Predicting Cache Contention in Multicore Processor Systems

Michael J. Zwick

Lehrstuhl für Datenverarbeitung

Technische Universität München

TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Datenverarbeitung

Predicting Cache Contention in Multicore Processor Systems

Michael J. Zwick

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr.-Ing. habil. Erwin Biebl

Prüfer der Dissertation:

1. Univ.-Prof. Dr.-Ing. Klaus Diepold
2. Univ.-Prof. Dr.-Ing. Georg Färber (em.)

Die Dissertation wurde am 22.11.2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 21.03.2011 angenommen.

Contents

Abstract

When several computer program applications are executed in parallel on a multithreaded processor system, they permanently compete for shared resources, one of them being shared processor caches. As some applications interfere much more than others, overall performance in multithreaded processor systems depends on the applications that are chosen to be executed in parallel, i.e. *co-scheduled*. In order to maximize overall performance, future operating systems might *predict cache contention* and co-schedule only those threads that minimize cache interference.

In this thesis, I present several state-of-the-art cache contention prediction methods, variations of them, and completely new methods in a unified framework that makes them both well defined and highly comparable.

I evaluate the methods by means of

- their ability to rank a given set of candidate co-schedules by the amount of cache contention they introduce to an application,
- their ability to select the candidate co-schedule that introduces the least amount of cache contention to an application,
- timing performance, and
- efficiency (gain vs. cost analysis).

My evaluation reveals that – with minimum effort – cache interference of *co-scheduled* applications is best predicted when applying the number of *stand-alone* cache misses as predictor. This is an interesting result, as most state-of-the-art cache contention prediction methods rely on the distribution of references to cache LRU stack positions, stand-alone cache hits as well as the total number of cache references as predictor.

I demonstrate that the accurate prediction of cache contention by stand-alone cache *misses* is caused by high temporal locality of computer program memory references; this can be observed by a highly concentrated distribution of stack distance histogram entries.

1 Introduction

For this thesis, the following notation applies: References in brackets, such as [Zwick et al., 2010], refer to the corresponding publication (cf. chapter *Bibliography*). References without brackets, such as Zwick et al., 2010, or simply Zwick et al., refer to the author(s) of the corresponding publication. References located at the end of a paragraph, behind the last period, apply to the whole paragraph.

1.1 Background

Clock Speed Limitations and Instruction Level Parallelism

During the past 40 years, chip manufacturers improved microprocessor performance primarily by two methods: The first method was simply to raise processor clock speed, and the second method was the introduction of more and more complex processor enhancements to exploit instruction level parallelism. Since such processors are capable to execute more than one instruction per processor clock cycle, they are called *superscalar*. Some years ago, both methods turned out to be a dead end.

When reaching processor clock speeds of about 3 GHz, the effort to further increase processor clock speed became unmanageable due to propagation delays and a too high heat dissipation on a too small area. Besides, methods to exploit single thread instruction level parallelism, such as multi-staged pipelines, out-of-order execution, and dynamic branch prediction for speculative instruction execution, heavily increased processor complexity. But an increased processor complexity, in turn, degraded program execution performance due to higher wire and gate delays. There is a clear reason for the heavy increase of processor complexity: Instruction level parallelism tries to execute as many instructions in parallel as possible, and its efficiency is negatively correlated with data dependency and conditional branches inherent in the executed code. However, as it is a key issue of computer programs that single instructions process results of previously executed instructions (data dependency), and as the probability of branches increases for longer instruction streams, it appears obvious that exploitation of instruction level parallelism has a severe

limitation: *It does not scale.* This limitation is emphasized by Olukotun et al.'s discovery that, for a superscalar processor architecture, the "implementation complexity of the dynamic issue mechanisms and size of the register files scales quadratically with increasing issue width and ultimately impacts the cycle time of the machine" [Olukotun et al., 1996]. In 1997, Hammond et al. stated that the "reliance on a single thread of control limits the parallelism available for many applications, and the cost of extracting parallelism from a single thread is becoming prohibitive" [Hammond et al., 1997] and that to "continue this trend will trade only incremental performance increases for large increases in overall complexity" [Hammond et al., 1997].
To overcome this dilemma and to further increase computer systems performance, *thread level parallelism* had been investigated and exploited.

