Cache Sharing Administration for Performance Fairness using D3C Miss Classification in Chip Multi-Processors

Claudio A. Carballal, José Luis Hamkalo, Bruno Cernuschi-Frías

Facultad de Ingeniería, Universidad de Buenos Aires.
Av. Paseo Colón 850, PB, (C1063ACV), Buenos Aires. Argentina.
ccarballal@gmail.com.ar
{jhamkal,bcf}@fi.uba.ar

Abstract. This work presents a study of fairness in cache sharing between processes in a chip multiprocessor (CMP). We propose a new algorithm that uses a metric based on the D3C miss classification and LRU Stack Distance, to measure the fairness in the administration of the resources to achieve an increase of the global IPC of all executed processes. Shared cache miss rate, IPC and bandwidth metrics were considered to analyze the simulation results obtained using three test sets. The obtained results showed that the proposed dynamic management policy compared to Capitalist management policy, has a lower global miss rate in shared cache and lower bandwidth usage for each test set studied and fulfills its objective of managing the shared cache space for every process while improving the overall IPC.

Keywords: Shared Cache, Multi-Process, CMP, Dynamic Cache Administration, Instrumentation, PIN.

1 Introduction

Multi-Core architectures or Chip Multi-Processors (CMP) developments proved to be a remarkable improvement in applications performance, as they provide an efficient way to run multiple applications, or just one (through different subtasks) simultaneously. This improvement in execution efficiency is achieved by multiple individual processing units with private and shared memory resources available [5].

At present, some processor architectures count on multiple cache memory levels. For the sake of simplicity and performance, some levels are kept private for each processor core, while other architectures explore sharing the last level of cache among different cores of the processor [11, 7, 20, 18]. Shared caches memories are highly beneficial to performance, because in the extreme case of a single core active while others inactive, the process being executed would count with all shared cache memory, which is larger than all those used as private memories for

each core. Moreover, they provide high performance when cores share information among themselves, reducing latency and coherence. However, they represent a major challenge in terms of administration, since multiple processes access it simultaneously giving place to situations that in private caches never occur, such as over-writing information from another process (causing inter-process misses) or anything that may affect performance even more, such as changes in the execution context of different programs when executed [9]. For this reason, it is desirable to grant to each process in execution an amount of shared cache memory so as to achieve the best possible performance without causing an efficiency decrease of other processes that are also in progress.

When searching to achieve this goal, in a previous study [6], a tool was introduced to capture and process, on the fly, instructions from multiple processes being executed simultaneously by using interchangeable modules. This tool uses the PIN framework [1, 16] to dynamically instrument executed processes and a process Controller to manage a step by step execution, and to send the collected information to the processing modules. A processing module that simulates a cache memory hierarchy of three levels was developed to obtain information from different management policies for shared caches. For shared cache memory administration three high-level policies are defined [13]. A "Communist" approach seeks to maximize fairness, ensuring that each thread or process bears an equal benefit from the presence of the cache. The goal for "Utilitarian" policy is to maximize the total benefit for the aggregate group, by maximizing total throughput and the "Capitalist" cache policy is an unregulated free-for-all (the most common policy in use today).

The present work is organized as follows: in the next section the PIN framework functioning and the tool to capture memory references of the processes in execution are presented. Next, the new dynamic management policy developed is introduced, which main objective is to improve the applications overall performance by using a metric based on the D3C miss classification, so as to measure and manage the space needed for each process in shared cache memories. Finally, it is presented the analysis of the results obtained from simulating a selected group of benchmarks from the SPEC CPU 2006 [2] grouped as test sets. This final analysis takes into account the shared cache miss rate, the IPC and the bandwidth used by the new management policy presented here, in and it is compared with Capitalist management policy, used as reference in other related works.

2 Workflow Design

Instrumentation is a technique to insert extra code into an application in order to observe its behavior. This can be done in several stages, such as: in the source code, during compilation, at the post-link stage or during execution. PIN provides a complete API (Application Programming Interface) that serves as an interface abstraction layer to interact with the applications to be implemented, making it possible to write different instrumentation tools called *Pintools*. A

JIT (just-in time) compiler is used to insert and optimize the instrumentation code. PIN compiles the application directly in the same ISA (Instruction Set Architecture) without passing through an intermediate code, storing the new compiled code for execution in an internal cache. The architecture has a virtual machine (Virtual Machine (VM)) which is the JIT compiler, an emulator that interprets, and emulates certain instructions that cannot be executed directly (usually system calls that require special treatment) and a dispatcher that is responsible for "dispatching" the instructions from the application to the JIT compiler.

