Online Prediction of Applications Cache Utility

Miquel Moretó*, Francisco J. Cazorla†, Alex Ramirez*† and Mateo Valero*†
*Universitat Politècnica de Catalunya, Departament d'Arquitectura de Computadors, Barcelona, Spain
HiPEAC. European Network of Excellence on High-Performance Embedded Architecture and Compilation
†Barcelona Supercomputing Center - Centro Nacional de Supercomputación, Barcelona, Spain
Emails: {mmoreto, aramirez, mateo}@ac.upc.edu, francisco.cazorla@bsc.es

***Abstract*— General purpose architectures are designed to offer average high performance regardless of the particular application that is being run. Performance and power inefficiencies appear as a consequence for some programs. Reconfigurable hardware (cache hierarchy, branch predictor, execution units, bandwidth, etc.) has been proposed to overcome these inefficiencies by dynamically adapting the architecture to the application needs. However, nearly all the proposals use indirect measures or heuristics of performance to decide new configurations, what may lead to inefficiencies.**

In this paper we propose a runtime mechanism that allows to predict the throughput of an application on an architecture using a reconfigurable L2 cache. L2 cache size varies at a way granularity and we predict the performance of the same application on all other L2 cache sizes at the same time. We obtain for different L2 cache sizes an average error of 3.11%, a maximum error of 16.4% and standard deviation of 3.7%. No profiling or Operating System participation is needed in this mechanism. We also give a hardware implementation that allows to reduce the hardware cost under 0.4% of the total L2 size and maintains high accuracy. This mechanism can be used to reduce power consumption in single threaded architectures and improve performance in multithreaded architectures that dynamically partition shared L2 caches.

I. Introduction

The limitation imposed by instruction-level parallelism has motivated the appearance of thread-level parallelism (TLP) as a common strategy for improving processor performance. TLP paradigms such as simultaneous multithreading (SMT) [1], [2], chip multiprocessing (CMP) [3] and combinations of both offer the opportunity to obtain higher throughput, but they also face the challenge of sharing architecture resources. Some studies deal with the resource sharing problem in SMTs at the core resources level [4] like issue queues, registers, etc. In CMPs, resource sharing is lower than in SMT, focusing in the cache hierarchy. Several mechanisms have been proposed to dynamically split the L2 cache in a CMP architecture in order to maximize throughput or fairness [5]–[9].

However, the problem of adapting resources to program needs is not only a problem of multithreaded architectures. Several mechanisms have been proposed in superscalar architectures to use reconfigurable hardware that adapts microarchitecture features when the characteristics of the program change [10]–[14]. The common problem with all these self-tuning techniques is that decisions are based on indirect performance metrics or empirical heuristics.

In this paper we focus on dynamic configuration of the cache hierarchy. In particular, we propose a mechanism that allows to accurately predict the Instruction Per Cycle (IPC) of an application as we vary the amount of cache we devote to it. We vary L2 cache size by activating/deactivating some ways of a set associative cache. Our mechanism combines *Stack Distance Histograms* [15] and ananalytical model for predicting processor IPC introduced in [16]. We also have improved the model [16] in order to increase our IPC prediction accuracy. On average over all SPEC CPU 2000 benchmarks, our mechanism obtains an average error of 3.11%, with a maximum error of 16.4% (`twolf`) and a standard deviation of 3.7%. The ability to predict IPC as we change the cache configuration can be applied in two different scenarios:

Cache sharing in MT architectures. A better sharing of the L2 cache among the running threads can be obtained. Previous work on this topic proposes static and dynamic partitioning of a shared L2 cache in CMP/SMT architectures in order to maximize throughput or fairness [5]–[9]. These proposals use indirect metrics of throughput such as total number of misses, or data reuse. The mechanism we propose provides a direct estimation of performance for different cache configurations, which is the appropriate metric to maximize total throughput.

Power reduction. A better reduction in cache energy dissipation can be obtained by adjusting the hardware resources. By having a direct estimate on the performance of the application, it is possible to obtain the desired trade-off between power consunmption and performance. Previous work consisted in statically switching on or off L2 ways [13] or switching off lines after a number of cycles without being accessed [14]. However, these proposals cannot bound performance losses and rely on empirical heuristics. Giving the real contribution of each way to the final IPC can be used to bound performance losses while saving power.

In both areas, previous work relies on empirical heuristics and thresholds to make decisions. To our knowledge, we are the first to mix runtime measurements with analytical models to dynamically predict the actual performance impact of such decisions. The main contributions of this work are:

1) A runtime mechanism to predict IPC for different cache configurations with high accuracy. This proposal can help to reconfigure L2 caches in CMP/SMT scenarios for dynamic cache partitioning and in single core scenarios to reduce power. The important difference with previous work in these areas is that new configurations are based on real measures of throughput instead of indirect measures or adhoc heuristics.

1-4244-1058-4/07/$25.00 ©2007 IEEE

2) An modified version of the memory model in [16] that allows to better predict the cost of an L2 miss.

3) A sampling technique to reduce hardware cost under 0.4% of the total L2 size without excessive accuracy loss (the average error raises to 4%).

4) A systematic benchmark classification so that results are consistent in every benchmark group. This classification extends previous intuitive classifications that were obtained by hand. [1]

The rest of this paper is structured as follows. In Section II we introduce the methods that we employ to predict IPC curves and in Section III we show how to combine them to obtain IPC predictions. Next, in Section IV we describe the experimental environment and in Section V we discuss simulation results. In Section VI we deal with a practical implementation in hardware as well as a sampling technique to reduce this cost. In Section VII we report related work and, finally, we conclude with Section VIII.