Thread Level Parallelism

In 1995, Tullsen et al. examined *simultaneous multithreading* (SMT), an extension to wide-issue superscalar processors that allows, besides *instruction level* parallelism, the exploitation of *thread level* parallelism [Tullsen et al., 1995]. To handle multiple threads in parallel, a hardware context has to be provided for each thread: A general purpose register file, a program counter register and other state and control registers. The complex issue queue and execution units however can be shared amongst different threads in order to reduce cost significantly. [Tang et al., 2005]
In 1996, Olukotun et al. discovered that a *single chip multiprocessor* (CMP) architecture with multiple, relatively simple processor cores on a single chip, might achieve a performance boost of 50% up to 100% for applications with *large grained* thread-level parallelism, compared to a superscalar architecture with the same clock speed. For applications featuring *fine grained* thread-level parallelism only, Olukotun et al. observed that a superscalar microarchitecture performs at most 10% better than the CMP architecture, given comparable clock rates. The authors anticipated, however, that the straightforward design of CMP architectures will allow higher clock rates than the complex design of superscalar architectures, eliminating this small performance difference. For applications that *cannot be parallelized*, Olukotun et al. observed CMP performance to lag behind performance of superscalar architectures by about 30%. [Olukotun et al., 1996]

As time went by, chip manufacturers revised their roadmaps: Predictions of a single-chip 10 GHz processor were altered to predictions of processors featuring much lower clock rates, but multiple cores. As an example, Intel CEO Paul Otellini announced a single-chip processor featuring up to 80 cores for *special purpose* applications. [Otellini, 2006]
For *general purpose* desktop and laptop computers, a hybrid SMT-CMP architecture, as it is exemplarily shown in figure 1, has evolved the standard processor design.

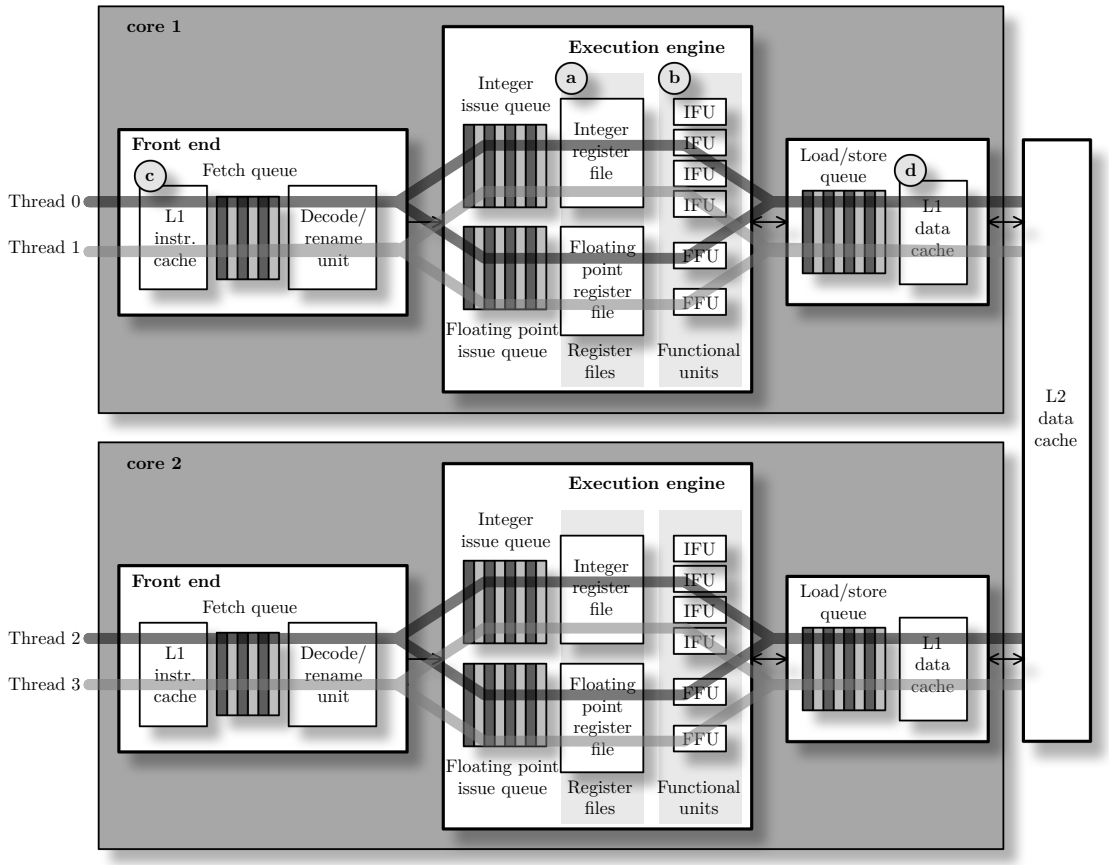


Figure 1: Dual Core CMP of partitioned dual-threaded SMT processors, as it is similarly shown in [El-Moursy et al., 2006]. Each core consists of an integer and a floating point register file (a) to store intermediate data, several integer (IFU) and floating point (FFU) functional units (b) to perform calculations, a level 1 (L1) instruction cache (c) and an L1 data cache (d) that both buffer memory references in order to reduce main memory access time. Within a core, the caches, queues, register files, and functional units can be shared by two threads in parallel; both processor cores share a common level 2 (L2) cache.