2.1 Execution Behavior

PIN is able to implement an executable binary even if it creates new child processes or new processing threads. A copy of the Pintool in the parent process will control each new child process created, while new threads will be controlled and synchronized using the API provided by the PIN inside the Pintool. Bearing in mind the behavior of the applications to be implemented, we have chosen the design presented below. It consists of a Pintool, a process Controller, two message buffers through which the Pintool and the process Controller communicate. The controller is responsible for administering each process implemented by the Pintool, during the execution process, indicating when the execution may proceed, prior record of the process in the controller [6].

1. The Pintool detects the beginning of the process and enters in the register buffer the PID (Process Identifier) of the new process to be implemented.
2. The Controller reads the PID of the new process to control and stores it in the iterative sequence of execution of processes.
3. Each Pintool sends instrumentation information to its exclusive communication buffer with the Controller.
4. Iteratively the driver takes the information from the communication buffer for each registered process and sends it to the processing module.
5. After the information is sent to the processing module, it is returned to the previous step.

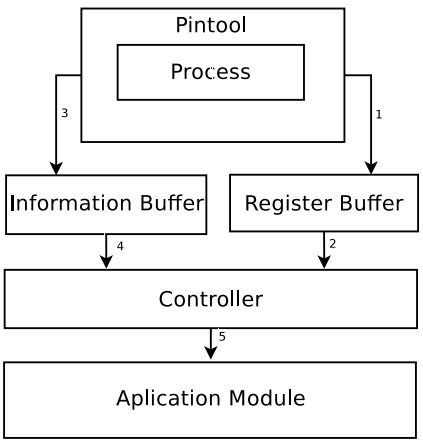


Fig. 1: Sequence diagram of the development environment

3 Algorithm for Dynamic Management

In multi-core architectures and CMPs, the last level of cache memory is typically shared by multiple cores to maximize the use of resources by avoiding low latency information duplication and by reducing traffic information by lowering coherence. Furthermore, shared cache memories allow dynamic allocation of more flexible resources; in an extreme case, a core can use all the cache space when all the other cores are in idle state (this technology is applied in Intel SmartCache and similars [17]). Shared cache memory uses space in a more flexibly way as it reduces the number of accesses to the main memory. However, it has a higher latency that propagates to lower levels. On the contrary, private cache memory has lower latency but it inefficiently uses space because the hierarchy contains multiple copies of data shared among processes.

Finally, the absence of isolation and administration on the use of space in cache memories results in performance degradation. The above described effects are increased by the competition for resources in multi-process environments. For these reasons, it is necessary to propose resources administration policies in order to minimize the above mentioned effects [21].

3.1 D3C Miss Classification

Miss classification according to 3C model [12], is conceptually clear and intuitive, and widely used in the scientific community for performance assessment. However, it presents anomalies in certain cases that result in negative conflict miss rates difficult to interpret when evaluating caches. These cases occur when associative caches in working sets have a lower miss rate than fully associative ones. When calculating conflict miss rates in the 3C model, all failed references in the fully associative cache are subtracted from the total number of failed references in the cache being studied, regardless of whether some of these references are correct in the cache memory being studied. Thus, the negative value in a conflict miss rate implies that it is higher the number of failed references in a fully associative cache than the correct references in the cache being studied. An additional drawback is the impossibility of classifying a miss individually, since it is a statistical model.

Therefore in [10], a Deterministic 3C Model (D3C) is defined which uses the same miss classification in terms of Capacity, Conflict and Compulsory but redefining each concept:

"A capacity miss in the D3C model is every reference to memory that produces a non-compulsory miss in the cache being studied which also fails in a fully associative one with LRU replacement of the same size. All the other non-compulsory misses are defined as conflict misses, i.e. references that fail in the cache being studied and are correct in a fully associative cache of the same size."

This new classification makes it possible to determine for each memory reference of a process the type of miss involved so as to modify the available resources

to improve its performance. If conflict misses were majority in a given number of memory references C, it would be desirable to increase the cache associativity, since the greater associativity the lesser conflict misses. If, instead, the majority were capacity misses, it would be advisable to increase the number of blocks per way. Compulsory misses should not be considered because compared to capacity and conflict misses they are really very few, and if necessary, eventually reduced or even eliminated, using the prefetching technique [11].