II. Basis of IPC curves prediction

We define the IPC curve of an application as the set of IPC values that the application obtains for different configurations of the L2 cache. These configurations have a *way* granularity. Thus, in a K-way L2 cache, IPC curves have exactly K points (as we assume that at least one way is assigned to the application). In order to accurately predict IPC curves, we have combined two instruments: the stack distance histogram (SDH) [15], and an analytical model for superscalar processors performance [16].

Stack Distance Histogram. In [15] the concept of stack distance is introduced to study the behavior of storage hierarchies. Common eviction policies such as Least Recently Used (LRU) have the *stack property*. Thus, each set in a cache can be seen as an 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. As an example, we can see in Table I a stream of accesses to the same set. In this situation, cache lines *A*, *B* and *C* have stack distances 3, 4 and 1 respectively.

TABLE I : Stream of accesses to given cache set

Cache Line	A	B	C	C	A	D	B
Number of Reference	1	2	3	4	5	6	7

To build SDHs 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 hit to a line in the ith 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 $C_{>K}$. SDHs can be obtained during execution by running the thread alone in the system [5] or by adding some hardware counters that profile this information [6], [7]. A characteristic of these histograms is that the number of cache misses for a smaller cache with the same number of sets can be easily computed. For example, for a K'-way associative cache, where $K' < K$, the new number of misses can be computed as:

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

As an example, in Table II we show a SDH for a set with 4 ways. In the example, 5 accesses would have missed in the cache. However, if we had reduced the number of ways to 2 (keeping the number of sets constant), we would have experienced 20 misses (5 + 5 + 10).

TABLE II : SDH example

Stack Distance	1	2	3	4	>4
Number of Accesses	60	20	10	5	5

Superscalar Processors Analytical Modeling. In [16], a model that estimates performance for a superscalar processor is proposed. This model computes an ideal Cycle Per Instruction (CPI) when no misses occur and adds CPI penalties for each type of hazard, including branch mispredictions, instruction cache misses and data cache misses. Some assumptions and simplifications are done in this model, but simulations prove that it is accurate enough. Comparing to detailed simulation, errors are within 5.8% on average and within 13% in the worst case.

We use this analytical model to predict the performance of superscalar processors as we vary the L2 cache size. In our scenario, we can assume that ideal CPI is independent of the cache configuration (it only depends on data dependencies of the particular application). We also assume that the branch miss penalty remains constant for different cache sizes. Thus, we are only interested in using the part of the model that concerns the cache hierarchy. The model considers L2 instruction and data misses separately.

1) Instruction Misses: Initially, the processor issues instructions at the steady-state IPC. When an instruction L2 miss occurs, instructions in the issue queue and the front-end pipeline maintain issue rate for some cycles, but when the issue queue drains, issue-rate drops to zero following a linear descend [17]. After a miss delay, ΔI, instructions are delivered from main memory. Then, IPC ramps up to its steady-state value. In [16], it is shown that lost cycles until steady-state IPC is attained compensate useful cycles until the issue queue drains. Thus, the penalty for an isolated instruction cache misses is approximately equal to the main memory latency. Furthermore, as instruction cache misses must be serialised, each miss in a burst of consecutive instruction cache misses has the same penalty as an isolated one: ΔI cycles.

2) Data Misses: The basic difference between instruction and data cache misses is that instruction fetch and issue continue after the data cache miss, and so several data misses can occur in parallel. After ΔD cycles, data is delivered from main memory. In [16] it is shown that the misspenalty for an isolated data L2 miss can be approximated by ΔD.

[1]This classification is not used in IPC predictions.

When we have a burst of n L2 cache data misses, they will overlap if they all fit in the reorder buffer (ROB). In this situation, the misspenalty of ΔD cycles is shared among all misses. Then, being M_D the total number of L2 data misses and N_i the number of times that we have a burst of i misses that fit in the ROB, we can approximate the *average L2 data misspenalty per miss* (DCM) with the following formula:

$$DCM = \frac{1}{M_D} \cdot \sum_{i=1}^{ROBsize} N_i \cdot \Delta D \quad (2)$$

III. Prediction of IPC curves

In this section, we detail how to obtain an Online Prediction of Applications Cache Utility. We call this methodology OPACU. We use SDHs to compute the number of misses for each possible L2 cache size, and the memory model described in the previous Section to determine the miss penalty of each L2 miss.

OPACU Methodology. In our baseline configuration, the L2 cache has a variable number of active ways. We start by assigning w ways and computing the throughput of an application during C cycles. We denote this value as $IPC_{real,w}$, with $IPC_{real,w} = \frac{I}{C}$, where I is the number of committed instructions in this period. This IPC value is valid for the given number of ways that are being used.

