# Explaining Dynamic Cache Partitioning Speed Ups

Miquel Moreto*, Francisco J. Cazorla†, Alex Ramirez*† and Mateo Valero*†
*Universitat Politecnica de Catalunya, Computer Architecture Department, 08034 Barcelona, Spain
HiPEAC. European Network of Excellence on High-Performance Embedded Architecture and Compilation
Email: {mmoreto, aramirez, mateo}@ac.upc.edu
†Barcelona Supercomputing Center, 08034 Barcelona, Spain
Email: francisco.cazorla@bsc.es

***Abstract*— Cache Partitioning has been proposed as an interesting alternative to traditional eviction policies of shared cache levels in modern CMP architectures: throughput is improved at the expense of a reasonable cost. However, these new policies present different behaviors depending on the applications that are running in the architecture. In this paper, we introduce some metrics that characterize applications and allow us to give a clear and simple model to explain final throughput speed ups.**

## I. Introduction

Recent technology advances have increased the number of available transistors for processor designers. However, the performance achievable by traditional superscalar processor designs has almost saturated due to the limitation imposed by instruction-level parallelism. As a consequence, thread-level parallelism has become a common strategy for improving processor performance. This strategy has led to a wide range of multithreaded processor architectures, such as simultaneous multithreading (SMT) and chip multiprocessing (CMP).

These architecture paradigms offer the opportunity to obtain higher throughputs, but they also have to face the challenge of sharing resources of the architecture. Simply avoiding any resource control can lead to undesired situations where one thread is monopolizing all the resources and harming the other threads performance. Some studies deal with the resource sharing problem in SMTs at core level resources [1] like issue queues, registers, etc. In CMPs, resource sharing is lower than in SMT, focusing in the cache hierarchy.

Some applications present low reuse of their data and pollute caches with data streams, such as multimedia, communications or streaming applications, or have many compulsory misses that cannot be solved by assigning more cache space to the application. Traditional eviction policies such as Least Recently Used (LRU), pseudo LRU or random are demand-driven, that is, they tend to give more space to the application that has more accesses to the cache hierarchy. Some previous work propose static and dynamic partitioning algorithms that monitor the L2 cache accesses and decide a partition for a fixed amount of cycles in order to maximize throughput [3], [5], [8] or fairness [4].

In [3], [4], [5], [8] the main tool used to decide cache partitions is *Stack Distance Profiling*. Each set in a cache can be seen as a LRU stack, where lines are sorted by their last access cycle. In that way, the first line of the LRU stack is the most recently used (MRU) line while the last line is the LRU line. For a $K$-way associative cache with LRU replacement algorithm, we need $K+1$ counters: $C_1, C_2, \ldots, C_K, C_{>K}$. On each cache access, one of the counters is incremented. If it is a cache access to a line in the $i^{th}$ position in the LRU stack of the set, $C_i$ is incremented. If it is a cache miss, the line is not found in the LRU stack and, as a result, we increment the miss counter $C_{>K}$. Stack distance profile can be obtained during execution. A characteristic of these profiles is that the number of cache misses for a smaller cache with the same number of sets can be easily computed using the stack distance profile. For example, for a $K'$-way associative cache, where $K' < K$, the new number of misses can be computed as:

$$misses = C_{>K} + \sum_{i=K'+1}^{K} C_i$$

Using the stack distance histogram of two applications, we can derive the optimal L2 cache partition that would minimize the total number of misses, as this last number corresponds to the sum of misses of each thread with the assigned number of ways. This mechanism is used in [3], [5], [8] in order to minimize the total number of misses and try to maximize throughput. Throughout this paper, we will call this policy as *MinMisses*.

