# Evaluating Cache Contention Prediction – A Ranking Based Approach –

Michael Zwick

***Abstract*—In multi-core processor systems, processor caches are generally shared among multiple processor cores. As a consequence, co-scheduled applications constantly displace each others data from the shared cache, which is called *cache contention*. As cache contention degrades application performance, efforts have been made to predict cache contention and then select application co-schedules that minimize interference. Several such methods have been proposed in the past. However, those methods have rarely been compared to one another, as an appropriate evaluation framework has been missing so far.**

**In this paper, I present a framework to precisely evaluate cache contention prediction techniques and compare them to one another. An implementation of the framework is licensed under GPLv3 and can be downloaded at http://www.ldv.ei.tum.de/cachecontention.**

***Index Terms*—cache-contention-prediction, multi-core, evaluation, framework.**

## I. Introduction

IN multi-core processor systems, applications are executed in parallel and simultaneously share resources such as memory controller, caches and busses (cf. figure 2). Sharing limited resources, however, applications frequently interfere with each other and degrade each other's performance. As an example, figure 1 shows L2 cache hitrate degradation of SPEC 2006 benchmark *milc* when milc is co-scheduled with other SPEC 2006 benchmark applications: The bold black line illustrates L2 cache hitrate of *milc* when *milc* is executed stand-alone and does not share any caches with any other applications. The other lines represent L2 cache hitrate of *milc* when *milc* is co-scheduled with applications *astar*, *gcc*, *bzip2*, *gobmk* and *lbm* respectively. Note that a higher L2 cache hitrate implies better performance. As you can see from the figure, L2 cache degradation of *milc* heavily varies with the selection of the co-scheduled application. While there are applications that have only little effect on L2 cache performance of *milc*, such as *astar* and *gcc*, there are co-schedules that have severe impact such as *gobmk* and *lbm*.
Generally speaking, application performance on multi-core processor systems does not only depend on the amount of resources provided by the computer system and the amount of resources the considered application applies for, but also on co-scheduled applications *sharing* those resources. As a consequence, resource-aware co-scheduling of applications has gained more and more attention the recent years and new scheduling techniques have been proposed that aim to optimize overall system performance by predicting and avoiding cache/resource contention. However, the proposed techniques have rarely been compared to other state-of-the-art methods, as an appropriate evaluation framework has been missing so far. In fact, such methods have rather been verified by comparing performance impacts of a state-of-the-art process scheduler to the impacts achieved by a modified version that predicts and minimizes resource contention by an adapted selection of co-schedules.

Manuscript received June 24, 2012; revised August 20, 2012.

M. Zwick is with the Department of Electrical Engineering and Information Technology, Technische Universität München, 80290 München, Germany; email: zwick@tum.de.

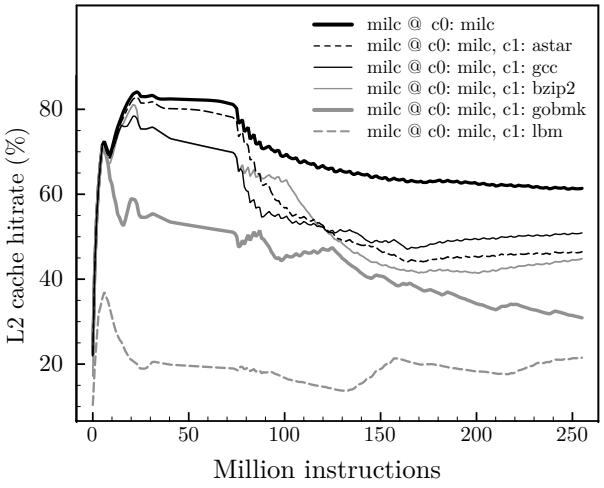


Fig. 1. L2 cache hitrate degradation introduced to SPEC 2006 benchmark *milc* when co-scheduling *milc* with each of the SPEC 2006 benchmarks *astar*, *gcc*, *bzip2*, *gobmk*, and *lbm* on a CMP dual-core architecture as presented in [1].

In the following, I

- present three techniques that have been applied in the past to evaluate cache contention prediction and discuss their benefits and limitations (sec. II),
- identify requirements on a new framework to evaluate cache contention prediction (sec. III),
- propose a new evaluation framework (sec. IV) and
- exemplarily show some evaluation results gained from the framework (sec. IV).

## II. State-of-the-Art Evaluation Techniques

### A. Fedorova et al.'s Evaluation Method

In [3], Alexandra Fedorova et al. analyze and evaluate prediction methods that estimate contention within a set of $|A| = 4$ applications $A = \{a_1, a_2, a_3, a_4\}$ that share resources on an Intel Quad-Core Xeon processor. The applied processor architecture can be described by two of the processors shown in figure 2 (greyed boxes) that are interconnected via the bus system.

For evaluation, the authors first apply the prediction method to *estimate* the best co-schedule for each of the applications in A. As an example, a prediction method might estimate the best co-schedule set to be $[(a_1, a_2), (a_3, a_4)]$, which means that applications $a_1$ and $a_2$ should be executed

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

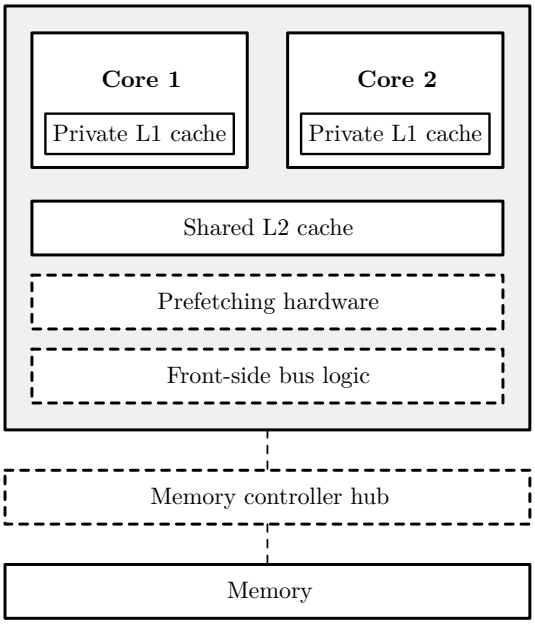


Fig. 2. Dual-core processor architecture as presented in [2]. While the L1 caches are private to the processor cores, the L2 cache is shared, which results in L2 cache contention.

on the one processor entity (greyed box in figure 2) and applications $a_3$ and $a_4$ should be executed on the other.

Then, the authors compare the degradation introduced to $a_1$, $a_2$, $a_3$, and $a_4$ on this *estimated* best application co-schedule to the degradation introduced to $a_1$, $a_2$, $a_3$, and $a_4$ when applying the *optimal* co-schedule set, e.g. $[(a_1, a_3), (a_2, a_4)]$.