Thanks to the SDH, we know whether an access would be a miss or a hit with a different number of ways $w' \in [1, K]$. Independently of the number of active ways, we store the LRU counters of the last K accesses of the thread and obtain the SDH for the whole K-way associativity L2 cache, as we explain in Section VI. On the other hand using the analytical model explained in Section II, we can estimate at runtime the number of times that we have a burst of i L2 data misses with an L2 cache with w' ways. We denote this number as $N_i^{w'}$ for $1 \leq i \leq ROBsize$ for any possible number of ways w', and the total number of instruction and data L2 misses as $M_I^{w'}$ and $M_D^{w'}$ respectively. These numbers are obtained at runtime using the hardware explained in Section VI. Using $N_i^{w'}$, we can compute the L2 data misspenalty when we use w' ways, $DCM^{w'}$, with Formula 2. Thus, we can estimate the variation in the *total misspenalty* for the new configuration due to data and instruction L2 misses, $\Delta C_D^{w,w'}$ and $\Delta C_I^{w,w'}$ respectively.

$$\Delta C_D^{w,w'} = DCM^{w'} \cdot M_D^{w'} - DCM^{w} \cdot M_D^{w} \quad (3)$$

$$\Delta C_I^{w,w'} = (M_I^{w'} - M_I^{w}) \cdot \Delta I \quad (4)$$

The value $\Delta C^{w,w'} = \Delta C_I^{w,w'} + \Delta C_D^{w,w'}$ may be positive or negative depending on the value of w and w'. Thus, we can predict the IPC when using w' ways, denoted as $IPC_{pred,w'}$.

$$IPC_{pred,w'} = \frac{I}{C + \Delta C^{w,w'}} \quad (5)$$

To illustrate this technique, we have chosen the `vortex` benchmark from SPEC CPU 2000 suite. In Table III we can see the different values when using a cache configuration with 16 ways, with just 7 active ways. We have an $IPC_{real,7}$ of 1.52 instructions per cycle. Using the previous formula, we predict the throughput for a configuration with 16 ways as 1.637, which is very close to the real value of 1.647. The relative error of this prediction is 0.65%.

TABLE III : IPC PREDICTION FOR `VORTEX`

I	C	w	w'	$\Delta C^{w,w'}$	ΔD
272M	178M	7	16	-12.5M	250

Improving the Memory Model. We have modified the memory model in [16] to increase accuracy in our predictions. The original model considers that two L2 data misses overlap if their ROB distance (in number of instructions) is less than the ROB size. One of the main assumptions of the model is that the ROB fills after an L2 data cache miss. However, this is not always true. To illustrate this point, we have measured the average ROB occupancy after an L2 data miss is serviced from main memory and commits for a ROB with 256 entries. This value varies depending on the application and is always less than the size of the ROB as it can be seen in Figure 1. We consider that this value remains constant when varying L2 cache size. This approximation works better than using the ROB size as a fixed value.

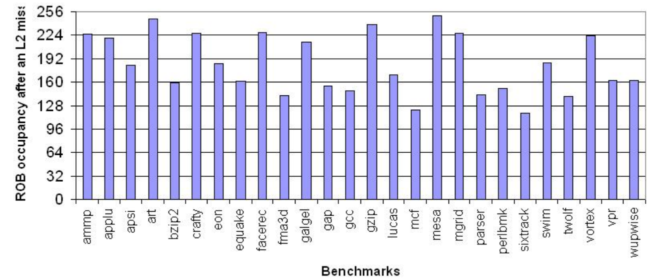


Fig. 1. Average ROB occupancy after an L2 miss commits

The actual problem is that the ROB is not always the bottleneck for performance. Sometimes issue queues are full with dependent instructions on the missing load, causing fetch and issue to stall. Thus, it is more representative to use average ROB occupancy after an L2 miss to determine if two L2 misses overlap. This value is easily obtained at runtime with a hardware counter. This intuition is confirmed by simulations in Section V. This improvement decreases our average error from 3.34% and maximum error of 20.9% (with the memory model from [16]) to an average error of 3.11% and maximum error of 16.4%.

Benchmark Characterization Using IPC Curves. We have extended previous classifications of benchmarks [5]–[7] based on the shape of the IPC curves. We have used this classification to analyze our simulations results. We simulate each SPEC CPU 2000 benchmark in our baseline architecture (see Section IV) which has a 1MB L2 cache 16-way associativity. We observe that the performance of each benchmark varies as we increase the number of ways given to it. As shown in Figure 2, 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 2.

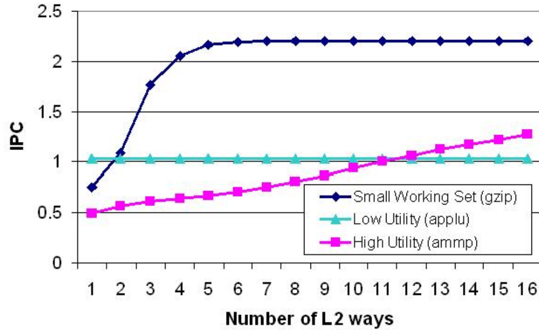


Fig. 2. IPC of benchmarks as we vary L2 cache size

In [6], this classification was done by hand. Here we propose a new metric to systematically classify benchmarks in these three groups.

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 Table IV we can see all SPEC CPU 2000 benchmarks classified depending on the value of $w_{90\%}$. Just note that H benchmarks have $8 < w_{90\%} \leq 16$, S benchmarks have $2 < w_{90\%} \leq 8$ and L have $1 \leq w_{90\%} \leq 2$.

TABLE IV : BENCHMARK CLASSIFICATION

H	$w_{90\%}$	S	$w_{90\%}$	L	$w_{90\%}$
ammp	14	crafty	4	applu	1
apsi	10	gcc	3	bzip2	1
art	10	gzip	4	eon	2
facerec	10	perl	5	equake	1
fma3d	9	swim	3	gap	2
galgel	15	vortex	7	lucas	1
mgrid	11			mcf	1
parser	11			mesa	2
twolf	15			sixtrack	1
vpr	15			wupwise	1

It is interesting to note that in average, SPEC CPU 2000 benchmarks need 6.11 ways to attain 90% of their peak IPC (in our baseline configuration). This means that, ideally, 61.8% of the 16 ways in the L2 can be turned off with just a 10% IPC degradation. This result is a good motivation for power and cache partitioning.