3.2 LRU Stack Distance

In 1970, Mattson et al. [8] published the first technique for evaluating various virtual memory replacement strategies using stacks. His algorithm takes a trace of memory references, cache line references or virtual page references in a program, and builds a stack as follows: every memory location is pushed onto the stack when it is referenced. Locations that have been referenced a long time ago sink towards the bottom of the stack, but locations that are referenced again are extracted from the stack and pushed back on top, being the LRU stack distance for a given reference defined by its depth in the stack [4].

This technique was so successful, since it was easy to implement and to apply in simulations, that it was quickly adopted and used by the scientific community. With this algorithm, an operational definition of the D3C classification can be defined:

Given a cache size equal to B blocks, and memory reference LRU distance D causing a miss, then the miss is classified as follows:

- *Compulsory: D not computable.*
- *Capacity:* $D > B$.
- *Conflict:* $D \leq B$.

Therefore, using the D3C classification together with the LRU distance it is possible to classify a process memory reference miss.

3.3 Metrics to evaluate performance fairness

Since it is necessary to evaluate fairness of resources allocation for each process during execution, it must be provided a definition of a metric to determine the actual performance of each process according to their allocated memory resources and the miss rate it produces in a time slice.

Metric for performance fairness in [14, 22] are expressed as follows:

$$\frac{T_i^{Sha}}{T_i^{Ded}} = \frac{T_j^{Sha}}{T_j^{Ded}} \tag{1}$$

This is the measure of fairness for two processes i, j. T_i^{Sha} time is the run time for the process i in an environment where they share resources with the process j. T_i^{Ded} is the execution time of process i with all resources for itself (dedicated). Of the variables included in the calculation of runtime dynamic management proposed in this paper is aimed at reducing the miss rate in the shared cache, leaving invariant the other variables involved. Therefore, the equation (1) can be expressed in terms of the miss rate in shared cache by applying the operational definition of LRU Stack Distance and the D3C model.

$$\frac{Misses^{Sha}_{capacity\ i}}{Misses^{Ded}_{capacity\ i}} = \frac{Misses^{Sha}_{capacity\ j}}{Misses^{Ded}_{capacity\ j}} \tag{2}$$

In order to obtain the ideal performance, the fraction value for each process is sought to be as close to unity as possible. As stated in [12], it is concluded that increasing associativity instead of cache size, only reduces conflict misses, while increasing size without increasing associativity can reduce conflict and capacity misses. This is also indicated in [3] for SPEC CPU 2000 benchmarks. With these concepts, the heuristic proposed provides total associativity to cache memory, and to decrease it, it should be evaluated if all processes make use of lower associativity than provided, or if external requirements such as an energy consumption reduction make it necessary.

The equation for multiple processes in progress (2) is defined by the expression FMDCM: Fairness Metric for Deterministic Cache Misses:

$$FMDCM = \sum_{i}^{N} \sum_{j}^{N} \left| \frac{Misses^{Sha}_{capacity\ i}}{Misses^{Ded}_{capacity\ i}} - \frac{Misses^{Sha}_{capacity\ j}}{Misses^{Ded}_{capacity\ j}} \right| \forall i \neq j \tag{3}$$

The practical technique for implementing (3) is described in the following sequence of steps:

1. Define LRU stack size equal to the number of blocks in the cache memory. Define a mark with the number of cache blocks allocated to the process to be analyzed.
2. If the memory reference miss is not found in the stack, a dedicated and shared capacity miss will be indicated (or compulsory).
3. If the memory reference is in the stack, and its LRU distance value is higher to the mark indicated (which is the amount of cache blocks allocated to the process at that moment) it will be a shared capacity miss.
4. Finally, if the memory reference is in the stack and its LRU distance value is lower than the mark indicated, it indicates a conflict miss.

To achieve the best performance of all processes being executed, the value of equation (3) must be minimized each time the amount of cache sets allocated to each process is administrated.

$$|FMDCM_{administration\ i}| < |FMDCM_{administration\ i-1}| \quad (4)$$

Metric variation (3) to remove or add a certain amount of cache sets as necessary is still to be defined, in order to accomplish the objectives in (4). Each constant time slices the equation (3) is applied for every executing process.