These studies demonstrate that *MinMisses* is really useful in some cases and, in average, gives interesting speed ups. However, they are mainly interested in the practical implementation of these techniques, which is necessary and challenging, but do not give a clear model to explain in which situations these algorithms are more profitable. Obtaining such a model provides a powerful tool to predict performance benefits of these proposals for any application, without the restriction of using a particular set of benchmarks. Hence, in a scenario where the set of applications is known, computer designers are interested in analyzing if the extra cost that cache partitioning supposes is worthwhile. Our main contribution is a clear guideline that explains the performance benefits of *MinMisses* over LRU. With this objective, we introduce a reduced number of metrics that characterize the behavior of benchmarks and that are obtained by individual simulation of each benchmark.

## II. Experimental Setup

We have targeted this study to the case of a CMP with two cores with their respective own data and instruction L1 caches

Posted to IEEE & CSDL on 4/3/2007
DOI 10.1109/L-CA.2007.3
1556-6056/07/$25.00 © 2007 Published by the IEEE Computer Society

and unified L2 cache shared among threads as in [4], [5], [8]. In this situation, the effects of the partitioning algorithm can be analyzed easier as there is no collision with effects concerning other shared resources as in the case of a SMT. Each core is single threaded and can fetch up to 8 instructions each cycle. It has 6 integer (I), 3 floating point (FP), and 4 load/store functional units and 32-entry I, load/store, and FP instruction queues. Each thread has its own 256-entry reorder buffer and 256 physical registers. We use a two-level cache hierarchy with 64B lines with separate 16KB, 4-way associative data and instruction caches, and a unified 1MB, 16-way L2 cache that is shared among all cores. Latency from L1 to L2 is 15 cycles, and from L2 to memory 300 cycles.

We extended the SMTSim simulator [9] to make it CMP. We collected traces of the most representative 300 million instruction segment of each program, following the SimPoint methodology [7]. We use the FAME simulation methodology proposed in [10] with a Maximum Allowable IPC Variance of $5\%$. This evaluation methodology measures the performance of multithreaded processors by reexecuting all threads in a multithreaded workload until all of them are fairly represented in the final IPC taken from the workload. As performance metrics we have used IPC throughput and the sum of individual IPCs to represent pairings final throughput. Average throughput is computed using the harmonic mean of final throughputs.

## III. Explaining Speed Ups

In [5] authors qualitatively classify the benchmarks based on the shape of the stack histogram, which determines the behavior of the application as more ways are assigned to it. This classification extends previous classifications in [3], [8]. In order to reproduce this experiment, we simulate each SPEC2K benchmark in isolation in our baseline CMP architecture. We observed that the performance of each benchmark varies as we increase the number of ways given to it. As shown in Figure 1, there are three different cases. Low utility (L) benchmarks are not affected by L2 cache space because nearly all L2 accesses are misses. Other benchmarks just need some ways to have maximum throughput as they fit in the L2 cache, that we call *Small Working Set* (S). Finally, *High utility* (H) benchmarks always improve their performance as we increase the number of ways given to them. Clear representatives of these three groups are `applu` (L), `gzip` (S) and `ammp` (H) in Figure 1. Thus, each 2-thread workloads can be classified into one of these 6 groups: HH, HL, HS, LL, LS, and SS.

We composed a total of $48$ workloads, 8 in each group, in which every benchmark appears between 3 and 5 times, which means that results are not biased by the behavior of any benchmark. The performance improvement of *MinMisses* over LRU is different for each group of workloads. For the workload types HH, HS and SS *MinMisses* has moderate losses, lower than $3\%$ [1]. In contrast, *MinMisses* improves LRU for LL and LS groups, presenting moderate gains lower than $3\%$. HL is the group with the best results, with average improvements of $27.3\%$. This performance results are consistent with results obtained by other authors. Although in [4], [5] signinficant performance benefits were presented (*MinMisses* harmonic mean had a $16.8\%$ and $14\%$ improvement over LRU, respectively), here we obtain an average improvement of $5.01\%$. The explanation for this significant difference is that the criterion to select pairings in these papers is different and many of their combinations belong to the HL group.

[1]We consider that simulations inside this $3\%$ threshold have similar performance to the LRU policy.