With such architectures, chip manufacturers try to satisfy the ever escalating demand for computational power by *parallelization on thread or process basis*. Hereby, they are

strongly assisted by Moore's law that allows to double the number of transistors on a processor about every two years. While the increasing number of transistors led to ever more complex microprocessors in the past, exploiting instruction level parallelism, the huge number of transistors available today is rather spent on additional processor cores and on-die caches than on an ever further increasing processor complexity, exploiting instruction level parallelism. Since Moore's law is still valid today, this approach seems to be a promising way to improve processor computational power significantly.

Processor Memory Gap

One important limitation that does not rely on processor clock speed but on the computational power of the processor is the ever increasing processor memory gap: Although both processor and DRAM (dynamic random access memory) performance grow exponentially over time, the difference between processor and DRAM performance grows exponentially as well. This happens due to the fact that "the exponent for processors is substantially larger than that for DRAMs" [Wulf and McKee, 1995] and "the difference between diverging exponentials also grows exponentially" [Wulf and McKee, 1995]. Therefore, *reducing memory traffic is a key issue regarding processor performance*, and instruction and data caches become more and more important.

Cache Contention

If several applications access identical cache regions in parallel, as it is the case for SMT or CMP processors, and if cache associativity is not large enough, then the applications displace each others' data from the cache in order to fill it with their own. If the displaced data has to be re-fetched, so-called *contention misses* occur.

As an example, figure 2 a) demonstrates cache contention misses introduced in a timeslice 3 to an application a_1, regarding *timesliced multithreading* on a single core processor; figure 2 b) depicts cache contention misses introduced in timeslice 3 to application a_1 for *timesliced and simultaneous multithreading* on a processor with 2 execution units sharing the same cache. In figure 2 a), timeslice 2, application a_2 displaces two cache lines that have been fetched by application a_1 in timeslice 1 (black dashed lines). In timeslice 3, application a_1 has to re-fetch both cache lines (grey dashed lines). This introduces addi-

tional penalty compared to stand-alone execution, i.e. when solely executing a_1 in each timeslice. In figure 2 b), timeslice 2, application a_2 again displaces two cache lines of a_1, introducing the same penalty as in figure 2 a). However, this time, there are further applications a_3 and a_4 sharing the same cache. In timeslices 1 and 2, applications a_3 and a_4 displace cache lines of a_1 that additionally have to be re-fetched by a_1 in timeslice 3. In timeslice 3, however, a_3 is also executed *in parallel* to a_1 and the amount of grey dashed lines indicates that this co-scheduling introduces many contention misses. However, still the question remains why there are so much more contention misses introduced from the application that is executed *in parallel*, compared to contention misses introduced from applications that have been executed on a previous timeslice. The answer is as follows: If

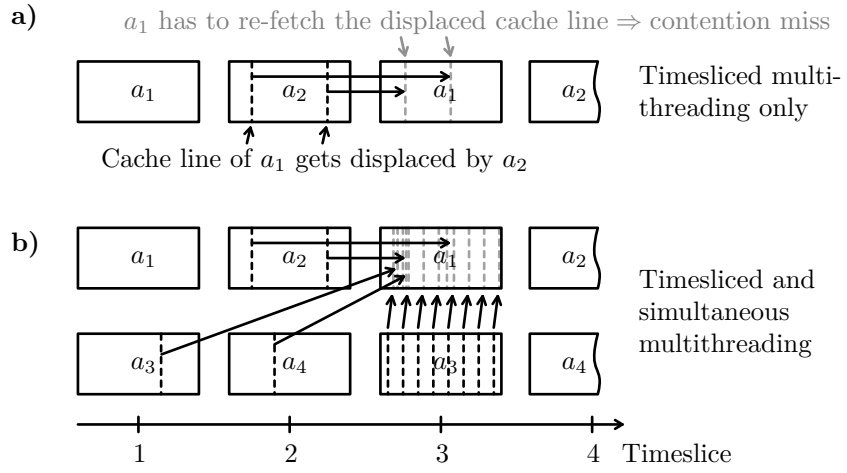


Figure 2: Inter-application cache contention misses on a) a single core processor applying timesliced multithreading, and b) a dual core CMP applying timesliced and simultenous multithreading.

a_3 displaces an entry of a_1, then a_1 has to re-fetch it and might hereby displace an entry of a_3. If a_3 then, again, re-fetches the entry that has been displaced by a_1, a_3 might again displace an entry of a_1 etc. These displace and re-fetch cycles might occur frequently on simultaneous multithreaded systems if applications that are executed in parallel share the same cache. However, if multithreading is performed on *timeslice basis* only, displace and re-fetch cycles (per cache line) can only occur at a maximum rate of 1 per 2 timeslices: If, in figure 2 a), application a_2 displaces an entry of a_1 in timeslice 2, then this entry can at the earliest be re-fetched at the beginning of timeslice 3. Even if this re-fetch displaces

an entry of a_2 from the cache, a_2 can re-fetch this entry not before the beginning of time-slice 4. As a consequence, sharing a cache in parallel has the potential to introduce far more inter-thread cache contention misses than timesliced multithreading.