IV. EXPERIMENTAL ENVIRONMENT

In order to focus on the correctness of IPC predictions, we have evaluated our ealuation on single core superscalar processors with separated L1 instruction and data caches, and a unified set associative L2 cache. The processor is single threaded and can fetch up to 8 instructions each cycle. It has 6 integer (I), 3 floating point (FP), 4 load/store functional units, and 32-entry I, load/store, and FP instruction queues. The processor has a 256-entry ROB and 256 physical registers. We use a two-level cache hierarchy with constant 64B lines with separate 32KB, 4-way associative data and 2-way associative instruction cache, and a unified 1MB, 16-way L2 cache. Latency from L1 to L2 is 15 cycles, and from L2 to memory 250 cycles. We use a 32B width bus to access L2 and a multibanked L2 of 16 banks with 3 cycles of access time.

We have extended the SMTSim simulator [2] to obtain SDHs and the memory model to obtain performance predictions. We collected traces of the most representative 300 million instruction segment of each SPEC CPU 2000, following the SimPoint methodology [18].

We have simulated each application on all available L2 sizes (from 1-way to 16-ways available). At the same time, we predict the performance of the same application on all other L2 cache sizes using our methodology, described in Section III. That is, when running with 3 ways, we predict performance for 1 to 16 ways. For each IPC prediction $IPC_{pred,w'}$, of a real IPC $IPC_{real,w'}$, we measure the relative error as:

$$rel_error = 100 \cdot \frac{|IPC_{pred,w'} - IPC_{real,w'}|}{IPC_{real,w'}}$$

Finally, we compute the arithmetic mean of all the relative errors.

V. EVALUATION RESULTS

In this section we show the accuracy of our IPC prediction mechanism, as well as a sensitivity analysis regarding different processor parameters.

A. Accuracy

Figure 3 shows the average relative error for each SPEC CPU 2000 benchmark. Benchmarks belonging to the same group (L, S, H) are shown together. Overall, the average prediction error is 3.1%. It is important to note that average error is consistent across benchmarks of the same type, with L and S benchmarks having lower error, and H benchmarks having higher error.

Figure 4 shows detailed IPC prediction results for two benchmarks representative of different error ranges. The figure shows IPC predictions for all 16 L2 cache sizes when using data obtained running on K L2 cache ways. In the case of `gap`, the average prediction error is 0.27%, and prediction accuracy is very high regardless of the number of cache ways used to make the prediction. On the opposite size, texttparser has an avergae error of 9.8%. Predictions are inaccurate in all cases, because the impact of L2 cache misses is being underestimated in IPC prediction.

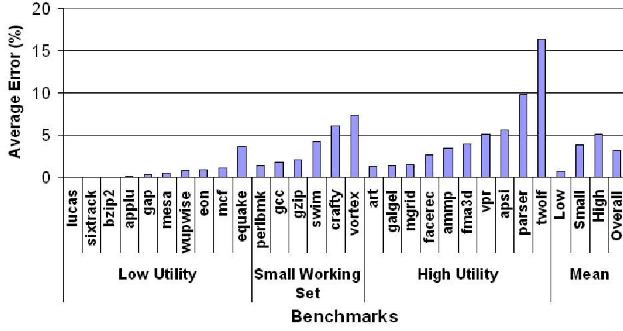


Fig. 3. Mean relative error for all SPEC CPU 2000 benchmarks

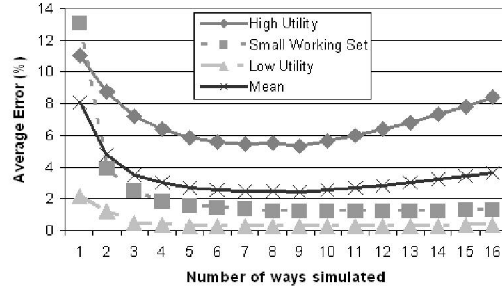
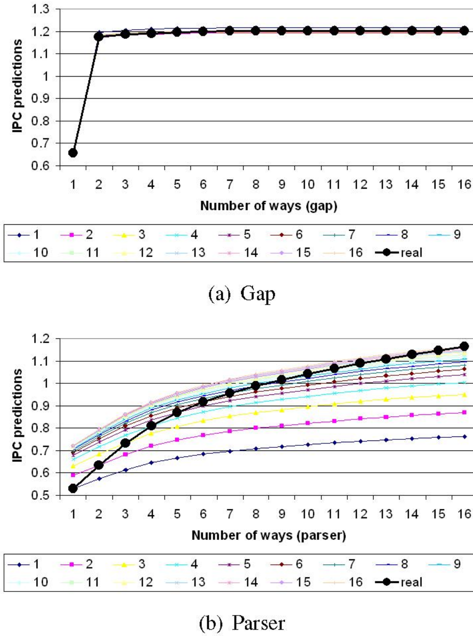


Fig. 5. Average relative error for H, S and L benchmarks when simulating a fixed number of ways



(a) Gap

(b) Parser

Fig. 4. IPC predictions for `gap` and `parser`

Global Error per Benchmark Group. It is significant that relative errors for each benchmark group H, S and L have similar results. In that way, we have that H benchmarks present an average relative error of 5.09%, while S benchmarks present lower errors (3.83% in average) and L benchmarks have negligible errors (0.72% in average). These results are intuitive as benchmarks that are memory bounded present more IPC variability and, as a consequence, predictions are less accurate.