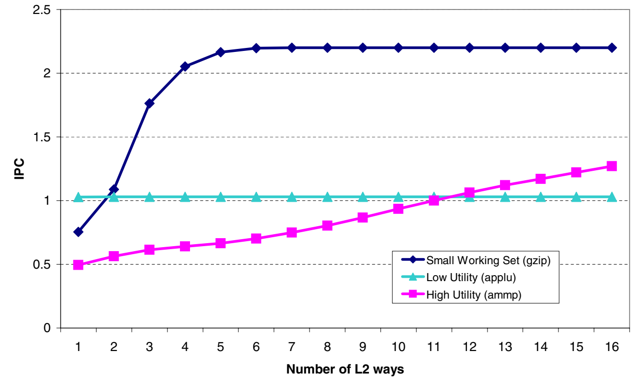


Fig. 1. Performance of benchmarks as we vary the number of ways.

The main motivation for this paper is that there is a high variability in the improvement that *MinMisses* obtains over LRU inside each group. Hence, it is necessary to find a new classification for the workloads so that the performance improvement of *MinMisses* over LRU in each group is consistent with some rules. With that purpose we introduce two metrics.

**Metric 1**. The $w_{P\%}(B)$ metric measures the number of ways needed by a benchmark $B$ to obtain at least a given percentage $P\%$ of its maximum IPC (when it uses all L2 ways). We have found that using a value of $P = 90\%$ as threashold gives a metric that accurately corresponds to the intuitive classification that we have previously introduced. In that way, we have the following benchmark classification depending on the value of $w_{90\%}$. In Table I we can see $w_{90\%}$ for all SPEC2K benchmarks. Just note that H benchmarks have $w_{90\%} > 8$, S benchmarks have $2 < w_{90\%} \leq 8$ and L have $1 \leq w_{90\%} \leq 2$.

- *High Utility*: ammp, apsi, art, facerec, galgel, mgrid, parser, twolf and vpr.
- *Small Working Set*: crafty, eon, gcc, gzip, perl and vortex.
- *Low Utility*: applu, bzip2, equake, gap, lucas, mcf, mesa, sixtrac, swim and wupwise.

**Metric 2**. The $w_{LRU}(th_i)$ metric measures the number of ways given by LRU to each thread $th_i$ both benchmarks run together. This can be done simulating all benchmarks alone and using the total number of L2 accesses in a fixed period of cycles for each benchmark. We denote the number of Accesses in a Period of 100 Thousand Cycles for thread $i$ to the L2 cache as $APTC_i$ (see Table I). Other authors have used this approximation and proved its accuracy [2].

$$w_{LRU}(th_i) = Associativity \cdot \frac{APTC_i}{APTC_0 + APTC_1}$$

TABLE I
BENCHMARK CHARACTERIZATION

| Bench | $w_{90\%}$ | APTC | IPC | Bench | $w_{90\%}$ | APTC | IPC |
|---|---|---|---|---|---|---|---|
| ammp | 14 | 23.63 | 1.27 | applu | 1 | 16.83 | 1.03 |
| apsi | 10 | 21.14 | 2.17 | art | 10 | 46.04 | 0.52 |
| bzip2 | 1 | 1.18 | 2.62 | crafty | 4 | 7.66 | 1.71 |
| eon | 3 | 7.09 | 2.31 | equake | 1 | 18.6 | 0.27 |
| facerec | 11 | 10.96 | 1.16 | fma3d | 9 | 15.1 | 0.11 |
| galgel | 15 | 18.9 | 1.14 | gap | 1 | 2.68 | 0.96 |
| gcc | 3 | 6.97 | 1.64 | gzip | 4 | 21.5 | 2.20 |
| lucas | 1 | 7.60 | 0.35 | mcf | 1 | 9.12 | 0.06 |
| mesa | 2 | 3.98 | 3.04 | mgrid | 11 | 9.52 | 0.71 |
| parser | 11 | 9.09 | 0.89 | perl | 5 | 3.82 | 2.68 |
| sixtrack | 1 | 1.34 | 2.02 | swim | 1 | 28.0 | 0.40 |
| twolf | 15 | 12.0 | 0.81 | vortex | 7 | 9.65 | 1.35 |
| vpr | 14 | 11.9 | 0.97 | wupw | 1 | 5.99 | 1.32 |

Next, we explain the classification of the 48 pairings of benchmarks using these two metrics as well as the rationale behind this classification.