This is done in the following steps:

- Run each application $a \in A$ stand-alone on the processor and measure execution time $t_a$, i.e. execution time of application $a$ when $a$ does not suffer from resource contention.

  Let $C_a$ be the set of applications that are co-scheduled with application $a$ on the same processor and share the L2 cache[1], and let $t_{C_a}$ be the execution time of application $a$ when application $a$ is co-scheduled with the applications in $C_a$, then
- for each $a \in A$ and each $C_a \in A\backslash\{a\}$, measure execution time $t_{C_a}$.
- For each $a \in A$ and each $C_a \in A\backslash\{a\}$, calculate the degradation $C_a$ introduces to $a$, i.e. $d_{C_a} = (t_{C_a} - t_a)/t_a$.

- Calculate average degradation of the *predicted* best co-schedule $d^{pred} = \sum_{a \in A} d_{C_a}/|A|$, where, for each $a \in A$, co-schedule $C_a$ is the *predicted* best co-schedule.
- Calculate average degradation of the *actual* best co-schedule $d^{act} = \sum_{a \in A} d_{C_a}/|A|$, where co-schedules $C_a$ is chosen such that $d_{C_a}$ is minimized.
- Determine evaluation metric E as percentaged performance degradation of $a$ when applying the *predicted* best co-schedule instead of the *actual* best co-schedule

$$E = ((d^{pred} - d^{act})/d^{act}) \cdot 100\,\%.$$

[1]If there are only two processor cores sharing a cache, then $C_a$ consists of only one application. Given a co-schedule set $(a_1, a_2)$ and let $a = a_1$, then $C_a = \{a_2\}$.

$E$ is the metric Fedorova et al. apply in [3] to evaluate methods that predict contention for shared resources on multi-core processors.

The authors limit their evaluation to prediction accuracy. An evaluation of the amount of time necessary to create or apply the predictors is not performed.

Fedorova et al. are one of only a few authors that evaluate accuracy of contention prediction methods in the context of state-of-the-art prediction techniques. As evaluation method, they compare program execution times of predicted best co-schedules to program execution times of actual best co-schedules, measured on a physically available processor, not a simulator.

Generally, Fedorova et al.'s method to evaluate the prediction of resource contention is a highly accepted approach, as it relies on *real* applications executed on a *physical* machine. And as long as this approach is applied to evaluate methods to predict *contention in general*, everything is fine.

But it comes to problems when you try to use this technique to soleley evaluate methods to predict *cache contention*, as execution time in general depends on more factors than just cache misses, and the ground-truth reference then would incorporate effects that do not origin from cache contention – and are not modeled in the prediction techniques. In [3], Fedorova et al. address this effect when they explain the surprisingly good performance of stand-alone cache misses to predict cache misses of application co-schedule sets, as it has been proposed by Knauerhase et al. [4].

As a consequence, a cache simulator would be more suited to evaluate the prediction accuracy of cache contention prediction methods, as results obtained from a cache simulator would more precisely reflect the characteristics addressed by cache contention prediction methods. However, when it comes to an evaluation of prediction accuracy regarding contention *in general*, i.e. when *overall* system performance is the measure to optimize, real hardware would definitely be the best choice.

### B. Chandra et al.'s evaluation method

In [5], Dhruba Chandra et al. propose and evaluate three methods to predict L2 cache contention. Just like Fedorova et al., Chandra et al. focus on prediction accuracy and do not evaluate the amount of time required to perform a prediction. Contrary to most other methods, their methods directly predict additional L2 cache misses introduced from sharing the L2 cache.

To evaluate their methods, the authors compare the *predicted* amount of additional cache misses to the *actual* amount of of additional cache misses obtained from a processor simulator (2 cores, private L1 cache for each core, shared L2 cache, as shown in figure 2) as follows. They

- apply the simulator to determine stand-alone L2 cache misses $\mu_a$ of $|A| = 9$ different applications $a \in A = \{a_1, a_2, \ldots a_9\}$,
- select 14 out of $(9-1)! = 40320$ possible application pairs $(a, C_a)$, $a \in A$, $C_a \in A\backslash\{a\}$, and apply the simulator to determine the amount of L2 cache misses $\mu_{C_a}$ of application $a$ when $a$ is co-scheduled with application $C_a$,

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

- calculate the *actual* amount of *additional* cache misses introduced to application $a$ when co-scheduling $a$ with $C_a$ as $\mu_{C_a}^{sim} = \mu_{C_a} - \mu_a$,

- apply the prediction method to calculate a *prediction* of the amount of additional cache misses $\mu_{C_a}^{pred}$ introduced to $a$ by co-scheduling $a$ with $C_a$,
- determine the prediction error regarding co-schedule set $(a, C_a)$ by $|\mu_{C_a}^{pred} - \mu_{C_a}^{sim}|$ for all mentioned co-schedule sets and
- take the arithmetic mean of the error

$$E_a = \frac{\sum_{(a,C_a)\in A} |\mu_{C_a}^{pred} - \mu_{C_a}^{act}|}{|A| \cdot (|A| - 1)}$$

and the geometric mean of the error

$$E_g = (\Pi_{(a,C_a)\in A}(1 + |\mu_{C_a}^{pred} - \mu_{C_a}^{sim}|))^{\frac{1}{|A|^2-|A|}} - 1$$

as evaluation metric.

As you can see from $E_a$ and $E_g$, the methods applied by Chandra et al. are well suited for their evaluation, as all their prediction methods provide an amount of additional L2 cache misses as outcome. However, applying Chandra et al.'s method as a *general* technique to evaluate cache contention prediction methods, you would run into trouble: What would you do if the methods you want to evaluate do not predict cache contention in terms of additional cache misses, as it is exemplarily the case with the *Pain* method proposed by Fedorova et al. [3]? As shown in [6], many prediction methods provide values without any technical meaning, enforcing an evaluation of prediction method only in relation to one another. To such methods, Chandra et al.'s evaluation technique cannot be applied.

### C. Settle et al.'s Evaluation Method

In [7], Alex Settle et al. propose a cache contention prediction method based on so-called *activity vectors*. Activity vectors are bit vectors that represent accesses to and misses in groups of cache sets. The authors do not compare their method to state-of-the-art techniques, but show the effectiveness of their approach by applying the following steps: They

- define a set of applications as processor workload, then
- execute the workload on a hyperthreading enabled Intel Pentium Xeon microprocessor applying a standard Linux scheduler and measure $IPC^{act}$ (instructions per cycle) and $ITKO^{act}$ (inter thread kick outs, i.e. the amount of an application's cache lines that are displaced by a co-scheduled application) values,
- adapt the standard Linux scheduler to incorporate the prediction method in the scheduling decision process,
- calculate the prediction from the workload and
- execute the workload with the adapted scheduler enabled, measure $IPC^{pred}$ and $ITKO^{pred}$ and
- take

$$E_{IPC} = (IPC^{pred} - IPC^{act})/IPC^{act} \text{ and}$$
$$E_{ITKO} = (ITKO^{pred} - ITKO^{act})/ITKO^{act}$$

as evaluation metric.