Figure 3 shows L2 cache hitrate of the SPEC 2006 test benchmark *milc* for the case that *milc* is executed stand-alone (topmost bold black curve), and for the case that *milc* is co-scheduled with each of the applications *astar*, *gcc*, *bzip2*, *gobmk*, and *lbm*. Obviously, there are applications that have a limited degradation effect, such as *astar* or *gcc*, while there are others that have more serious impact, like *lbm*.

Given this information, it seems obvious that operating systems, that currently apply basic load balancing only when (co-)scheduling applications [Knauerhase et al., 2008], could achieve a much better (co-)scheduling performance if they would account for cache contention. Therefore, accounting for cache contention is one optimization criterion of the job shop scheduling multicriteria optimization problem regarding SMT and CMP processor architectures with (partially) shared caches. In the following, I refer to (execution intervals of) applications as *co-scheduled* if they are executed *in parallel* and *simultaneously* share a common cache.

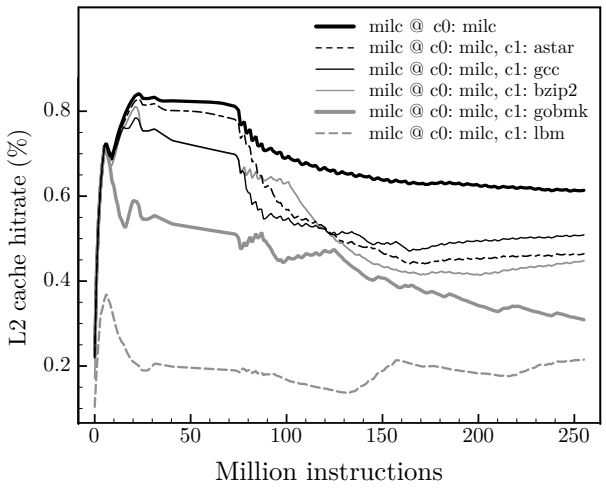


Figure 3: L2 cache hitrate degradation introduced to application *milc* when co-scheduling *milc* with each of the applications *astar*, *gcc*, *bzip2*, *gobmk*, and *lbm* on a CMP architecture; simulation has been performed applying the MCCCSim simulator [Zwick et al., 2009a].

Cache Contention Aware Co-Scheduling

Cache contention aware co-scheduling minimizes penalties introduced from cache contention by

- a proper mapping of applications to processor cores, and
- an appropriate scheduling of the applications,

ensuring that only those applications get co-scheduled, i.e. executed in parallel, sharing a common cache, that compete as little as possible. For this minimization, cache contention has to be *predicted*. Therefore, some predictors have either to be read from memory (if the predictors are, for example, delivered with the binary), or from special purpose performance registers (if the predictors are determined at runtime). Given the predictors, some computational performance has to be spent to calculate several predictions and select the best. If gain in memory hierarchy performance is higher than degradation of memory hierarchy performance introduced from reading predictors and calculating predictions, then cache contention aware co-scheduling can be regarded as a way to *transform computational performance into memory hierarchy performance*. To illustrate this idea, I present figure 4 that shows computational performance and memory hierarchy performance of an application a that is being co-scheduled with another application.

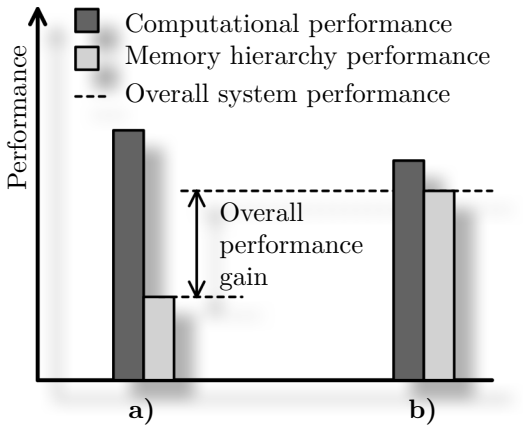


Figure 4: Computational performance and memory hierarchy performance applying a) standard co-scheduling and b) cache contention aware co-scheduling.