- We study the cases where $w_{90\%}(th_0) + w_{90\%}(th_1) \leq Associativity$. In this case, we can assure that with an optimal static partitioning of the L2 cache we can obtain at least 90% of each benchmark throughput. To better understand our results, we have split this case in two possible situations.
  - When $w_{90\%}(th_i) < w_{LRU}(th_i)$ for bothe treads. We call this situation *Case 1*.
  - When $w_{90\%}(th_A) > w_{LRU}(th_A)$ and $w_{90\%}(th_B) < w_{LRU}(th_B)$. We call this situation *Case 2*.
- When $w_{90\%}(th_0)+w_{90\%}(th_1) > Associativity$. We call this situation *Case 3*.

**Case 1**. When $w_{90\%}(th_i) < w_{LRU}(th_i)$ for both threads. Theoretically, in this situation LRU attains at least 90% of each benchmark IPC. Thus, it is intuitive that *MinMisses* should obtain similar results to LRU policy. We have seen that 16 out of the 19 benchmarks belonging to this subgroup present performance improvement between $+3\%$ and $-3\%$. In the case of `gzip+mesa` and `swim+eon`, *MinMisses* obtains a slightly better result as LRU assigns too few ways to `mesa` and `eon`, respectively. The case of `apsi+crafty` shows worse results, as it looses $5,4\%$ in comparison to LRU. For this pairing, *MinMisses* is unable to assign 4 ways to `crafty` and, as a consequence, performance drops.

**Case 2**. When $w_{90\%}(th_A) > w_{LRU}(th_A)$ and $w_{90\%}(th_B) < w_{LRU}(th_B)$. In this situation, LRU is harming the performance of thread $A$, because it gives more ways than necessary to thread $B$. Thus, in this situation LRU is assigning some shared resources to a thread that does not need them, while the other thread could benefit from these resources. This is confirmed by simulations, as in this situation we obtain an average improvement of $11,7\%$.

Analyzing all the 15 pairings belonging to this group, we have noticed that normally the higher the difference $w_{90\%}(th_A) - w_{LRU}(th_A)$, the higher performance benefits. This behavior can be seen in Figure 3, where we represent values of speed up depending on the value of $w_{90\%}(th_A) - w_{LRU}(th_A)$. Just to illustrate that these values have a linear relationship, we have estimated the corresponding linear regression. We can see that this estimation fits well the cloud of points. In fact, we have observed that when the harmed thread is the thread with higher IPC, then performance speed ups are higher than expected. Conversely, if the harmed threads are the ones with lower IPC, then performance gains are lower than expected. This reasoning explains the behavior of points that are far away from the linear regression. The clearest one is `twolf+mcf`, as the speed up attains 1.59 because LRU harmed `twolf`, which has an IPC 12 times higher than `mcf`.

Using the linear regression, we can estimate a threshold of $w_{90\%}(th_A) - w_{LRU}(th_A)$ to obtain performance improvements higher than 3%. In fact, this point corresponds to $w_{90\%}(th_A) - w_{LRU}(th_A) = 1.57$. Thus, in situations where this threshold is exceeded, performance gains are expected to be high enough to consider this method. An other interesting point is to note that the correlation factor $r$ of this linear regression is 0.54. Values near 1 mean that the linear approximation is accurate, while values near 0 mean that the approximation is erroneous. In this case, we can apply an independence test [6] to know the probability that the observed relation is generated at random with a confidence factor of 95% and we obtain that this hypothesis has really low probability ($\approx 0.0004$).