At first sight, the evaluation technique applied by Settle et al. looks applicable to evaluate cache contention prediction methods in general, as it

- adapts the scheduler, as it is proposed for scenarios that exploit cache contention,
- evalutes the prediction technique by means of IPC, that is correlated to *execution time* and, besides that,
- evaluates the prediction technique by means of ITKO, a metric that directly reflects the measure of interest, and
- compares to be improved values to actual values.

In contrast to the evaluation method applied by Chandra et al., this evaluation method

- does not rely on a specific format of prediction outcome (e.g. additional cache misses), what makes it applicable to a broad range of prediction techniques.

However, there are some limitations using this evaluation technique:

- First, the evaluation technique only works on larger *sets* of applications; that means that you are not able to evaluate a predictor regarding cache interference on a specified pair of applications, but only on a larger set. Although this might not seem to be a crucial point at first sight, it makes a systematic analysis of predictors impossible, as you are not able to correlate application characteristics (number of memory references, number of misses, shape of the stack distance histogram) to prediction accuracy. As a consequence, you are not able to experiment with selected application characteristics in order to analyse and improve predictor accuracy.
- Secondly, scheduler internals have a significant impact on choosing the applications to be co-scheduled. This means that scheduling decisions are not only based on the predictors, but of course also on priorities etc. As a consequence, you evaluate prediction techniques by means of scheduler decisions; but those scheduler decisions do not 100% originate from the prediction method, but also from many side effects. As an example, two applications that perfectly minimize cache contention might never be co-scheduled if their priorities tell the scheduler not to do so. And the priorities might be someting you cannot control as they change at runtime. This introduces errors and might overlay evaluation results such that an actually better application co-schedule might be evaluated worse.
- Thirdly, as scheduling decisions also rely on priorities, they also rely on the past, as priorities rely on scheduler activities performed in previous steps. Then, as a consequence, not only cache contention relies on the past due to the memory characteristic of the cache's LRU stack, but then also the evaluation system does, while prediction methods do not. This introduces a huge amount of complexity in the evaluation process and renders an in-depth analysis of cache contention prediction techniques nearly impossible.

As a consequence, Settle et al.'s evaluation technique might be a good choice to show the effectiveness of a specific cache contention prediction method in general, as it has been intended by the authors, but it does not seem to be suited for an in-depth analysis and comparison of multiple cache contention prediction techniques.

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

## III. Ideas and Requirements on a Framework to Evaluate Cache Contention Prediction

Given some insights gained from the analysis of state-of-the-art evaluation methods — how should a framework to evaluate cache contention prediction methods look like?

- As I showed in the discussion of Fedorova et al.'s evaluation method in the previous section, execution time on a real hardware system is not a valid ground-truth reference if it comes to separate the effects of *cache* contention from the effects of contention *in general*. Therefore, a new framework to evaluate cache contention prediction methods should calculate a ground-truth measure applying an appropriate cache simulation framework.
- There are many cache contention prediction techniques that provide values that can only be used for an evaluation of prediction methods *in relation to each another*, as I pointed out in the previous section. To support evaluation of such methods, a new evaluation framework should be able to evaluate cache contention prediction techniques by means of a ranking list.
- State-of-the-art evaluations of cache contention prediction methods have generally focused on prediction accuracy. A comparative evaluation of the amount of time required to perform a prediction has – to the best of my knowledge – never been performed. Remedial action has to be taken here and a new evaluation framework should support a timing analysis of cache contention prediction techniques.
- In order to be able to perform a fair evaluation of prediction time, it is necessary to determine which prediction steps have to be performed at runtime, and which steps can be performed prior to runtime. Therefore, it seems to be useful to distinguish between *prediction* and *predictors* as follows:
  - *Prediction* is the process of combining a set of predictors of candidate co-schedules in order to predict cache contention. This process adds to prediction time, as it cannot be calculated before runtime, i.e. before the set of candidate co-schedules is known.
  - *Predictors* are the basic blocks to store application characteristics. In order to support arbitrary co-schedule setups, predictors hold information of solo applications only, i.e. they do *not* hold information of *multiple* applications. As a consequence, all information required to calculate predictors is available prior to runtime, when no information of candidate co-schedules is available. Therefore, predictors can be calculated prior to runtime and their calculation does not add to prediction time.
- As prediction time has not been evaluated in the past, a comparative cost vs. gain analysis of prediction techniques is also missing. Therefore, a new evaluation framework should support such an analysis.
- State-of-the-art analysis of cache contention prediction techniques has generally been performed in the context of a given processor architecture, a given set of applications and a fixed amount of instructions. What has been missing so far is a sensitivity analysis of prediction accuracy regarding
  - the amount of applications sharing a cache in parallel (e.g. a comparative analysis of prediction accuracy in case a cache is shared by 2, 4 or 8 processor cores) and
  - the amount of instructions applied for a prediction (interval size).

  To overcome this problem, an evaluation framework should not focus on a specific architecture or a specific amount of processor cores sharing a cache, but be indifferent with respect to those characteristics. Further, the amount of instructions to be incorporated in the prediction process should be parameterizable.
- Further, the framework should be made up from building blocks, rendering it possible to add evaluation methods as plug-ins.
- In order to make evaluation results reproducible and errors easy to find, a notation should be applied that makes it easy to determine the application, interval size, co-schedules etc. applied within a prediction process.

## IV. A New Cache Contention Prediction Evaluation Framework

In this section, I present a new cache contention evaluation framework that implements all the requirements proposed in the last section. See figure 3 for an overview of the evaluation framework. As the framework has to support techniques that predict cache contention of application co-schedules only in relation to one another, the framework evaluates these techniques by comparing ranking lists of actual vs. predicted cache contention as follows:

Part (A) extracts memory references from applications and stores them to tracefiles; part (B) calculates predictors; part (C) applies those predictors to estimate cache contention; part (D) calculates a ground truth reference; part (E) compares the predictions to the ground truth reference and evaluates the prediction methods accordingly.

In the following I present those five steps in more detail.

