than those which make fewer accesses. Another important characteristic of applications is its Memory Reuse or Temporal locality [5]. It has been observed that applications that reuse their data well (have good temporal locality) may experience less misses despite having more accesses. Another important characteristic that needs to be discussed here is that some applications may cause more contention and affect co-runners substantially while may itself not get affected by its co-runners. The converse is also noted to be true that applications may get affected by others but may not harm others performance considerably [3]. The Working set size of an application also determines performance of the application as a function of cache size. Some applications show low miss rates when the amount of cache allotted to them increases. There are others who show low performance improvement to larger caches [4].

B. Effects of Pre-fetching mechanism

Most Multi-core processors include Hardware Pre-fetchers that are triggered to prefetch into the cache from main memory. Pre-fetching is one such technique that helps alleviate potential bottlenecks, by fetching instructions and/or data from memory into the cache well before the processor needs it, thus improving the load-to-use latency. For eg. Intel Pentium 4 includes Automatic Hardware Pre-fetch and Adjacent Cache Line Pre-fetch. The Hardware Pre-fetcher operates transparently, without programmer intervention, to fetch streams of data and instruction from memory into the unified second-level cache. The Pre-fetcher is capable of handling multiple streams in either the forward or backward direction. It is triggered when successive cache misses occur in the last-level cache and a stride in the access pattern is detected. The Adjacent Cache-Line Pre-fetch mechanism, like Automatic Hardware Pre-fetch, operates without programmer intervention. When enabled through the BIOS, two 64-byte cache lines are fetched into a 128-byte sector, regardless of whether the additional cache line has been requested or not [13].

To minimise cross-core interference , it is important to accurately measure the extent of interference and also the cause for it. This is done by performing Workload Characterisation .A dependable heuristic to measure the interference needs to be devised which can be easily obtained in on-line as well as off-line studies. Section III discusses related work in Workload Characterisation as well as interference measurement.

III. Related Work

Several other studies have proposed various methods to perform workload characterisation of applications which is an important step in measuring Cross-core interference. As mentioned in the earlier Section, application characteristics play an important role in determining the effect on their own as well as their co-running application's performance when both applications are sharing resources.

A. Workload Characterization

Qureshi et al. ,in his paper has studied characteristics of various applications based on their response to size of cache [4]. The general observation is that some applications demonstrate lower miss rates as the size of cache allotted to them is increased . These are called high utility applications while those who do not show any substantial change in miss rate are called low-utility applications. Another category of applications called saturating utility applications show a good response as cache size increases to a certain point after which their response remains almost constant. Tang et al. propose two metrics namely Contentiousness and

Sensitivity [3]. An application's contentiousness is defined as the potential performance degradation it can cause to co-running application(s) due to its heavy demand on shared resources. On the other hand, an application's sensitivity to contention is defined by its potential to suffer performance degradation from the interference caused by its contentious co-runners. The paper concludes that applications' Contentiousness and Sensitivity are not strongly co-related and all all applications can be 1) contentious and sensitive; 2) not contentious and insensitive; 3) contentious but not highly sensitive; 4) not highly contentious but sensitive. Xie et al. in his paper has proposed a classification algorithm for determining the "personalities" of the programs with respect to their cache sharing behaviors [8]. They classify benchmarks into intuitive "animal" personalities based on a few simple heuristic metrics. Applications are classified into four categorized namely Turtles, Sheep, Rabbits and Devils. Turtles are some applications that simply do not make much use of the shared last-level (L2) cache. This may be because the program simply has very few memory instructions to begin with, or it may be that the program has a very small working set that completely fits within the level-one caches and therefore rarely accesses the L2 cache. Sheep applications are those which are not sensitive to the number of ways allocated to them. These applications may actually exhibit a high rate of L2 accesses, but even with an allocation of only a few ways, these programs can achieve a low L2 miss rate. Some applications are very sensitive about the number of ways allocated to them. Such applications access the L2 cache fairly frequently, but if provided with a sufficient number of ways, the overall miss rate can be kept low. These are called Rabbit applications. The last class of applications called Devils simply do not "play well with others." These applications access the L2 cache very frequently, but still have very high miss rates. As a result, such applications do not derive much benefit from occupying the cache (in terms of hit-rate reduction), and furthermore they tend to negatively impact other applications

B.LLC Miss Rate as an indicator of Cross-core Interference ?

Several studies have proposed different heuristcs , direct and indirect , which can be used to measure cross-core interference. Zhuravlev et al. have summarised classification schemes namely Stack distance Competition(SDC), Animal Classes, Miss rate and applications based on a metric called Pain proposed by them. The paper indicates that a simple metric like the cache miss rate can be used to classify applications and this scheme performs almost if not better than the other methods [6]. Tang et al. have concluded that miss rate cannot be used to measure the contentiousness and sensitivity of all types of applications as low miss rate applications are also noted to cause contention to other co-running applications [3].