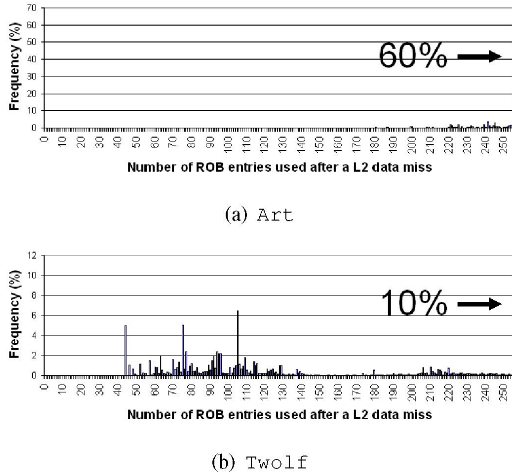
Error per Cache Size. Figure 5 shows the average IPC prediction error as we change the number of ways simulated to make the rest of the predictions. That is, the average error for predicting performance on 2-16 ways when running on 1 way, and so on. We have averaged the relative error for a given number of active ways among all benchmarks in the same group.

Our reults show that average error is higher for very small and very large L2 caches. In fact, if we have a number of ways between 4 and 12, the average relative error is under 3%. For small caches, we can see that errors are higher when predicting IPC for large caches. The same happens the other way round. This situation is intuitive as we are trying to predict IPC for caches up to 16 times larger or smaller. The plot shows a high peak error for the smallest L2 cache size. In this particular case, we are predicting the performance of a highly associative cache based on results of a 64KB direct mapped L2 cache and 32K L1 caches. This unusual setup naturaly leads to high prediction errors.

We have observed that H benchmarks present less accuracy when they are executing with just one or two ways, while S benchmarks have problems when using just one way. However, this situation is unlikely to happen. On the one hand, in a low power scenario, as these benchmarks satisfy $w_{90\%} > 2$, if we decide to loose 10% performance to reduce power, then these benchmarks will have at least 3 active ways (they would use exactly $w_{90\%}$ active ways). On the other hand, in a CMP scenario, we have done an experiment in a CMP architecture with two cores with a shared 16-way 1MB L2 cache and private instruction and data L1 4-way caches of 32KB. We have implemented the policies used in [5]–[7] to minimize the total number of L2 misses. 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. In average, S benchmarks receive more than one way in 92% of the decisions, while in the case of H benchmarks, this happens in 96% of all decisions. H benchmarks receive more than two ways in 91% of the decisions. Thus, in these situations the mean IPC prediction error would be even lower than the reported 3.11%.

Outliers. The highest errors among benchmarks are obtained for `parser` and `twolf`. These benchmarks do not satisfy the approximation that the misspenalty for an isolated data L2 miss is approximately ΔD. If there are instructions before the L2 miss in the pipeline, the processor can do some useful work while waiting for data to come from memory. This useful work is not easy to measure and even harder to predict for different sizes of L2. This obsrvation was also shown in [16].

Another source of inaccuracy is the use of a fixed value of ROB occupancy after a data L2 miss commits. It is clear that when the ROB size is the bottleneck, ROB will fill nearly always. This is exactly the case for `art`, which is the H benchmark with less error. In Figure 6(a) we show art's ROB occupancy after an L2 miss. However, in the case of `twolf` ROB occupancy after a data L2 miss varies a lot and the mean value is less representative of overlaps (see Figure 6(b)). Thus, when the ROB is not the main bottleneck for performance, we obtain higher errors in predictions.



(a) `Art`

(b) `Twolf`

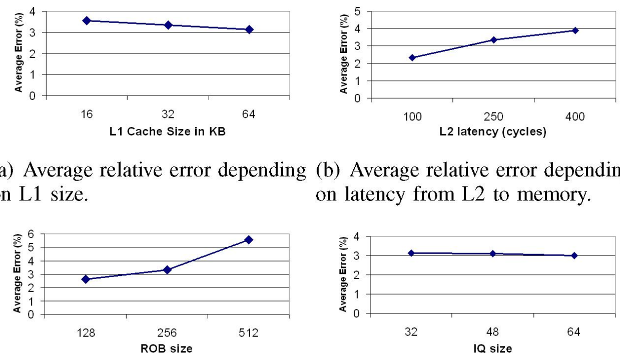
Fig. 6. ROB occupancy after a data L2 miss

B. Sensitivity Analysis

In this section, we check the sensitivity of our OPACU methodology regaring several processor parameters. We perform six different studies. The first two concern a parameter of the cache hierarchy, while the last four are related to the analytical model. The rest of the processor parameters remain constant.

Data and Instruction L1 size. First, we vary the L1 cache size from 16KB to 64KB, while keeping the associativity constant. When L1 cache size varies, the number of L2 accesses also varies. Larger L1 caches lead to less L2 accesses. A new hit to the L1 instruction or data cache would probably be a hit in L2 if the whole L2 cache is active. However, if just some ways of the L2 are active, this L1 hit could have been a miss in the L2 due to its low associativity. Thus, it is intuitive that the mean relative error will decrease with L1 cache size. In Figure 7(a) we can see the evolution of the mean relative error, confirming this intuition.