### A. Extracting Memory References

Part (A) of the evaluation framework addresses the proposed requirements that

- ground-truth values should be generated by an appropriate cache simulator and
- prediction is solely based on predictors of solo applications,

as it provides memory references as input to the ground-truth simulator and the process of predictor calculation. As indicated in figure 3, the evaluation framework applies the Pin toolkit [8] to extract memory references $M = \{M_{a_1}, M_{a_2}, \dots\}$ from a set of applications $A = \{a_1, a_2, \dots\}$ and stores them to tracefiles: Given an application $a \in A$, Pin interrupts program execution on each memory reference of $a$ and stores the referenced address to a buffer. Whenever the buffer represents memory references of $2^{20}$ instructions, the buffer is written to disk. Then, the buffer is reset and memory references of the next $2^{20}$ instructions are recorded. This procedure repeats iteratively until a specified number of instructions has been processed. This step results in a tuple of memory references $M_a$.

In order to allow the evaluation to be performed on a

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

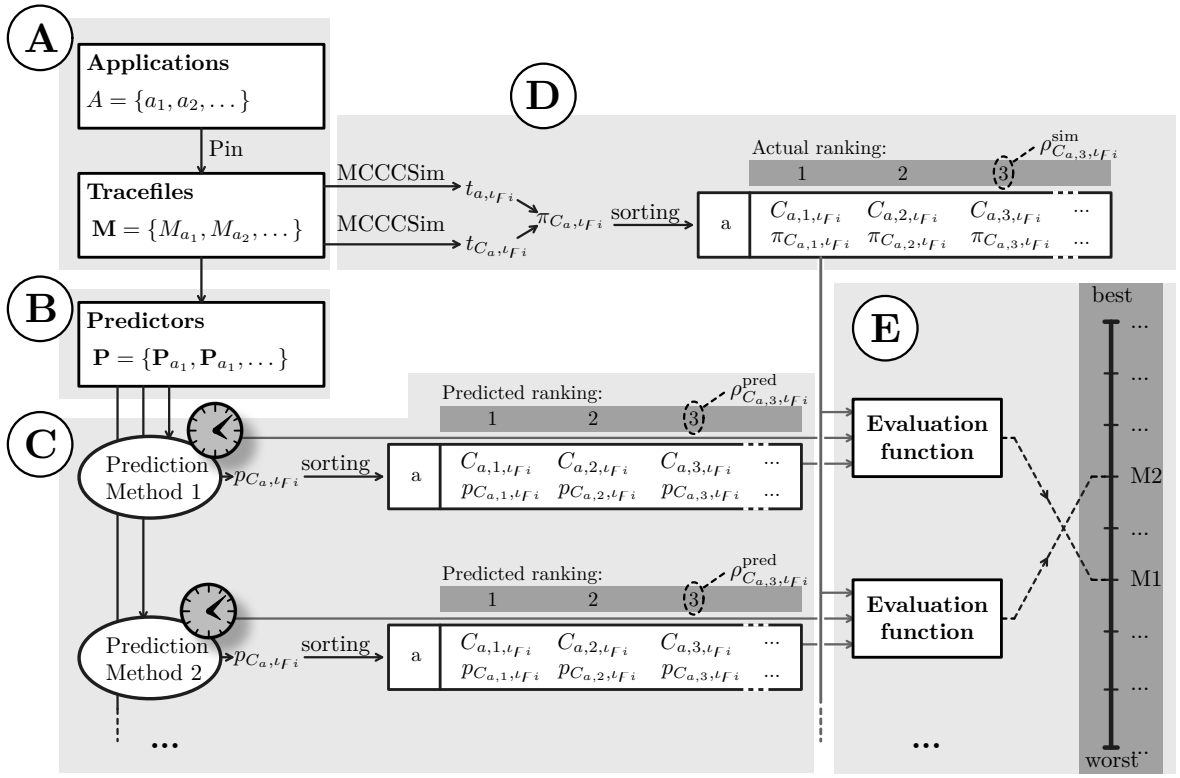


Fig. 3. Evaluation framework made up of five parts: (A) Extraction of memory references; (B) predictor calculation; (C) prediction of cache contention; (D) calculation of ground-truth reference; (E) evaluation. See section IV for a definition of the various symbols.

broad range of interval sizes as claimed in section III, I partition $M_a$ into tuples of memory references that refer to interval sizes of $F \in \boldsymbol{F} = \{ 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}, 2^{25}, 2^{26}, 2^{27}, 2^{28}, 2^{29}\}$ instructions, i.e. $\forall F \in \boldsymbol{F} : M_a \mapsto M_{a,\iota_F} = (M_{a,\iota_{F1}}, \dots M_{a,\iota_{F|M_{a,\iota_F}|}})$. In this definition, $\iota$ means "instruction interval", $\iota_F$ is the set of "instruction intervals each referring to $F$ instructions", and $\iota_{Fi}$ means "instruction interval no. $i$ in the set of instruction intervals each referring to $F$ instructions"; $|M_{a,\iota_F}|$ is the total number of instructions related to memory references $M_a$, divided by interval size $F$. Applying this partitioning scheme and the corresponding notation allows for a unique identification of the instruction intervals the memory references have been extracted from. When it comes to prediction, this identification scheme makes it possible to unambiguously relate the value of a prediction $p_{C_a,\iota_{Fi}}$ to the application $a$, the co-scheduled applications $C_a$, and the instructions the predictor has been obtained from, as instruction interval $\iota_{Fi}$ refers to instructions $i{\cdot}F \dots (i{+}1){\cdot}F{-}1$. This transparency allows the whole research community to easily review and reproduce the results — or easily identify or locate errors if there are any, as it has been claimed in section III.

### B. Calculating Predictors

Part (B) of the evaluation framework calculates the predictors as basic building blocks for the predictions, as claimed in section III. For this purpose, the evaluation framework applies the tuples of memory references $M_{a,\iota_{Fi}}$ to calculate predictors for the various cache contention prediction methods to be evaluated. As an example for a prediction method, take one of the methods referenced by Fedorova et al. [3] or Chandra et al. [5]. As an example for a predictor, take solo application miss rate [4] or an application's stack distance histogram [9] [5].

As it is the key point for the introduction of predictors to distinguish between calculations performed at runtime and prior to runtime (cf. section III), a rule has to be defined to decide which calculations have to be merged to a predictor, and which calculations have to be calculated during the prediction phase. Without such a rule to unambiguously identify the parts that have to be calculated prior to prediction and which have to be calculated during the prediction process, prediction time of the various prediction methods would not be comparable and the evaluation results would be unreliable.

In this framework, I apply the rule that predictors should *incorporate as much information as possible*.