In figure 4 a), memory hierarchy performance of application *a* is heavily degraded from cache contention; if application *a* is memory bound, then its overall performance is restricted by its memory hierarchy performance. In part b) of the figure, however, memory hierarchy performance of application *a* is increased by cache contention prediction and proper application co-scheduling. As the prediction will cost some computational performance, the computational performance of application *a* achieved in a) is higher than in b). If the overall performance of application *a* is memory bound, however, this might be without any effect on overall performance. Note that I assume applications to be rather memory bound than bound on computational performance in the future, as they will likely be programmed to fully exploit the high amount of parallelism available in SMT/CMP computer systems.

If the gain in memory hierarchy performance, achieved from proper co-scheduling, is higher than the degradation of memory hierarchy performance that has been introduced by the prediction, then there is an overall *memory hierarchy* performance gain. Therefore, application *a* shows a higher memory hierarchy performance in figure 4 b) than in figure 4 a). If performance of application *a* is memory bound, then this higher memory hierarchy performance of *a* will result in an *overall performance gain* of application *a*.

Therefore, if application performance is *memory bound*, which is quite likely for SMT or CMP systems, then cache contention aware co-scheduling allows to *transform computational performance into memory hierarchy performance.*

Coupling memory hierarchy performance with computational performance in order to let memory hierarchy performance also benefit from improvements in computational performance seems to be a way to attenuate the growth of the processor memory gap, and increase overall system performance.

The Ideal Cache Contention Prediction Method

In order to achieve the best overall performance gain, an ideal method has to predict cache contention *as accurately and as fast as possible*, applying a *minimum set of resources* (computational and memory performance, size of predictors) at the same time.

The *most accurate prediction* will apparently be performed if a method would be able to exactly predict the number of additional cache misses that a set of applications $C_a^\psi = \{c_{a,1}, c_{a,2}, \dots c_{a,\psi-1}\}$, named (candidate) co-schedule, introduces to an application a on a processor architecture that can execute ψ threads in parallel, sharing the same cache. Adding both additional cache misses the applications in C_a^ψ introduce to a and the misses a introduces to the applications in C_a^ψ, you get the total number of cache misses that will be introduced from co-scheduling a and the applications in C_a^ψ. If such a number is determined for all applications a in a set of applications A and all possible co-schedules $\{C_{a,1}^\psi, C_{a,2}^\psi, \dots\}$, $C_{a,j}^\psi \in A \backslash \{a\}$, then the set of applications in A that minimizes overall cache contention can easily be determined.
Nevertheless, in many cases a prediction method might be sufficient that determines a candidate co-schedule *ranking*, i.e. sorts the set of all possible candidate co-schedules $\{C_{a,1}^\psi, C_{a,2}^\psi, \dots\}$ of an application a by the amount of cache misses they introduce to a.
There might also be many cases that suffice a method that selects that candidate co-schedule $C_{a,j}^\psi$ from the set of all possible co-schedules that minimizes the contention introduced to an application a.
Cache contention prediction methods should generally be able to predict performance of co-scheduled applications by characteristics obtained from the *particular* applications, and not from characteristics obtained from co-scheduled applications. Otherwise, each combination of applications would have to be executed in order to determine co-schedule performance, as it is exemplarily the case in [Snavely and Tullsen, 2000], rendering prediction infeasible if the set of applications A is not very small.

1.2 State-of-the-Art Methods and their Limitations

Several cache contention prediction methods have been proposed in the past.

In [Chandra et al., 2005], the authors introduce two heuristics and one probabilistic approach to predict cache contention. The heuristic methods are based on the estimation of a 'reduced cache associativity', determined by cache access frequency and the amount of references to the various LRU stack positions; the probabilistic method calculates for each stand-alone cache hit the probability to become a miss under cache sharing. The authors evaluate prediction accuracy of their methods by calculating the difference between additional cache misses that were *predicted*, and the *actual* additional cache misses observed using a simulator. An evaluation regarding timing performance is not performed. Sometimes, the description of the methods turns out to be ambiguous. For example, it is sometimes not clear, which calculations have to be performed per cache set, and which have to be performed only once, using average per-cache-set data.

In [Settle et al., 2004], the authors describe a method to predict cache contention that tries to exploit spatial locality. They cluster cache sets to groups and indicate for each group by a single bit, if it suffers from *many* accesses/misses, or just a *few*. However, from their publication, you cannot extract the cutoff frequency of accesses/misses they apply to distinguish between *many* or *few* accesses/misses. A further publication of a co-author however brings up an evaluation of several cutoff measures and it turns out that good prediction results can be achieved within the "third quartile used as the cutoff" [Kihm and Connors, 2004]. But you cannot extract if this result has been applied in [Settle et al., 2004], or not. In order to evaluate prediction accuracy, the authors integrate their prediction method into a scheduler and compare program execution time achieved with the modified scheduler to program execution time measured when applying standard round-robin priority scheduling. An evaluation of the time necessary to perform a prediction is not performed.