The next section discusses the relation between IPC and LLC Miss rate when applications share caches with other applications.

IV . EXPERIMENTAL SETUP

For our experimentation , we use Virtutech SIMICS full system simulator which has been extended to include a cache hierarchy [10] . For the setup, we have used a X86-440BX target .This target supports various configurations namely tango, enterprise, cosmo or hippie.

Tango has Fedora Core 5 installed. The base configuration has a single 20 MHz Pentium 4 processor, 256 MB memory, one 19GB IDE disk and one IDE CD-ROM [5]. There is also an AGP based Voodoo3 graphics card and a PCI based DEC21143 network adapter. Our setup comprises of a dual core processor using the tango configuration with each core having a separate code and data Level 1 cache and Level 2 cache being shared between the two cores . Table 1 explains the setup in more detail .

Number of Cores	2
Level 1 (L1) cache/Core	32 KB Instruction cache+ 32 KB Data cache ,lru replacement policy, Read/Write penalty =3 cycles
Level 2 (L2)cache	1 MB Unified cache, lru replacement policy, Read/write penalty =10 cycles

Table 1: Baseline Configuration

The experimentation has been carried out by running programs of the SPECCPU 2006 Benchmark suite. The SPEC CPU2006 benchmark suite consists on a set of 12 programs for integers (SPECCPUint2006) written in C and C++ and 17 programs for floating point (SPEC CPUfp2006) written in C, C++ and FORTRAN [12]. The objective of these computer-intensive programs is to provide portable, credible and real-world application-based benchmarks for quantifying the performance of the set processor, memory and compiler.

The experimentation has been carried out in two steps. First, the programs are run solo in the dual core processor by binding it to Core 0 and statistics are collected. The application is bound to the core using the taskset utility. Secondly, the each program is run with different co-runner applications, by binding the applications to Core 0 and Core 1 and paired statistics are collected [11].

Since the SPECCPU 2006 has a large instruction set as compared to its predecessor , the SPECCPU 2000 suite , the simulation times involved are huge. Hence, a representative subset for the same is used for the experimentation [9]. Ten benchmarks from the SPEC2006 benchmark suite have been selected to represent a wide range of cache access behaviours. The cache miss rates and access rates for every application in the SPEC2006 benchmark suite were obtained from a third party characterization report [6] and a clustering technique was employed to select the ten representative applications namely mcf, lbm, milc, soplex, astar, sphinx3, libquantum, namd, gamess, povray.

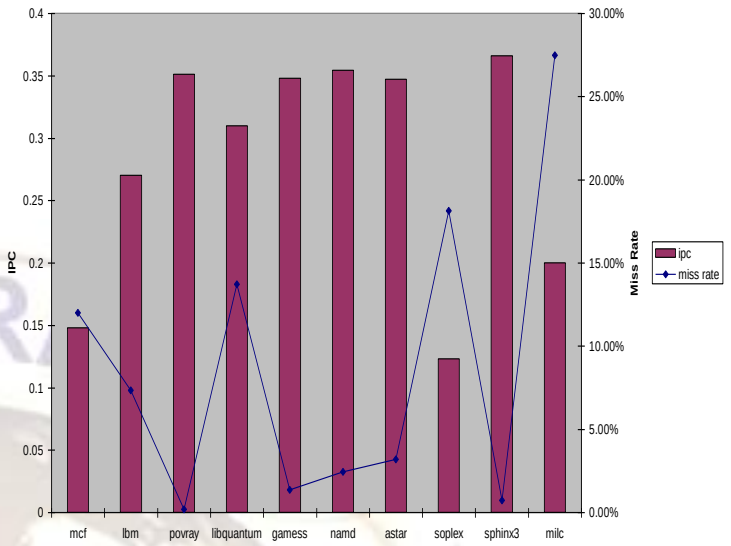
Also, all the programs are run using the runspec utility with ref inputs. As running the programs in a simulated setup takes several orders of time more than running the programs on a real machine, a sampled window of instructions has been chosen to run the experiments. All the applications are fast-forwarded past their initialization codes and then run for 1 billion instructions [7]. Then after caches have been attached, they are further run for 100 million instructions to "warm" up the caches. After the caches have been flushed to eliminate the effect of of compulsory cache misses, the benchmarks are further run for 500 million instructions before collecting statistics.

V. RESULTS

The statistics were collected or solo as well as paired run for the benchmarks. The statistics collected are IPC (Instructions per cycle) for core and Level 2 cache misses (Data read+ data write+ Instruction fetch).

Figure 2 shows the solo characteristics of the benchmarks.