What does that mean? Generally, cache contention prediction techniques merge information of co-scheduled applications to calculate their predictions. This means that there is a point in time within the prediction process you can only pass if you know the candidate co-schedules. As this information is definitely not available before runtime, this is a good point to distinguish between calculations that have to be included in the predictors and therefore *do not* account for prediction time, and those calculations that have to be performed during the prediction process and therefore *do* account for prediction time. As a consequence, predictors exclusively rely on characteristics of solo applications, i.e. there is a predictor for each $a \in A$ and this predictor is not allowed to contain information of any other application $C_a \in A \backslash \{a\}$.

### C. Prediction

Part (C) of the evaluation framework applies the predictors calculated in part (B) to predict the amount of cache

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

contention introduced to an application $a$ by each application candidate co-schedule. As you can see from figure 3, predictions are referenced as $p_{C_a,\iota_{F}i}$. This means that their value is a measure of cache contention introduced to application $a$ when co-scheduling instructions $i \cdot F \ldots (i+1) \cdot F - 1$ of $a$ with instructions $i \cdot F \ldots (i+1) \cdot F - 1$ of each application in the set of candidate co-schedules $C_a$. As an example, $a$ might be application *astar* and $C_a$ might be {*bzip2*, *gcc*, *h264ref*}, if applications are chosen from the SPEC 2006 benchmark set.

As you can see from figure 3, the output of part (C) is a ranking of candidate co-schedules based on the predicted amount of cache contention the candidate co-schedules introduce to application $a$. There is a separate ranking list calculated for each application $a$, each interval size $F \in \mathbf{F}$ and each interval $i$ as follows:

- For each candidate co-schedule of application $a$ in interval $\iota_{F}i$, apply the prediction method to estimate cache contention introduced to application $a$ as presented in algorithm 1.
- Measure *user* time $\tau^{\text{user}}_{C_a}$, *system* time $\tau^{\text{syst}}_{C_a}$, and *elapsed* time $\tau^{\text{elap}}_{C_a}$ as presented in algorithm 1.
- Sort all the predictions and assign each candidate co-schedule $C_{a,j}$ its predicted ranking position $\rho^{pred}_{C_{a,j},\iota_{F}i}$, for example "1" if this candidate co-schedule is predicted to introduce the least amount of cache contention to application $a$.

**Algorithm 1** Measuring user time, system time, and elapsed time.

1: Extract parameters $a$, $C_a$ and $\iota_F$
2: Begin timing by calling `gettimeofday(...)` and `getrusage(...)`
3: **for all** $\iota_{F}i$ **in** $\iota_F$ **do**
4: Read predictors of interval $\iota_{F}i$ for the selected method
5: Calculate $p_{C_a,\iota_{F}i}$
6: Store $p_{C_a,\iota_{F}i}$ to buffer in RAM (random access memory)
7: **end for**
8: End timing by calling operating system functions `gettimeofday(...)` and `getrusage(...)`
9: Calculate $\tau^{\text{user}}_{C_a}$, $\tau^{\text{syst}}_{C_a}$ and $\tau^{\text{elap},}_{C_a}$ from `timeval` and `rusage` structs
10: Write $\tau^{\text{user}}_{C_a}$, $\tau^{\text{syst}}_{C_a}$ and $\tau^{\text{elap}}_{C_a}$ to disk
11: Write $p_{C_a,\iota_{F}i}$ for all $\iota_{F}i \in \iota_F$ to disk

Compared to many state-of-the-art cache contention prediction evaluation techniques such as the one of Chandra et al. [5], the *ranking* of candidate co-schedules allows for a broad range of prediction methods to be evaluated with this framework: It is not necessary that a prediction technique estimates contention in terms of additional cache misses — the evaluation can also be applied on prediction techniques that only go for a relative evaluation of candidate co-schedules, where the absolute value of a predictor might not have any technical meaning, as it is claimed in section III.

### D. Ground Truth Reference

Part (D) of the evaluation framework applies a cache simulator to calculate the *actual* amount of cache contention for each candidate application co-schedule and creates a ranking list (cf. figure 3). In order to support a broad range of processor architectures, as claimed in section III, I apply the Multi-Core Cache Contention Simulator MCCCSim [2] for this task. As you can see from figure 3, the MCCCSim simulator takes as input the tuples of memory references $M_{a,\iota_{F}i}$ that have been generated by part (A). Given the memory references, the framework performs the following steps:

- Simulate stand-alone execution for each application $a \in A$ for each interval size $F \in \mathbf{F}$ and each interval $\iota_{F}i$ and determine memory accesstime $t_{a,\iota_{F}i}$ as described in [2].
- For each possible candidate-coschedule $C_a \in A \backslash \{a\}$ of application $a$ and each application $a \in A$, simulate co-scheduled execution of applications $a$ and $C_a$ for each interval size $F \in \mathbf{F}$ and each interval $\iota_{F}i$, and determine memory accesstime $t_{C_a,\iota_{F}i}$ of application $a$ in case $a$ is co-scheduled with $C_a$, as described in [2].
- For each application $a \in A$, calculate penalty $\pi_{C_a,\iota_{F}i} = t_{C_a,\iota_{F}i} - t_{a,\iota_F}$ that co-schedule $C_a$ introduces to $a$ for each interval size $F \in \mathbf{F}$ and each interval $\iota_{F}i$.
- For each application $a$, each $F \in \mathbf{F}$ and each interval $\iota_{F}i$, sort the penalties by value and determine ranking position $\rho^{\text{sim}}_{C_{a,j},\iota_{F}i}$ for the corresponding candidate co-schedules.

For each application $a$, each interval size $F$ and each interval $\iota_{F}i$, these rankings $\rho^{\text{sim}}_{C_{a,j},\iota_{F}i}$ determine the candidate co-schedule that best/second best/third best/... minimizes the amount of cache contention introduced to application $a$.

### E. Evaluation Functions

Part (E) of the evaluation framework analyzes and combines

- the amount of time to perform a prediction,
- the predicted ranking positions $\rho^{pred}_{C_a,\iota_{F}i}$, and
- the simulated ranking positions $\rho^{sim}_{C_a,\iota_{F}i}$

to generate an overall evaluation measure. As claimed in section III and as you can see from figure 3, evaluation functions can easily be integrated in the evaluation framework, as the framework provides a kind of plug-in mechanism.

In the following, I present some of the evaluation functions integrated in the evaluation framework.