Latency from L2 to memory. In this experiment we vary the L2 cache latency from 100 cycles to 400 cycles, while keeping other parameters constant. In this situation, average relative error increases according to latency from L2 to memory. When this latency increases, the contribution of L2 misses to the total CPI becomes more important and the error increases. In Figure 7(b) we can see the evolution of the mean relative error, showing this behavior.



(a) Average relative error depending on L1 size.

(b) Average relative error depending on latency from L2 to memory.

(c) Average relative error depending on ROB size.

(d) Average relative error depending on issue queue size.

Fig. 7. Sensitivity Analysis

Interaction between Branch Misspredictions and Cache Misses. Next, we have observed that in some benchmarks many wrong path instructions are executed (for example, in the case of `twolf`, 40% of the fetched instructions are from the wrong path). Thus, we check the assumption that there is no interaction between L2 misses and branch prediction misses. We have run these benchmarks with perfect branch predictor and the mean relative errors result in approximately the same values. Hence, this hypothesis is confirmed to be correct.

Interaction between Instruction and Data Cache Misses. We have observed that in some benchmarks, the number of instruction L1 misses is high (for example, in the case of `crafty`). It is important to know if the interaction between instruction misses and data misses is accurately modeled. To verify this assumption, we have considered a perfect L1 instruction cache. However, simulations make no substantial differences in the results. Thus, instruction and data misses can be treated separately.

ROB size. In this study, we vary the ROB size from 128 to 512 entries. Figure 7(c) shows that for smaller ROB sizes, the mean relative error decreases, while the opposite occurs for greater ROB sizes. In our model, we expect the ROB size to be the main bottleneck after a data L2 miss. When the ROB is larger, it is more likely to happen that the issue queue becomes the bottleneck. Thus, the mean ROB occupancy after an L2 miss becomes less representative of overlaps. When the ROB is smaller, ROB occupancy after an L2 miss is concentrated around the full ROB size.

Queues size. Finally, we vary issue queues size from 32 to 64 entries. Although it does not appear directly in the formula, it is clear that smaller issue queues lead to more conflicts, becoming the main bottleneck instead of the ROB. Consequently, we have larger error in IPC predictions. In Figure 7(d) we can see the evolution of the mean relative error.

VI. HARDWARE IMPLEMENTATION

In this Section we show a possible implementation of our IPC prediction method. There are two main hardware components: First, we need a set of counters to know the number of committed instructions, the number of cycles and the average ROB occupancy after an L2 miss commits. Second, we need some hardware to track L2 accesses as other proposals have already shown [6], [7]. For each cache configuration [2] we use hardware to determine at runtime if two L2 accesses overlap, count the number of bursts of overlapped L2 misses, and predict IPC.

Overlap Counters. For each possible cache size we have a counter $overlap_w$ that counts the number of bursts of overlapped L2 misses. When we have a miss, we need to know if this miss is overlapped with previous misses. According to our memory model, instruction L2 misses do not overlap. Thus, bursts are always of only one miss. In case of an instruction L2 miss with stack distance i, we can directly increase the counter $overlap_i$. In case of a data L2 miss, we need to know the average ROB occupancy after an L2 miss commits to know if this particular L2 miss overlaps with previous ones. This information can be tracked with a hardware counter that we call *AROAL2M* (stands for Average ROB Occupancy After an L2 Miss commits). This counter is updated every time a served L2 miss commits. This value can be obtained with some cycles of latency, as this latency is not crucial because the mean value should not vary too much in a period.

Countdown Counter. When a new non-overlapped L2 miss with stack distance i appears, we set a *countdown counter* to the value of AROAL2M. This counter, called cdc_i, is different for each L2 cache size. Each time an instruction is committed, we decrease the counters, which saturate to zero. Thus, when a new L2 miss with stack distance i appears, if $cdc_i \neq 0$, then it overlaps with the previous L2 miss. Notice that we can count committed instructions that are actually prior to the miss. However, this is not a big problem as the number of instructions in the ROB when an L2 miss occurs is normally small [16]. In any case, if we want to check that the committed instruction is after the data L2 miss, we can add an instruction identifier that is assigned at fetch time and sorts instructions.

Predicted Cycle Variation Counters. Next, we need a counter for each cache configuration that stores the variation in total misspenalty due to L2 misses. We call this counter PCV_w for a cache configuration with w ways.

Instructions and Cycles Counters. Finally, we need a hardware counter that gives the total number of instructions committed in a fixed period of cycles, denoted as *Icounter*. The amount of cycles elapsed since we began tracking L2 accesses must be also stored in a counter denoted *Ccounter*.

Auxiliary Tag Directory (ATD). Normally, L2 caches have two separate parts that store data and address tags to determine if the access is a hit. Basically, our prediction mechanism needs to track every L2 access, and store a separate copy of the L2 tags information in an ATD, coupled with the LRU counters. As we have a cache with 16 ways, we need 4 bits to encode the stack distance. As we described in Section II, an access with stack distance larger than the associativity corresponds to a cache miss. Thus, with this ATD we can determine whether an L2 access would be a miss or a hit in the 16 possible cache configurations.

In Figure 8 we sketch a diagram of this hardware implementation. Our main contributor to hardware cost corresponds to the ATD, for which we propose to use a sampled version.