In [Knauerhase et al., 2008], the authors investigate methods to improve scheduling and load balancing on multicore processors. Knauerhase et al. state that they "explored various definitions of cache weight to find the most useful input to scheduling policy [... and in] the end, our experimentation found that cache misses per cycle are the best indication of

cache interference" [Knauerhase et al., 2008]. However, a traceable analysis of the "various definition of cache weight" is not presented. To evaluate the applicability of the cache misses approach, they perform adaptions to a scheduler and compare execution times to those achieved by a non-adapted scheduler. An evaluation regarding the time necessary to perform co-scheduling predicitons is not presented.
In [Fedorova et al., 2010], the authors propose a method based on the *sensitivity* of an application to suffer from contention misses, and the *intensity* of an application to introduce cache misses to another application. They evaluate their method in relation to two other state-of-the-art prediction methods by comparing performance degradation of *predicted* best co-schedules to that of *actual* best co-schedules. A performance evaluation regarding *prediction time* is not performed.

In the following, I present the most severe limitations of previous publications:

- There are cache contention prediction methods that have not been well-defined; as a consequence, you sometimes do not exactly know what the published results actually express.
- Often, cache contention prediction methods have not been compared to one another, but to a ground truth measure. As most proposals, however, applied *different* ground truth measures that are not comparable to one another, it is infeasible to extract accuracy of the methods *in relation to one another* from the published results. Further, all evaluations have been performed considering individual processor architectures, applying different sets of input data (memory references), individual sizes of input data, and different performance measures. As a consequence, an evaluation of state-of-the-art cache contention prediction methods *in relation to one another* cannot be performed from the given results.
- Besides the ambiguous definition of cache contention prediction methods and the application of various uncomparable evaluation settings, the applied *evaluation process* has often not been defined properly as well, rendering the results untraceable.

- Previous publications evaluate cache contention prediction methods primarily by means of prediction *accuracy* and rarely perform a *timing* analysis. Sometimes, authors state that the time to perform a prediction is very small and can be disregarded, which seems to be an information of limited use. As previous publications spent only minimum effort, if any, regarding timing of cache contention prediction methods, it is obvious that state-of-the-art cache contention prediction methods cannot be compared to one another by means of timing performance at all.
- Generally lacking a timing evaluation, a gain-cost analysis of cache contention prediction methods cannot be extracted from published evaluation results.
- A unified description of state-of-the-art cache contention prediction techniques, applying the same notation, is completely missing; this makes it hard to see similarities and differences between the various methods, although this would be a good starting point to extract performance information of the *underlying characteristics* of the various methods.

As you can see from this list, it is impossible to determine a performance ranking of cache contention prediction methods, i.e. to evaluate cache contention prediction methods in relation to one another. It is even impossible to determine the best performing method, i.e. the method that best approximates an ideal prediction method.

1.3 Formulation of Research Problem

The research problem addressed in this thesis is an

- *analysis and precise reprocessing of state-of-the-art cache contention prediction methods*, the
- *introduction and proper definition of new cache contention prediction methods* that do not belong to state-of-the-art methods yet, and an
- *evaluation of the presented methods* regarding prediction *accuracy* and *timing* performance.

As it has not been common with most other publications regarding cache contention prediction, this thesis focuses on an *unambiguous*, *precise* and *at any time replicable* definition and evaluation of cache contention techniques. Only if these conditions are met, a high credibility of the presented results can be guaranteed, rendering the presented results reasonable to be used by others in the future.

Evaluation Objectives

In order to evaluate cache contention prediction methods, the methods should be evaluated in relation to one another regarding their

- *ability to rank the elements in a set of candidate co-schedules* $\{C^{\psi}_{a,1}, C^{\psi}_{a,2}, \dots\}$ by the amount of additional cache misses they introduce to an application a, their
- *ability to select the candidate co-schedule* $C_{a,j}$ from a set of candidate co-schedules that introduces the least cache misses to an application a,
- *timing performance*, i.e. the time necessary to perform a prediction, and
- *prediction efficiency* (gain vs. cost analysis).