- NMRD (Normalized Mean Ranking Difference) is an evaluation function that determines the average number of positions the predicted ranking is off the simulated ranking.
  Let
  - $\psi$ be the amount of parallelism of the processor architecture, i.e. the number of processor cores that share the same cache (e.g. $\psi = 2$), and let
  - $\mathbf{C}^{\psi}_a$ be the set of all possible co-schedules of application $a$ that can be obtained from $A \backslash \{a\}$ in case of a processor with $\psi$ processor cores, and let
  - $|\mathbf{C}^{\psi}_a|$ be the numbers of entries in set $\mathbf{C}^{\psi}_a$, i.e. the number of all possible co-schedules to an application $a$ that can be obtained from $A \backslash \{a\}$ in case of a processor with $\psi$ processor cores, and let

$$\Delta\rho_{C_a,\iota_{F}i} = \left| \rho^{\text{pred}}_{C_a,\iota_{F}i} - \rho^{\text{sim}}_{C_a,\iota_{F}i} \right|$$

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

be the distance between the predicted and the actual ranking position of co-schedule $C_a$ in interval $\iota_{Fi}$, then I calculate so-called mean ranking difference MRD by

$$MRD_{\mathbf{C}_a^\psi,\iota_{Fi}} = \frac{1}{|\mathbf{C}_a^\psi|} \sum_{C_a \in \mathbf{C}_a^\psi} \Delta\rho_{C_a,\iota_{Fi}},$$

as presented in [1].

Figure 4 shows an example of an MRD mean ranking distance value for the set of SPEC2006 benchmark applications $A = \{$astar, bzip2, gcc, gobmk, h264ref, hmmer, lbm, mcf, milc, povray$\}$, $a$ = astar, $\psi = 2$. In interval $\iota_{Fi}$, the actual best co-schedule to *astar* is *hmmer*, the worst co-schedule is *lbm*. For *hmmer*, there is a difference between actual and predicted ranking positions of $\Delta\rho_{\text{astar@}\{\text{hmmer}\},\iota_{Fi}} = |1-5| = 4$.

As the range of MRD values depends on the number of candidate co-schedules, MRD values for different $\psi$ have to be normalized to the total number of candidate co-schedules in order to make them comparable to one another. To determine the maximum MRD value, I present figure 5. It is obvious that figure 5 a) represents a ranking with maximum MRD value for an even number of co-schedules and figure 5 b) represents a ranking with maximum MRD value for an odd number of co-schedules. Therefore, maximum *MRD* calculates by

$$MRD^{\max}_{\mathbf{C}_a^\psi} = \frac{\left\lceil \frac{|\mathbf{C}_a^\psi|}{2} \right\rceil \cdot \left\lfloor \frac{|\mathbf{C}_a^\psi|}{2} \right\rfloor \cdot 2}{|\mathbf{C}_a^\psi|}.$$

Given $MRD^{\max}_{\mathbf{C}_a^\psi}$, I normalize *MRD* by

$$NMRD_{\mathbf{C}_a^\psi,\iota_{Fi}} = \frac{MRD_{\mathbf{C}_a^\psi}}{MRD^{\max}_{\mathbf{C}_a^\psi}} = \frac{\sum_{C_a \in \mathbf{C}_a^\psi} \Delta\rho_{C_a,\iota_{Fi}}}{\left\lceil \frac{|\mathbf{C}_a^\psi|}{2} \right\rceil \cdot \left\lfloor \frac{|\mathbf{C}_a^\psi|}{2} \right\rfloor \cdot 2}$$

to obtain the so-called Normalized Mean Ranking Difference $NMRD_{\mathbf{C}_a^\psi,\iota_{Fi}}$ that allows relative comparison of cache contention prediction methods even for different values of $\psi$.

In order to get an overall NMRD value for a given $\psi$ and a given $F$, $NMRD_{\mathbf{C}_a^\psi,\iota_{Fi}}$ has to be averaged over all $a \in A$ and all $\iota_{Fi} \in \iota_F$:

$$NMRD(\psi, F) = \frac{1}{|A|} \cdot \sum_{a \in A} \frac{1}{|\iota_F|} \cdot \sum_{\iota_{Fi} \in \iota_F} \frac{\sum_{C_a \in \mathbf{C}_a^\psi} \Delta\rho_{C_a,\iota_{Fi}}}{\left\lceil \frac{|\mathbf{C}_a^\psi|}{2} \right\rceil \cdot \left\lfloor \frac{|\mathbf{C}_a^\psi|}{2} \right\rfloor \cdot 2}.$$

Figure 6 shows an example output of the evaluation framework applying the *NMRD($\psi$, $F$)* evaluation function as presented in [6]: Column 1 shows the name of the prediction method, while columns 2, 3, and 4 show NMRD performance in case $\psi = 2$, $\psi = 4$, and $\psi = 8$ processor cores sharing the L2 cache. Lower NMRD values indicate better performance. You can see that there is no great performance difference of the prediction methods regarding a variation in the number of processor cores. Further, variations in the size $F$ of execution intervals have only limited effect on general ranking performance. Further, you can see that the "Miss rate" method shows good prediction performance, as already discussed regarding Fedorova et al.'s method [3].

Besides NMRD, the evaluation framework supports many other evaluation functions:

- MP (Mean Penalty) enhances NMRD by evaluating execution time instead of ranking positions.
- PPBAB (Penalty Predicted Best vs. Actual Best) does not evaluate general ranking performance, but the ability of a cache contention prediction method to determine the *best* co-schedule for a given application.
- PPBRS (Penalty Predicted Best vs. Random Selection) evalutes the gain in memory access time the best predicted co-schedule achieves compared to the average memory accesstime of all candidate co-schedules.
- The timing performance module calculates average user, system and elapsed time for a prediction.
- *Gain vs. Cost* is an evaluation function that combines PPBRS values with the time required to perform a prediction in order to determine if a prediction method takes more time for prediction as it will gain from proper co-scheduling. In combination with an evaluation of different interval sizes $F$, this method can be used to determine the minimal interval size to make a prediction method become beneficial.

## V. Conclusion

In this paper, I analyzed three state-of-the-art methods to evaluate cache contention prediction techniques. I showed that a hardware based evaluation, as it has been performed by Fedorova et al., might be a good technique to evaluate prediction accuracy of methods that do not only evaluate a prediction of cache contention, but a prediction of contention regarding the whole processor system. However, when it comes to an evaluation of cache contention prediction techniques only, a simulator based approach seems to be a better choice, as it makes ground-truth values rely on the same criteria as the predictors – although they might be less accurate predicting contention introduced from other components than caches.

Based on the discussion of state-of-the-art methods, I identified and discussed several requirements for a new cache contention evaluation framework. A key point has been evaluating the difference between predicted vs. simulated ranking positions, as there are cache contention prediction techniques that allow for a relative evaluation only. A further key aspect has been the proper definition of predictors to distinguish between prediction steps that are performed at runtime and prediction steps to be performed prior to runtime.