When using a technique similar to that proposed in [15] to manage the space allocated to each process, tiles containing a fixed number of cache sets are created, and also exchanged with other processes. Each tile contains information about the number of hits and shared or dedicated capacity misses that occurred in the execution period before the management heuristics was applied.

By removing a tile allocated to a process, an increase in the metric (2) for that process must be obtained. This value is the result of the addition of the deallocated tile shared misses (misses having a LRU distance greater than the amount of cache blocks allocated and lower than the total number of shared cache blocks) and the total number of shared misses, preserving the amount of dedicated misses. Experimentally, it was found that this calculated value is a good predictor of performance decline in the process whose cache available space was reduced. The calculation of capacity misses reduction for the process which was allocated a new tile (resulting in a performance increase) is less accurate since this value cannot be determined in a simple way because it depends on the use the process makes of it (in terms of number of hits). The value adopted was the tile value with less shared misses. So, the final value calculated was obtained by subtracting the shared misses from the available tiles and the shared misses from the tiles with less shared misses, preserving the amount of dedicated misses. Though this may be objected claiming that total average of capacity misses or misses from tiles with the lowest and highest number of accesses could be more accurate, the test results demonstrates that this metric is effective, practical and easy to implement.

The last consideration to take into account is that, in certain iterations, the value of dedicated misses is equal to 0. This can happen because the process had mainly shared misses or only conflict misses, which would cause shared misses being equal to 0. In the first case in which dedicated misses are equal to 0, a metric was adopted to obtain the amount of shared misses. The reason for this decision is that if a process only has this kind of misses, it will be benefited with increased space, since data size is the same or lower than the shared cache. If multiple processes are in this state, the metrics will give more space to the one with the highest number of misses. In the second case, the metric obtains the ideal value of 1, since having no shared dedicated misses indicates the ideal value for (2).

Metrics applied for evaluating performance on simultaneous processes in progress were presented as well as the shared cache dynamic management policies used.

4 Experimental Methodologies

4.1 Simulated Architecture

Bearing in mind the memory hierarchies used in the most recents CMP's models, the architecture chosen for the simulation is the following:

- 2 cores.
- Private split L1. 32KB, 4 ways y 64B block size. 1 cycle for hits.
- Private and unified L2. 256KB, 8 ways y 64B block size. 7 cycles for hits.
- Shared and unified L3. 2MB, 16 ways y 64B block size. 20 cycles for hits.
- 380 cycles for main memory latency.
- 4 instructions in parallel execution per core.
- 50 cycles of time of time of administration (if applicable).
- 300000 cycles run between administrations (*time slice*).

Three test sets consisting of two benchmarks each were applied.

(A) H264 and Libquantum.
(B) Gcc and Libquantum.
(C) H264 and Gcc.

Every test set was executed for 1000 million instructions without being instrumented to eliminate the startup effects of every benchmark, and then instrumented for a total of 200 million instructions.

4.2 Simulation Results

In order to achieve the main objective of the proposed management policy, different parameters for measuring performance of processes in progress are affected. For this reason, several metrics were considered to analyze the simulation results obtained. The metrics used were:

1. *Shared cache miss rate*
2. *Instructions Per Cycles (IPC)*
3. *Bandwidth used*

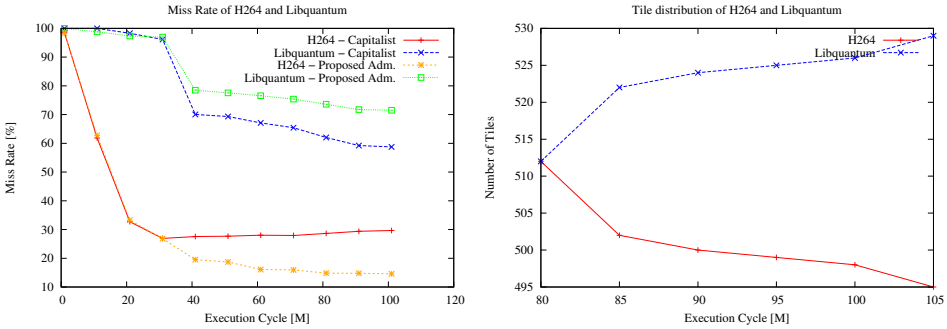
These three metrics proposed were compared with the capitalist management policy which is used in related and reference works.