Further, the evaluation should

- *reveal performance of the underlying characteristics of a method* (cache accesses, hits, misses, ...), and it should
- point out *why methods of some type perform better than others.*

Evaluation Preferences

As underlying condition, methods should be evaluated on

- *different sets of execution interval sizes* to determine if a method performs better on a smaller or larger amount of instructions. For this task, I define a minimum interval size of $I = 2^{20}$ and a maximum interval size of $I = 2^{29}$ instructions to be sufficient. Further, the evaluation should be performed considering
- 2-, 4- *and* 8*-fold parallelism* ψ in order to investigate the methods' ability to predict cache contention in cases where there are many references contending for the shared cache in parallel, or just a few.

At all times,

- *evaluation should be unambiguous and traceable* in order to make the results re-usable by others. Note that this requirement also requests that
- *a precise definition of the applied ground truth reference* has to be performed.

Method Description

In order to make the evaluation meaningful and reproducible, the

- *applied cache contention prediction techniques have to be defined unambiguously*; besides *new* methods, this also includes *state-of-the-art* methods.

To make cache contention techniques better comparable to one another and to enable an easy detection of the underlying characteristics of each method,

- *methods should be described uniformly.* Therefore, *a single equation* should be specified to define, for each method, how the prediction of additional cache misses introduced from a set of applications $C_a^{\psi} = \{c_{a,1}, c_{a,2}, \dots\}$ to an application a in an execution interval of size I is performed.
  If possible, the variables contained in this single equation should easily be traceable to the elements of a so-called stack distance histogram.

In order to make timing performance *reproducible*, the

- *predictors applied in the calculation of the prediction should be specified.* This way, the calculation of the predictions will become much more transparent and it will be easy to see which calculations are performed at *runtime* and add to timing performance, and which will not.

To make timing performance *comparable*, the

- *applied predictors should ideally include all calculations that can be performed in advance to runtime*, i.e. before knowing which applications are about to be co-scheduled.

Limitations

For the evaluation, I define the following limitations to be appropriate:

- *The evaluation exclusively incorporates data references*, as it has been done in [Huffmire and Sherwood, 2006]; *instruction* references are omitted and effects arising from *instruction* $\leftrightarrow$ *instruction* or *instruction* $\leftrightarrow$ *data* interference are not included in the evaluation. Note that this limitation *increases* evaluation accuracy: State-of-the-art cache contention prediction methods incorporate only *one* memory reference stream (e.g. data); effects from a *second* stream (e.g. instructions) are not modelled. A ground truth measure, however, *would* model effects arising from another interference source. But if the assumptions incorporated in the prediction model do not match the assumptions incorporated in the ground truth model, then errors caused by this inconsistent modelling would overlay prediction errors and render the evaluation less accurate.

- *Processor caches process virtual addresses* rather than physical addresses; this definition makes contention between co-scheduled applications more related to cache misses than to TLB (translation look-aside buffer) misses. This is a reasonable approach, as state-of-the-art cache contention prediction methods are modelled to predict *cache* contention, and not contention introduced from additional *TLB* misses.

- *There do not occur any waitstates caused by busy resources* such as busses, shared caches etc., and co-scheduling penalties are defined to exclusively originate from displacing already fetched items from a shared cache.
- *Timeslices are of infinite length.* As state-of-the-art cache contention prediction methods generally do not incorporate timeslicing in their models, this assumption ensures that a ground truth reference does only incorporate characteristics of those applications that are also included in the prediction model.
- *Memory access time and cache hit time to any address is constant at any time.* This means, in particular, that there is no timing difference between two consecutive memory accesses to same or different memory banks.
- *Caches and TLBs apply LRU (least recently used) replacement policy*, as it is common in many processor systems.
- *Caches apply write back policy*; multi-level cache inclusion property (cf. [Baer and Wang, 1988]) can be disregarded.
- *Each application is single threaded*, and memory references that are processed in parallel exclusively originate from different applications.
- *There is no data sharing among applications*; therefore, applications can only suffer a performance *degradation* from co-scheduling; an overall performance *improvement* due to inter-application prefetching effects cannot occur.

Let $\iota_{F i}$ be one execution interval in the set of execution intervals $\iota_F = \{\iota_{F1}, \iota_{F2}, \ldots \iota_{F|\iota_F|}\}$, let $|\iota_F|$ be the amount of intervals $\iota_{F i}$ in ι_F, and let each $\iota_{F i} \in \iota_F$ represent the number of F instructions.

- *The contents of a cache at the beginning of an execution interval* $\iota_{F i}$ is identical to the contents of the cache at the end of interval $\iota_{F i-1}$ if the same set of applications has been executed in both intervals, what is the case for the applied ground truth measure. At the beginning of interval $\iota_{F 0}$, the cache is empty.
- *Inter-interval effects*, such as cache contention introduced in an interval $\iota_{F i}$ from other intervals $\iota_{F j}$, $j \neq i$, are not regarded in the prediction.