Driven by the insights gained from the previous steps, I presented a new evaluation framework that extracts memory references from a set of computer applications, applies those memory references to calculate predictors and exploits those predictors to predict rankings of application co-schedules that minimize cache contention. To evaluate prediction accuracy, the rankings are compared to ground truth reference data obtained by the Pin framework and analyzed by a set of evaluation functions.

As a last step, I presented a typical output of the evaluation framework. The output compared prediction accuracy of several cache contention prediction methods regarding general

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

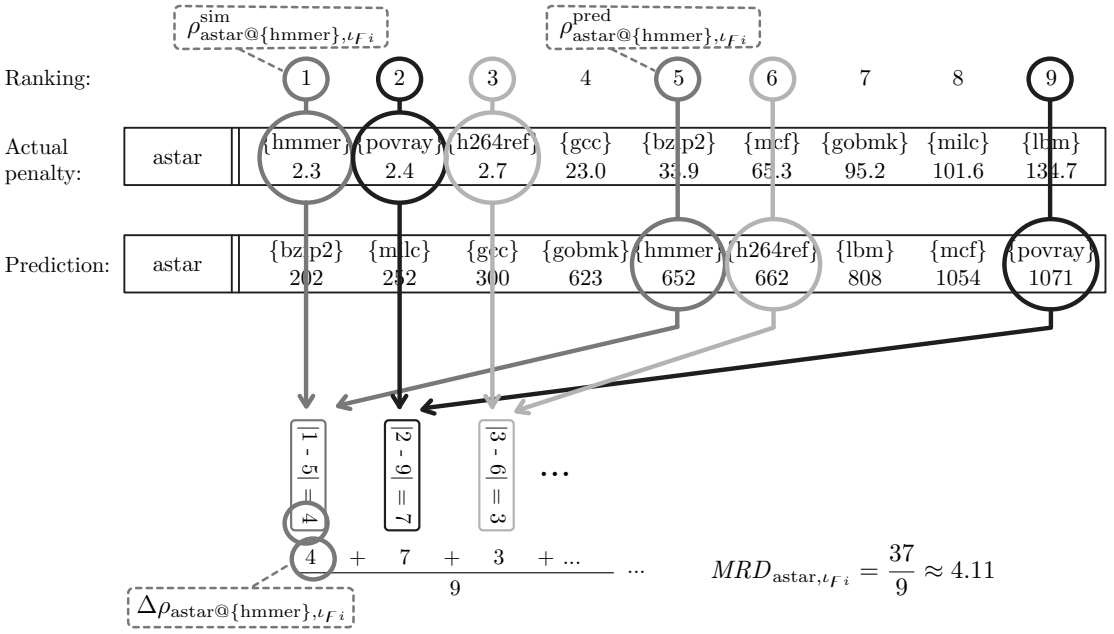


Fig. 4. MRD mean ranking distance value as presented in [1].

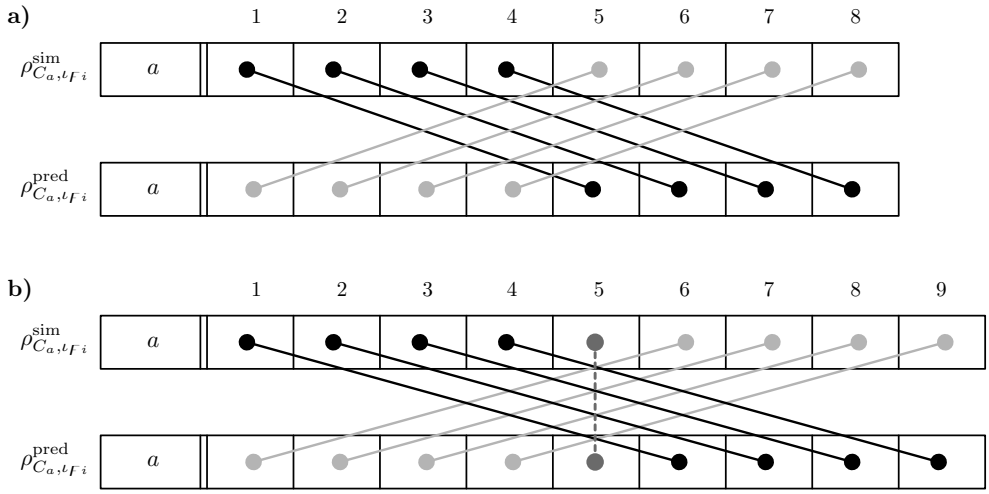


Fig. 5. Determination of the maximum MRD mean ranking distance value for **a)** an even number of candidate co-schedules, and **b)** an odd number of candidate co-schedules. Identical co-schedules in both simulated and predicted rankings are indicated by connecting lines.

ranking performance. As presented in [3] by Fedorova et al., miss based prediction methods showed good prediction accuracy. Further, no significant variance in prediction accuracy has been observed varying either the number of instruction applied for prediction or the amount of applications accessing the shared cache in parallel. The evaluation framework is licensed under GPLv3 and can be downloaded from http://www.ldv.ei.tum.de/cachecontention.

## References

[1] M. Zwick, F. Obermeier, and K. Diepold, "Predicting Cache Contention with Setvectors," in *Proceedings of the International MultiConference of Engineers and Computer Scientists 2010*, S. I. Ao, O. Castillo, C. Douglas, D. Dagan Feng, and J.-A. Lee, Eds. Newswood Limited, Hong Kong, 2010, pp. 244–251.

[2] M. Zwick, M. Durkovic, F. Obermeier, W. Bamberger, and K. Diepold, "MCCCSim - A Highly Configurable Multi Core Cache Contention Simulator," Tech. Rep., 2009.

[3] A. Fedorova, S. Blagodurov, and S. Zhuravlev, "Managing Contention for Shared Resources on Multicore Processors," *Communications of the ACM*, vol. 53, no. 2, pp. 49–57, 2010.

[4] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using OS Observations to Improve Performance in Multicore Systems," *IEEE Micro*, vol. 28, no. 3, pp. 54–66, 2008.

[5] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture," *Proceedings of the 11th International Symposium on High-Performance Computer Architecture, 2005*, pp. 340–351, 2005.

[6] M. Zwick, "Predicting Cache Contention in Multicore Processor Systems," Ph.D. dissertation, Technische Universität München, Apr. 2011.

[7] A. Settle, J. L. Kihm, A. Janiszewski, and D. A. Connors, "Architectural Support for Enhanced SMT Job Scheduling," in *Proceedings of the 13th International Conference of Parallel Architectures and Compilation Techniques, 2004*, Sep. 2004, pp. 63–73.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klausner, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005*, 2005, pp. 190–200.

[9] M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches," in *IEEE Transactions on Computers*, 1989, pp. 1612–1630.