It is also remarkable to note that points near to the boundary between the first and second cases show similar

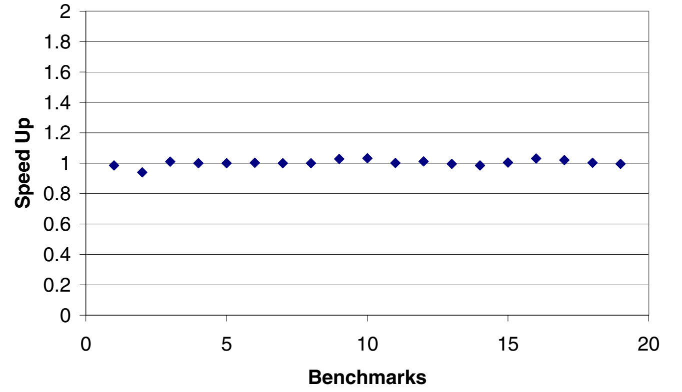


Fig. 2. IPC speed up when $w_{90\%}(th_i) < w_{LRU}(th_i)$.

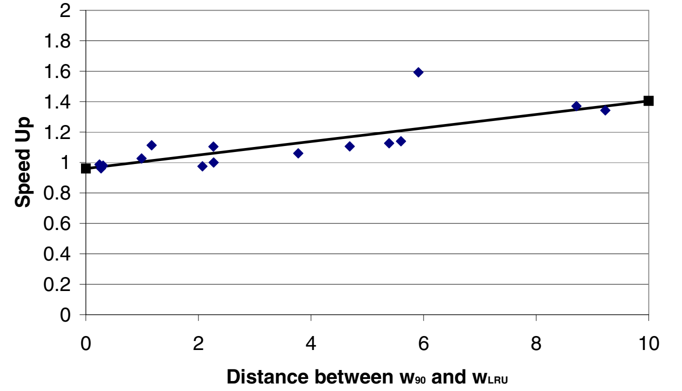


Fig. 3. IPC speed up when $w_{90\%}(th_A) > w_{LRU}(th_A)$ and $w_{90\%}(th_B) < w_{LRU}(th_B)$.

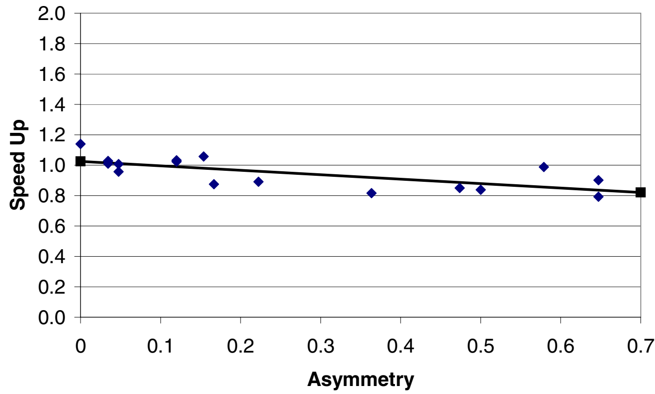


Fig. 4. IPC speed up when $w_{90\%}(th_0) + w_{90\%}(th_1) > Associativity$.

results. Finally, note that no other situations can occur when $w_{90\%}(th_0) + w_{90\%}(th_1) \leq Associativity$.

**Case 3**. When $w_{90\%}(th_0) + w_{90\%}(th_1) > Associativity$. In this situation, our L2 cache configuration is not big enough to assure that both benchmarks will have at least a $90\%$ of their peak performance. Here, simulation results present higher variability. Five out of the 14 pairings belonging to this group present performance improvements higher than $3\%$, while just one pairing has improvement between $+3\%$ and $-3\%$ and, finally, eight benchmarks present improvements lower than $-3\%$, with an average loss of $7.1\%$. We have observed that pairings belonging to this group show worse results when the value of $|w_{90\%}(th_0) - w_{90\%}(th_1)|$ grows. In fact, we have shown in Figure 4 the relation between speed ups and the value of $\frac{|w_{90\%}(th_0) - w_{90\%}(th_1)|}{w_{90\%}(th_0) + w_{90\%}(th_1)}$. Higher values of asymmetry lead to higher performance losses. In this case, we have a thread that requires much less L2 cache space than the other to attain $90\%$ of its peak IPC. LRU treats threads equally and manages to satisfy the less demanding thread necessities. In case of *MinMisses*, it assumes that all misses are equally important for throughput and tends to give more space to the thread with higher L2 cache necessity, while harming the less demanding thread.