4.3 Shared Cache Miss Rate

The first results to be analyzed correspond to the shared cache miss rate obtained using the dynamic management proposed and the capitalist management policy. Minimizing the amount of misses is sought as these entail very high penalties that produce the IPC decrease. For each test set assessed three figures are shown. The first figure corresponds to the miss rate in shared cache for each individual process, the following shows the distribution of tiles for each process performed

by the dynamic management proposed, while the last figure shows the miss rate of the global test set analyzed.

In the test set A, it can be observed the miss rate continuous decline up to the execution cycle 33M, from which it gets into a steady state. This is the result of the number of hits needed to stabilize the shared cache memory. These early misses can turn into compulsory misses, so it can be assumed that as from execution cycle 37M shared cache memory gets into an execution "regime".



(a) L3 Miss Rate (b) Tiles Distribution

Fig. 2: Test Set A.

After execution cycle 40M the heuristic developed here starts to differentiate from the capitalist one, by eliminating inter-process misses and giving space to each benchmark. When H264 benchmark reaches this point, it has the necessary space for data, does not have interferences with Libquantum and the miss rate decreases while in Libquantum space for data is diminished, available space is restricted and the miss rate increases.

Inter-process misses represent 34.4% of the total misses produced in shared cache under the Capitalist management and 0% under the administration proposed. The maximum global miss rate reduction achieved for this test set under the administration proposed is about 20%, and according to the information obtained after the simulation, Libquantum ended up occupying 51.6% of space and H264 the remaining 48.4% of the shared cache.

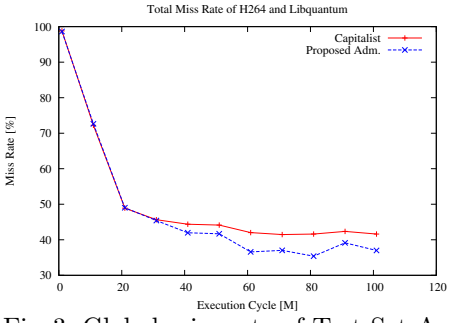
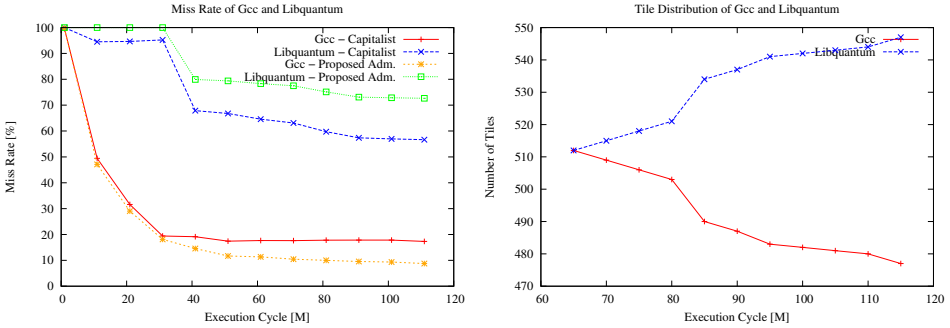


Fig. 3: Global miss rate of Test Set A.

Each tile includes two sets of simulated shared cache. This results in a size of 2KB per tile. Thus, it is concluded that the heuristic provided more space

to Libquantum benchmark, which has lower data locality for data than H264, but the latter was assigned the necessary space to achieve its optimal performance according to the metrics applied. Regarding the global miss rate for this test set, it can be observed that space restriction, when separated into disjoint tiles, made no difference to the miss rate in the first execution cycle (the heuristic did not perform reallocations), while when space managing process started, it caused a decrease in the global miss rate. Although H264 miss rate decreases and Libquantum increases, taken globally, the benefits of successfully applying the dynamic management policy to the shared cache memory can be seen.

The test set B analysis shows that a number of execution cycles similar to those of the test set A is required to get into a cache regime. For individual miss rates in each benchmark, it can be observed the same effect as in the test set A.



(a) L3 Miss Rate (b) Tiles Distribution

Fig. 4: Test Set B.

However, when compared to the achievements of implementing test set A, it is clear that a lower improvement was achieved in Gcc miss rate, and that the range of increase in the Libquantum rate was preserved. This is because Gcc has 67.2% more accesses to the shared cache than H264 does, resulting in inter-processes miss rates for this test set of 35.5% implemented under the capitalist management policy and 0.2% under the administration proposed.