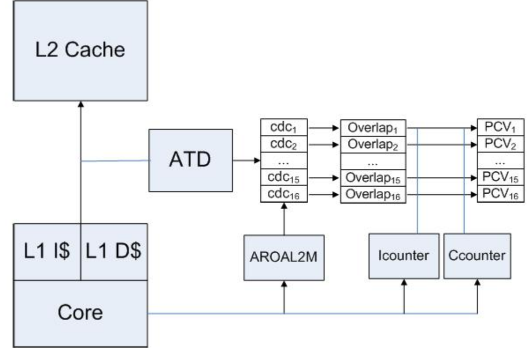


Fig. 8. Hardware implementation

Sampled ATD. Instead of monitoring every cache set, we can decide to track accesses from a reduced number of sets. This idea was also used in [6] in a CMP environment to determine the cache partition that minimizes the total number of misses using a sampled number of sets. Instead, we use it in a different situation, say to predict IPC with a sampled number of sets. We define a *sampling distance* d_s that gives the distance between sampled sets. For example, if $d_s = 1$, we are tracking all the sets. If $d_s = 2$, we track half of the sets, and so on. Sampling reduces the size of the ATD at the expense of less accuracy in IPC predictions. In this situation, some accesses are not tracked and, as a consequence, the information in the overlap counters is always less than real values.

Figure 9 shows the four error curves that are measured on the left y-axis. It is clear that for L benchmarks sampling makes no difference as their IPC is nearly constant for any L2 cache size. In S and H accuracy degradation is more important and errors quickly become significant. With a sampling distance of 8 we obtain average errors around 9%. We think that this is an interesting point of design.

Hardware Cost. We suppose a 40-bit physical address space. Each entry in the ATD needs 29 bits (1 valid bit + 24-bit tag + 4-bit for LRU counter). Each sampled set has 16 ways, so we have an overhead of 58B for each set. We also need 16 cdc_i counters of 1B because the ROB has 256 entries. It is necessary an extra counter of 1B for AROAL2M. Next, we need 16 $overlap_i$ counters of 4B, supposing a total of 64B. For the predicted cycles variation stored in PCV_w, we need

[2] We work at way granularity. Hence, we have as many configurations as the number of ways in the L2 cache associativity.

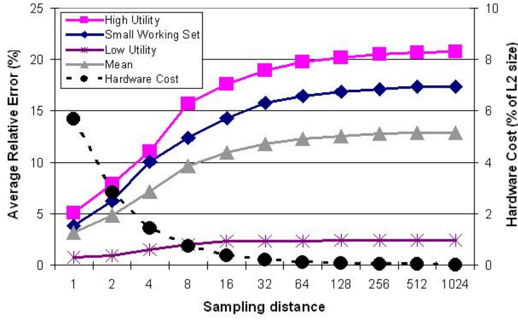


Fig. 9. Average relative error and hardware cost depending on the sampling distance

16 registers of 4B. Finally, we need two registers of 4B for Icounter and Ccounter. In that way, the hardware overhead of this circuits can be seen in Figure 9 measured on the right y-axis.

Improving Accuracy. In order to reduce even more prediction errors, we can use two methods. First, as sampling leads to a number of untracked accesses, we can scale the counters in the following way:

$$counter_{estimated} = scaling_factor \cdot counter$$

However, it is not easy to stablish such a scaling factor. It is clear that the number of misses overlapped by the first L2 miss depends on the size of available L2 cache. For example, if we have a smaller L2 cache, then we obtain more L2 misses and, as a consequence, bursts of overlapped L2 misses contain more L2 misses. Just to illustrate this point, in the case of `twolf`, the misspenalty for an L2 miss goes from 137 cycles to 192 cycles when we have 1 or 16 active ways. Thus, this scaling factor should depend on the number of ways that we are predicting. Furthermore, this value depends on the application. In that way, we have empirically found the optimal scaling factor for each sampling distance. These values must be stored in hardware. Figure 10 shows the accuracy in IPC predictions as we increase the sampling distance.

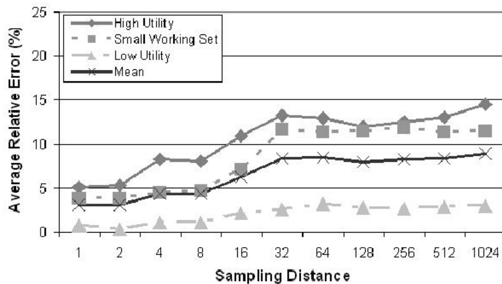


Fig. 10. Average relative error with optimal sampling factor

Another option is to combine the sampled ATD with information inside the L2 cache. If we have w ways active, we can use the LRU counters inside the L2 cache to detect any access with stack distance less or equal to w. Hence, this proposal requires that on every L2 hit we can read the LRU counter of the accessed line. Please note that this information is readable as it should be sent to the logic that determines which line to evict on an L2 miss. Thus, we only have to drive this information outside the L2 cache, which should not suppose many changes in the cache design. With this information, we can check if the access overlaps with previous accesses. The sampled ATD gives misses information for larger caches configurations. In this situation, IPC estimations are more accurate, as it can be seen in Figure 11, and we manage to obtain average errors under 5%. An interesting point of design is obtained with a sampling distance of 16, where we obtain average errors around 4%.