It is interesting to note that when both benchmarks need a great part of the L2 cache to attain their peak IPC ($w_{90\%} \approx Associativity$), *MinMisses* presents higher speed ups. In these situations, LRU policy suffers from high inter-thread cache misses as both benchmarks have many accesses with high stack distance. *MinMisses* prevents a percentage of these inter-thread misses and, as a consequence, presents performance benefits higher than $3\%$. This intuition is consistent with Figure 4 as the asymmetry value is near to zero.

In this case, correlation between the cloud of points and the linear regression is not as accurate as in *case 2*, as we obtain a correlation factor of $0.46$, but it continues to denote a clear trend. We have also applied an independence test, obtaining as in the previous case a very low probability ($\approx 0.007$).

*Example:* Just to illustrate how to use the explained model, we can analyze the case of `vpr+equake`, which is not in the selected 48 pairings. We have $w_{90\%}(vpr) = 14$, $w_{90\%}(equake) = 1$, $APTC_{vpr} = 11.9$ and $APTC_{equake} = 18.6$. Thus, we have $w_{LRU}(vpr) = 16 \cdot \frac{11.9}{11.9+18.6} = 6.24$, $w_{LRU}(equake) = 9.76$. Here `equake` has $8.76$ more ways than necessary, while `vpr` is lacking of $7.76$ ways. Then, using the linear regression of *case 2*, we obtain

$$Predicted\ Speed\ Up = 0.044 \cdot 8.76 + 0.96 = 1.35$$

which is very close to the simulated speed up of $1.38$.

## IV. Conclusions

Throughout this paper we have characterized the behavior of final IPC speed ups when a dynamic partitioning algorithm such as *MinMisses* is used in a shared L2 cache. To succeed in that goal, we have used two metrics, $w_{90\%}$ and $w_{LRU}$, together with individual IPCs. In that way, we have detected one situation where this policy shows similar results to traditional LRU (Case 1), one situation where significant speed ups are obtained (Case 2) and a third situation where *MinMisses* obtains performance losses (Case 3).

Thus, it is really important to know exactly which kind of applications will be run in the architecture in order to correctly decide if such a dynamic partitioning mechanism is appropriate. In order to avoid the situation where losses are obtained, we have two possible options. First, we could let the Operating System decide between LRU and *MinMisses* depending on which applications are being run or, second, try to improve the *MinMisses* decision algorithm by giving weight to misses. These options should be explored in future work.

## Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN-2004-07739-C02-01, and grant AP-2005-3318. The authors would like to thank Jim Smith for his technical comments in the development of this paper. Authors would also thank to Carmelo Acosta, Ayose Falcon, Daniel Ortega, Jeroen Vermoulen and Oliverio J. Santana for their work in the simulation tool.

## References

[1] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dynamically controlled resource allocation in SMT processors. In *37th MICRO*, 2004.
[2] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In $11^{th}$ *HPCA*, 2005.
[3] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *Design Automation Conference*, 2000.
[4] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In $13^{th}$ *PACT*, 2004.
[5] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In $39^{th}$ *MICRO*, 2006.
[6] V. Rohatgi. *An Introduction to Probability Theory and Mathematical Statistics*. John Wiley and Sons, 1976.
[7] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 2003.
[8] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1):7–26, 2004.
[9] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In $22^{nd}$ *ISCA*, 1995.
[10] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. A novel evaluation methodology to obtain fair measurements in multithreaded architectures. In *MoBS*, 2006.