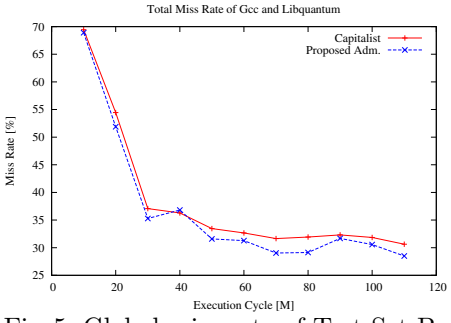


Fig. 5: Global miss rate of Test Set B.

The latter, ended providing less space to Gcc than to H264, since the former has higher data locality up to

the execution cycle 85M, whereas Libquantum has much lower data locality than the size of the shared cache studied, and all the same it fails to make a considerable difference, because the space is higher than the one obtained in the test set A. The maximum global miss rate reduction achieved for this test set using the administration proposed is about 11%.

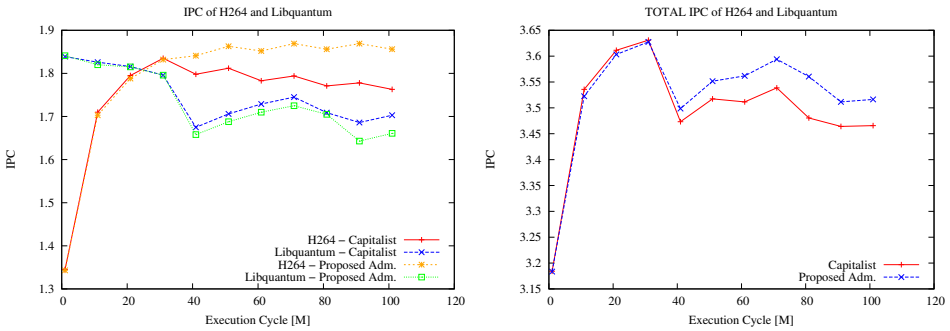
Tile allocation by dynamic management for this test set confirms what was concluded when analyzing individually each benchmark miss rate. Apart from efficiently allocating space in cache memories, space allocation rates managed to adapt quickly enough to changes in access rates of the test set. This can be observed when allotted space variation rate is higher than in test set A.

4.4 Instructions Per Cycles (IPC)

Comparing the IPC metric of an executed process under the management policy with the one under the Capitalist policy gives an idea of the increase in performance improvement. This increase can be calculated using the following expression:

$$PerformanceSpeedUp = \frac{IPC_{DynAdm.}}{IPC_{Cap.Adm.}} \quad (5)$$

The metric analysis for the test set A, indicates that the decrease in benchmark H264 miss rate is clearly reflected in its IPC increase, while for Libquantum, having a lower rate of memory hits, the increase in the miss rate does not affect its performance significantly. The maximum improvement achieved for benchmark H264 was 10%, while the maximum performance decrease for Libquantum was 3%.

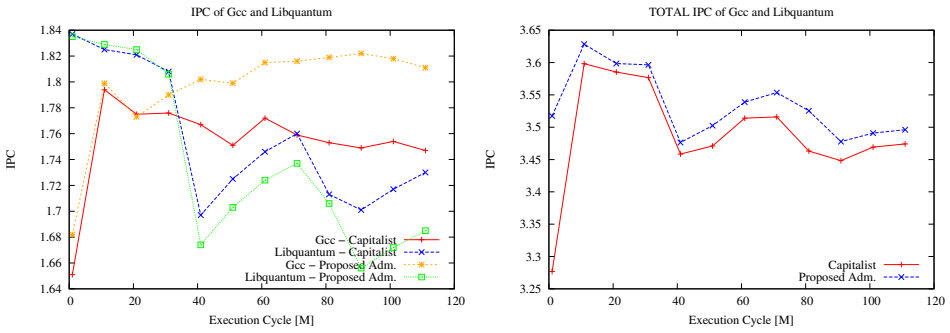


(a) IPC (b) Total IPC

Fig. 6: IPC of Test Set A.

The total performance for this test set shows that maximum improvement was achieved at 2.35%. This indicates that improving or worsening individual performance of a process in progress is not to the advantage of the overall performance. Finally, it is observed that the results obtained comply with the corollary stated in [19], which indicates that fairness improving in resource management, results in an increase of global performance in multi-process environments.