Figure 2 Solo IPC and Miss rates of SPECCPU 2006 programs



From the above solo characteristics, we can broadly classify the programs as high miss rate (milc, soplex ,mcf, libquantum, lbm) and low miss rate applications (povray, namd, astar,gamess,sphinx3).

Further, each application was run with different co-runners and statistics were collected. Since the number of combinations is considerable, some of the readings are shown in the table below

		Solo	soplex	astar	milc	namd
mcf	IPC	0.148	0.25	0.106	0.096	0.108
	MR	11.98%	13.82%	14.16%	15.75%	10.59%
astar	IPC	0.3472	0.285	0.309	0.33	0.342
	MR	3.1966%	16.7%	8.87%	4.56%	3.53%
povray	IPC	0.351	0.351	0.351	0.351	0.351
	MR	0.166%	3.38%	1.156%	5.96%	0.61%
milc	IPC	0.2	0.47	0.29	0.319	0.27
	MR	27.47%	1.43%	7.02%	3.18%	7.52%
sphinx3	IPC	0.366	0.362	0.292	0.361	0.364
	MR	0.723%	9.13%	7.21%	5.31%	0.223%

Table 2 : IPC and miss rates of applications (solo and paired)

Interpretation of results:

Applications under test are categorised as high miss rate and low miss rate applications. Low miss rate applications may have low miss rates either because they have less memory accesses or may have more accesses but show low miss rates due to excellent memory reuse.

For eg. Low miss rate applications like astar show variation in miss rate almost upto 16.7% depending on co-runner and a proportional variation in IPC is also noted. However, another low miss rate application like povray shows lesser variation in miss rate and its IPC remains constant . This is

because povray has excellent temporal locality (good memory reuse). Also . it is less sensitive as compared to astar to co-running applications but because the number of accesses is large , the small variation in miss rate does not change IPC. Another low miss rate application sphinx3 shows variation in miss rate upto 9.13% and a proportional change in IPC.

However, high miss rate application like mcf shows a decrease in IPC with increasing miss rates. However the IPC gets impacted largely for small changes in miss rate as the number of misses is large. Another high miss rate application like milc however shows speedups rather than slowdown when run with other applications, This can be attributed to the pre-fetching mechanism in Pentium 4 as discussed in Section II. As milc hardly reuses its data, prefetching benefits are more prominent as compared to mcf.

VI. CONCLUSIONS

From the above experiments , two conclusions can be made . The LLC miss rate is not a dependable measure of cross-core interference across different workloads as it does not clearly quantify the extent of interference which is crucial in resource allocation The variation also depends on the number of accesses as well as misses experienced by the application (solo characteristics). Another important conclusion is that by using workload characterisation to study effects of pre-fetching across different workloads, better performance benefits can be derived. For effective Shared resource management in Multi-core processors, both these conclusions are important. Future work may include studying the effectiveness of pre-fetching across application workloads .

REFERENCES

1. Alexandra Fedorova, Margo Seltzer , Michael D. Smith, Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler, Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques(PACT'07), Page(s): 25-36.
2. Alex Settle, Dan Connors, Enric Gilbert, Antonio Gonzalez, A dynamically reconfigurable cache for multithreaded processors, Journal of Embedded Computing, Volume 2 Issue 2, April 2006, Page(s):221-233.
3. Lingjia Tang, Jason Mars, Mary Lou Soffa, Contentiousness vs Sensitivity: improving contention aware runtime systems on multicar architectures, Proceedings of 1st International Workshop on Adaptive Self-tuning Computing Systems for the Exaflop Era (EXADAPT'11), Pgs. 12-21
4. Moinuddin K. Qureshi and Yale N. Patt, Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches, in Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39). IEEE Computer Society: Orlando, Florida, USA, 2006, Pgs. 423–432
5. Nikrouz Faroughi, Profiling of parallel processing programs on shared memory multiprocessors using Simics, ACM SIGARCH, Pgs.51-56.
6. Sergey Zhuravlev, Sergey Blagodurov, Alexandra Fedorova, Addressing Shared Resource Contention in Multi-core Processors via Scheduling , Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems,ASPLOS'10, Pgs.129-142.
7. Xiaomin Jia,Jiang Jiang , Tianlei Jhao,Shubo Qi,Minxuan Zhang, Towards Online Application Cache Behaviors Identification in CMPs, Proceedings of the High Performance Computing and Communications (HPCC), 2010 , Pgs. 1-8
8. Y Xie, G H Loh. Dynamic classification of program behaviors in CMPs. Proc Workshop on Chip Multiprocessor Memory Systems and Interconnects. Beijing, China, 2008., Pgs .28–36.
9. Rafael Rico, SPEC CPUint2006 characterization, Technical Report TR-HPC -01-2009
10. Simics Programming Guide, Version 3.0.
11. Simics User Guide for Unix , Version 3.0.
12. www.spec.org
13. www.software.intel.com