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

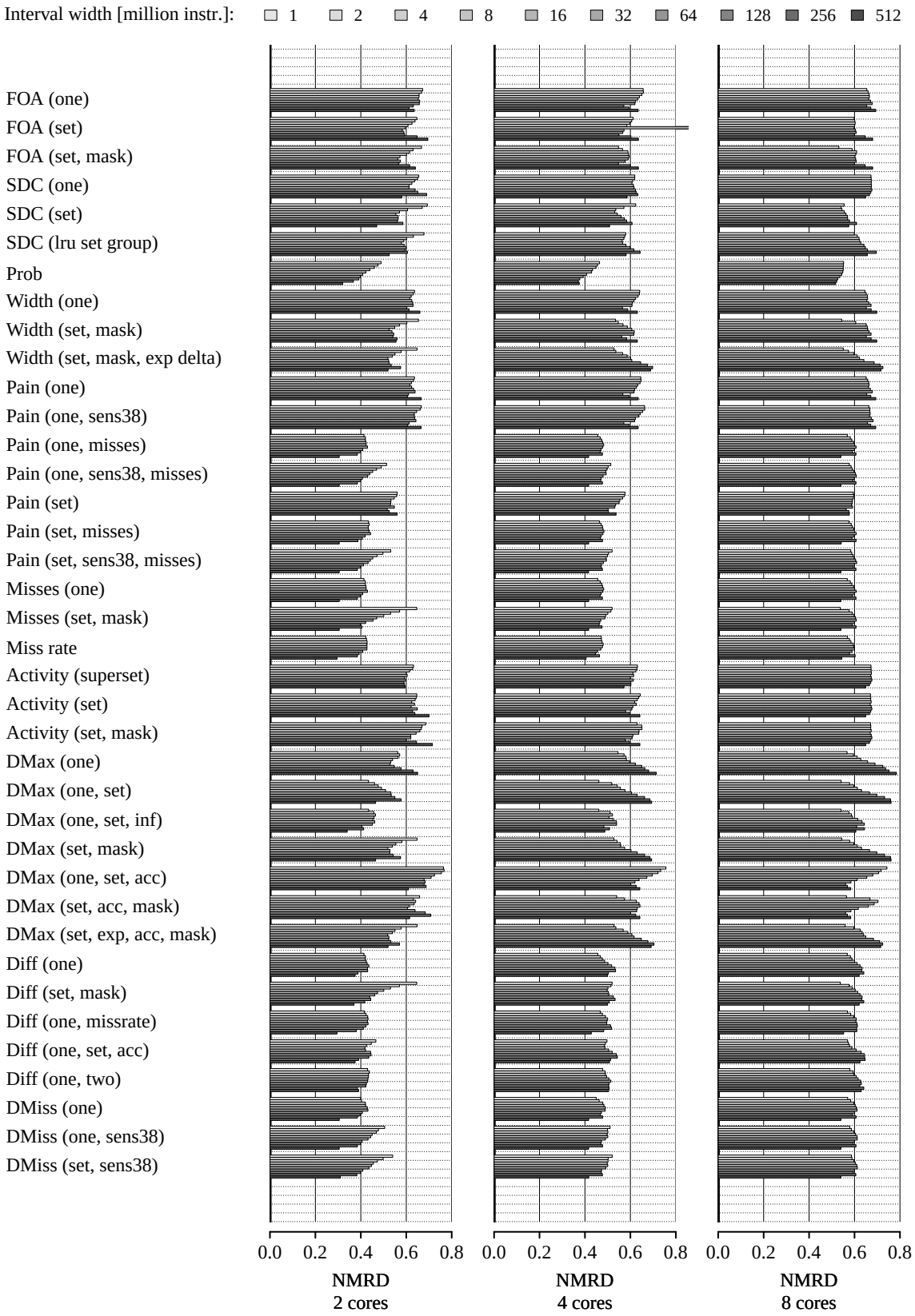


Fig. 6. *NMRD* performance evaluation of several cache contention prediction techniques (left column) for $\psi \in \boldsymbol{\psi} = \{2, 4, 8\}$ processor cores and interval sizes $\mathit{\Gamma} \in \boldsymbol{\Gamma} = \{2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}, 2^{25}, 2^{26}, 2^{27}, 2^{28}, 2^{29}\}$ instructions, averaged over intervals $\iota_{\mathit{\Gamma} i} \in \iota_{\mathit{\Gamma}}$ referring to the first 512 million instructions of every SPEC 2006 benchmark application $a \in A = \{$*astar*, *bzip2*, *gcc*, *gobmk*, *h264ref*, *hmmer*, *lbm*, *mcf*, *milc* and *povray*$\}$.

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

## APPENDIX
## FREQUENTLY USED SYMBOLS

| Symbol | Meaning |
|---|---|
| a | Application |
| $A = \{a_1, a_2, ...\}$ | Set of applications |
| $\|A\|$ | Number of applications in set A |
| act | Actual |
| $C_a$ | Set of applications $\in A\{a\}$ that are co-scheduled to application $a$ |
| $d_{C_a}$ | Degradation that the set of co-schedules $C_a$ introduces to application $a$ |
| $\mathit{F}$ | Interval size; amount of instructions on which the simulations are performed |
| $\boldsymbol{F}$ | Set of interval sizes, e.g. $\boldsymbol{F} = \{2^{20}, 2^{21}, ..., 2^{29}\}$ |
| E | Evaluation metric |
| IPC | Instructions per cycle |
| ITKO | Inter-Thread-Kick-Outs |
| $\mu_a$ | Amount of cache misses of application $a$ on stand-alone execution |
| $\mu_{C_a}$ | Amount of cache misses of application $a$ when $a$ is co-scheduled with $C_a$ |
| NMRD($\psi$, $\mathit{F}$) | Normalized Mean Ranking Difference evaluation function to evaluate general ranking performance |
| $\pi_{C_a,\iota_{\mathit{F}i}}$ | Penalty (amount of cache contention) introduced to application $a$ in execution interval $\iota_{\mathit{F}i}$ when co-scheduling $a$ with candidate co-schedule $C_a$ |
| pred | Predicted |
| $\rho^{pred}_{C_{a,j},\iota_{\mathit{F}i}}$ | Predicted ranking position of candidate co-schedule $C_{a,j}$ |
| $\rho^{sim}_{C_{a,j},\iota_{\mathit{F}i}}$ | Simulated (i.e. actual) ranking position of candidate co-schedule $C_{a,j}$ |
| $t_a$ | Memory access time or program execution time of application $a$ on stand-alone execution |
| $t_{C_a}$ | Memory access time or program execution time of application $a$ when $a$ shares processor resources with applications in $C_a$ |

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)