In the test set B, the Gcc benchmark is benefited by the proposed dynamic management since the rate of cache hits is higher than in H264. By efficiently managing the space allotted by each benchmark, varying it at the necessary speed to match the changes in the shared cache rate of hits, and also by eliminating inter-process misses with space distribution in tiles, it is possible to achieve the increase in the IPC metrics for Gcc. This is lower than that obtained by H264, because Gcc has lower data locality, but it is benefited by the removal of interferences produced by Libquantum. Regarding the latter, the decrease in IPC is 2.6%, being similar to that in the execution of the test set A. The analysis of the total IPC metric for this test set shows a considerable input when working in a shared cache memory regime. The maximum improvement achieved when implementing capitalist management policy was 1.80%.



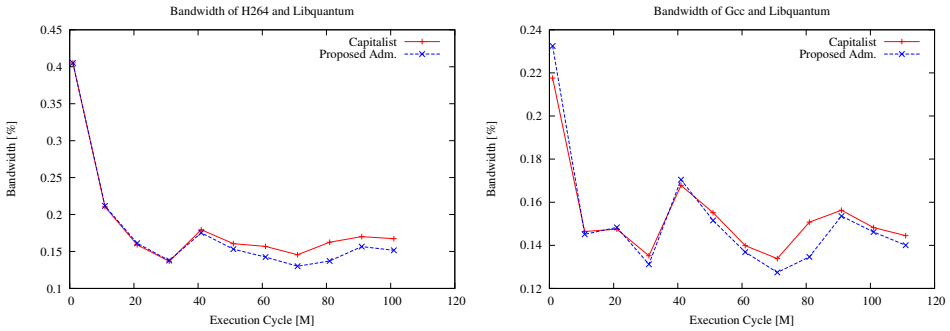
(a) IPC (b) Total IPC

Fig. 7: IPC of Test Set B.

4.5 Bandwidth Used

This metric provides a measure of usage of main memory by execution processes. By reducing the bandwidth used by processing cores, other devices may have access to it, thus, improving the general system performance. Decreasing miss rates in shared cache levels, implies less hits to the main memory (by the simu-

lated memory hierarchy), which should decrease the bandwidth used by the test sets, as the global miss rate of the shared cache is reduced significantly by the proposed administration, so it is reduced the bandwidth used. This is clearly seen in the following figures, showing the bandwidth of each test set when capitalist management policies and the policy developed in this paper are implemented. For test set A, the reduction of the bandwidth usage is about 20% and for the test set B is about 12%.



(a) BandWidth Usage of Test Set A (b) BandWidth Usage of Test Set B

Fig. 8: BandWidth Used.

Test set C deserves being analyzed soon afterwards. Gcc and H264 benchmarks showed no variation when applying the proposed dynamic management policy as regards the capitalist one. This result is due to the fact that both benchmarks did not present enough inter-process misses so as the proposed heuristics could prove a performance improvement when varying the allocated space to each process. When capitalist management policies were implemented, inter-process misses represented only 2.4% of shared cache misses, so that when implementing the policy developed, it assigned the same amount of tiles to each benchmark and made no tiles exchanges, resulting in a negligible variation of the miss rate, IPC and bandwidth metrics.

5 Conclusions

It was observed that the proposed policy has a lower miss rate in global shared cache for each test set studied. Certain processes increased their miss rates by space restrictions, so that other processes could make better use of that space, thus increasing the global IPC. Also, it was proved that a more efficient use of space in shared cache memory is achieved by eliminating inter-process misses

and adapting available space for each process in progress at a suitable speed according to the hit rate to shared cache memory. The maximum global miss rate reduction achieved for the test set A was 20% and for the test set B was 12%. The dynamic management heuristic implemented fulfills its objective of managing the shared cache space for every process while improving the overall IPC. The maximum global IPC achieved for the test set A was 2.35% and for the test set B was 1.8%.

Using the capitalist management policy, the test sets had no inter-process misses that would cause a high process overlap inside the shared cache, this was due to the fact that the memory hierarchy used counted with a level of resources higher than those normally used in related works, resulting in the decrease of such misses. This highlights the benefits of the metrics chosen to evaluate the space to be occupied by each process (the metrics provided by equations (3) and (4)), applied in the heuristic for dynamic management policy proposed.