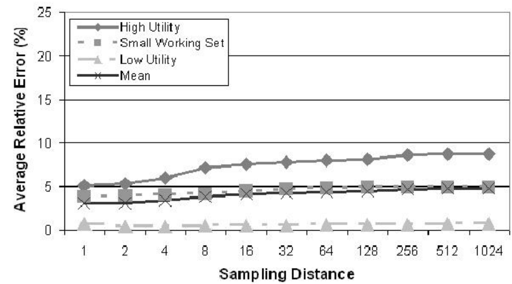


Fig. 11. Average relative error using L2 cache information

VII. Related Work

Many papers focus on predicting the IPC of applications. On the one hand, some papers try to reduce time complexity of design space exploration. Some studies have focused on reducing simulation time by selecting a representative small trace of an application [19]. A different approach consists in sampling design space points to train artifitial neural networks that predict performance in the whole design space with high accuracy [20]. However, our proposal lies in a different scenario, as it is a runtime mechanism. On the other hand, other papers predict IPC at runtime [21]. They analyze IPC in a window of time together with information obtained at compile time, and predict the future value of IPC. The main difference is that the processor configuration remains constant in IPC predictions.

Several papers have the objective of dynamically adjusting the cache size assigned to each thread in a multithreaded scenario. In [5], *column caching* is introduced. It allows to partition the caches in a cache hierarchy. In this paper, the control of cache partitions is let to the software (or the programmer). In [7], a dynamic cache partitioning technique is developed for SMT systems extending the previous paper. Here, partition sizes are varied at runtime using the total number of L2 misses. In [8], a similar dynamic scheme is introduced that decides partitions depending on the average data reuse of each application. In [9], instead of using the number of misses

for each application, they use the missrate to ensure fairness among threads. Finally, in [6] the authors present a suitable and scalable implementation of the technique appeared in [7] using sampling and showed similar performance gains with just an extra 0.2% space in L2 cache. However, all these proposals are using indirect measures of performance such us misses or data reuse. Our proposal is to use direct estimations of performance to obtain optimal partitions.

Regarding reconfigurable hardware for single threaded processors, several proposals try to reduce power consumption without loosing too much performance. In [11], *working set signatures* are used to represent programs instructions working set and find the minimal instruction cache size to minimize instruction misses (data misses are not treated). In [13], *selective cache ways* are introduced. These caches just precharge lines in active ways. This study is expanded in [12], where some algorithms are given to dynamically decide to switch on/off cache ways. However, they are unable to ensure a quality of service as they are using indirect measures of IPC. For example, they report a maximum IPC reduction of 52%. In [14], Cache Decay is proposed to dynamically switch off a cache line when it is highly probable that no more accesses will be done to this line. This occurs after a fixed number of cycles or other more agressive variants of this approach. In [22], the authors present a runtime decision algorithm for activating or inactivating cache lines based on the number of accesses to the LRU and MRU active lines. Finally, in [10], issue width and the number of execution units is dynamically modified depending on the present IPC. When the IPC is under a given threshold, then issue width is decreased. In this scenario, mainly all proposals cannot estimate performance degradation as they depend on empiric thresholds and heuristics concerning indirect measures of performance. Our proposal gives the opportunity to bound these losses.

VIII. Conclusions

Throughout this work we have presented a runtime mechanism that accurately predicts IPC as L2 cache size varies. We have shown average errors of 3.11% with predictions that follow the shape of the real IPC curve. To obtain these results, we have modified previous memory models to obtain higher accuracy in predictions. Furthermore, we have systematically classified benchmarks so that results are consistent in every benchmark group. We have also discussed a practical implementation that has an extra cost between 5.68% of the L2 cache size (best accuracy) and under 0.4% (for a 4% error). Hardware cost is reduced using a sampling technique.

Our mechanism can be used to reduce power consumption in single threaded architectures as it can be used to give the real contribution of each way to the final IPC and bound performance losses. A second possible application is to improve performance in multithreaded architectures that dynamically partition shared L2 caches. The mechanism we propose gives direct estimation of performance for different cache configurations, instead of other indirect measures of performance that are currently used to maximize total throughput.

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 Carmelo Acosta, Ayose Falcon, Daniel Ortega, Jeroen Vermoulen and Oliverio J. Santana for their work in the simulation tool. The authors would also like to thank anonymous reviewers for their helpful comments.

References

[1] M. J. Serrano, R. Wood, and M. Nemirovsky. A study on multistreamed superscalar processors. Technical Report 93-05, University of California Santa Barbara, 1993.

[2] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA*, 1995.

[3] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.

[4] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dynamically controlled resource allocation in SMT processors. In *MICRO*, 2004.

[5] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *Design Automation Conference*, 2000.

[6] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.

[7] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1):7–26, 2004.

[8] A. Settle, D. Connors, E. Gibert, and A. Gonzalez. A dynamically reconfigurable cache for multithreaded processors. *Journal of Embedded Computing*, 1(3-4), 2005.

[9] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.

[10] R. Iris Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *ISCA*, pages 218–229, 2001.

[11] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA*, 2002.

[12] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO*, 2000.

[13] D. H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *MICRO*, 1999.

[14] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *ISCA*, 2001.

[15] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[16] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA*, 2004.

[17] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *PACT*, 1999.

[18] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 2003.

[19] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. De Bosschere. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 2003.

[20] E. Ypek et al. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS*, 2006.

[21] S. Chheda et al. Combining compiler and runtime ipc predictions to reduce energy in next generation architectures. In *CF*, 2004.

[22] H. Kobayashi, I. Kotera, and H. Takizawa. Locality analysis to control dynamically way-adaptable caches. *Comput. Archit. News*, 33(3):25–32, 2005.