Finally, the bandwidth consumed for each policy was analyzed and it is concluded that when making better management of space in shared cache memories, misses are reduced, and therefore the accesses to the main memory, thus diminishing bandwidth use, according to the reduction in the global miss rate.

6 Future Work

Future works will concentrate on evaluating dynamic management policies in working environments provided with more cores (more than 8), in order to analyze the behavior of the heuristic developed in over-exploited shared cache memories. Another aspect to be studied is the application of the heuristic to simultaneous multi-threading (SMT) environments and in multiple cores with multiple processing threads. This new analysis will imply updating the Pintool developed so as to detect different execution threads for each process. The process controller will not be modified but some processing modules will be added to classify information according to the thread and the process it belongs to. Another aspect to be explored is the Pseudo-LRU stack algorithm and its impact on the implementation of the metrics proposed, since it diminishes precision for miss classification.

7 Acknowledgments

This work was supported by the University of Buenos Aires and the National Council of Scientific and Technical Research (CONICET).

References

1. Pin - a dynamic binary instrumentation tool. `http://www.pintool.org`.
2. Spec cpu2006. `http://www.spec.org/cpu2006/`.
3. Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. *Proceedings of the 42nd annual Southeast regional conference on - ACM-SE 42*, page 267, 2004.
4. George Alm. Calculating stack distances efficiently. *ACM SIGPLAN Notices*, 38(2 supplement):37–43, February 2003.
5. D. Burger and S.W. Keckler. Exploring the design space of future CMPs. *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 199–210, 2001.
6. Claudio A Carballal, José Luis Hamkalo, and Bruno Cernuschi-Frías. A Modular Workflow to Dynamically Instrument and Treat Information in Multi-Process Environments. (ISSN 1850-2776):14, 2010.
7. Jonathan Chang, Ming Huang, Jonathan Shoemaker, John Benoit, Szu-liang Chen, Wei Chen, Siufu Chiu, Raghuraman Ganesan, Gloria Leong, Venkata Lukka, Stefan Rusu, and Durgesh Srivastava. The 65-nm 16-MB Shared On-Die L3 Cache for the Dual-Core Intel Xeon Processor 7100 Series. 2007.
8. J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.
9. Greg Hamerly and Brad Calder. Discovering And Exploiting Program Phases. *Ieee Micro*, pages 84–93, 2003.
10. José Luis Hamkalo and Bruno Cernuschi-Frías. A Taxonomy for Cache Memory Misses. *Proc. of the 11th Symposium on Computer Architecture and High Performance Computing*, pages 67–73, 1999.
11. J L Hennessy and D A Patterson. *Computer Architecture: A Quantitative Approach*, volume 3rd. Morgan Kaufmann, 2006.
12. Alan Jay Hill, Mark adn Smith. Evaluating Associativity in CPU Caches. *IEEE*, (vol 38 No 12), 1989.
13. Lisa R Hsu, Ann Arbor, and Steven Reinhardt. Communist , Utilitarian , and Capitalist Cache Policies on CMPs : Caches as a Shared Resource. 2006.
14. Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. 2004.
15. Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P Sadayappan. Enabling Software Management for Multicore Caches with a Lightweight Hardware Support. 2009.
16. Chi-keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Steven Wallace, Vijay Janapa, and Geoff Lowney. Pin : Building Customized Program Analysis Tools. 2005.
17. Ruud Van Der Pas and High Performance. Memory Hierarchy in Cache- Based Systems. 2002.
18. L Peng, J Peir, T Prakash, C Staelin, Y Chen, and D Koppelman. Memory hierarchy performance measurement of commercial dual-core desktop processors. *Journal of Systems Architecture*, 54(8):816–828, 2008.
19. G. Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical Cache Models with Application to Cache Partitioning. 2001.
20. Simon Tam, Stefan Rusu, Jonathan Chang, Sujal Vora, Brian Cherkauer, and David Ayers. A 65nm 95W Dual-Core Multi-Threaded Xeon Processor with L3 Cache. pages 15–18, 2006.

21. Carole-jean Wu and Margaret Martonosi. A Comparison of Capacity Management Schemes for Shared CMP Caches. *7th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD) in conjunction with ISCA-35*, 2008.
22. Xing Zhou, Wenguang Chen, and Weimin Zheng. Cache Sharing Management for Performance Fairness in Chip Multiprocessors. *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 384–393, September 2009.