1.4 Contributions

Precise Definition of Cache Contention Prediction Techniques

In this thesis, I introduce a new and consistent notation to unambiguously describe cache contention prediction techniques. Applying this notation, I present several *state-of-the-art* techniques and introduce *variations* of them to investigate, in particular, whether or not

- applying cache set granularity would achieve better prediction results,
- different weightings of stack distance histogram entries are favorable,
- prediction accuracy is correlated rather with stack distances observed on cache LRU stack accesses, cache misses, or the total number of cache accesses.

Besides those state-of-the-art methods and their variations, I further present and investigate some *new* cache contention prediction methods. Hereby, the applied notation makes it easy to reveal similarities and differences between the various methods. To ensure traceable timing analysis, I specify which calculations have to be performed at runtime (prediction), and which do not (predictor).
To the best of my knowledge, this thesis is unique in specifiying such a large number of cache contention prediction techniques both such uniformly and precisely at the same time.

Unambiguous Definition of the Evaluation Process

Besides a proper definition of cache contention prediction techniques, I present the whole *evaluation process* I apply with mathematical definiteness, including

- the *evaluation measures and their application*, even for various lengths of prediction intervals and various number of processor cores,
- the way I generate the ground truth reference,
- the input data I apply to the predictions and how they are obtained.

As *evaluation measures*, I apply

- the ability of a prediction method to rank candidate co-schedules $\{C^{\psi}_{a,1}, C^{\psi}_{a,2}, \dots\}$ by the amount of cache contention they introduce to an appliciation a,
- the ability of a prediction method to select that candidate co-schedule $C^{\psi}_{a,j}$

from a set of candidate co-schedules $\{C^{\psi}_{a,1}, C^{\psi}_{a,2}, \dots\}$ that introduces *least* cache contention to an application a,

- the time necessary to perform a prediction, and a
- gain vs. cost analysis.

To the best of my knowledge, neither has the process of evaluating cache contention predictions methods ever been described such precisely and extensively before, nor has a gain vs. cost analysis of cache contention prediction methods been performed so far.

Evaluation Results

Amongst others, my evaluations demonstrate that

- cache set granularity is not required to achieve good prediction results,
- an enhanced weighting of stack distance histogram entries does not achieve significant improvements, as
- *cache contention prediction is best performed by stand-alone cache misses*;
- *cache accesses* as well as *distribution of LRU stack distances are of limited use* regarding cache contention prediction due to *high temporal program locality*, although most state-of-the-art prediction methods rely on these measures.

Note that the last two points are the most significant result of this thesis. They support and further explain the surprising observation in [Fedorova et al., 2010] that the application of *stand-alone cache misses* is generally one of the best ways to predict cache contention introduced from *parallel* application execution.
For an explanation *why* stand-alone cache misses turn out to be superior regarding cache contention prediction, and *why* the amount of cache accesses or stand-alone cache hits is a rather poor predictor, see

- section 2.5, method *Width*, variation 'set mask exp delta',
- section 2.6, method *Pain*, variation 'one, misses', and
- section 3.2, *General Ranking Performance*, subsections
  - 'Poor performance of access or hit based methods' and
  - 'Good performance of miss based and related methods' and the
  - 'Big Picture'.

1.5 Overview

The remainder of this thesis is organized as follows:
In *chapter 2*, I present a uniform description of state-of-the-art cache contention prediction methods, extensions to those methods, and completely new methods.

To evaluate the presented techniques, I apply *chapter 3*. In this chapter, I define evaluation measures, their application, and present the evaluation results achieved.

Chapter 4 concludes this thesis.

In the *appendix*, I provide

- a *cache glossary* that defines the applied terminology regarding processor caches,
- several stack distance histograms that reveal temporal activity and reuse behavior of the SPEC 2006 test benchmark applications I apply in my evaluation,
- a short overview of data distributions of the averaged evaluation results presented in chapter 3, and
- a *list of symbols and abbrevations*.

2 Techniques to Predict Cache Contention

Many techniques to predict cache contention have been proposed in the past. Needless to say, most of the methods have been proposed using their own nomenclature. As a consequence, comparing the various methods makes it hard to extract similarities and differences. However, this has not been a big issue yet, as an extensive comparison of cache contention prediction methods has not been performed so far. This thesis, however, not only compares several state-of-the-art methods in relation to one another, but also introduces many variations to them and further proposes new methods. The amount of cache contention prediction techniques defined and evaluated in this thesis exceeds all other publications by far. In order to make the description of the techniques, despite their amount, as comprehensible as possible, I apply a homogenous description that is mostly based on stack distance histograms. This allows an easy extraction of similarities and differences from the various methods, making it simple to extract the methods' underlying characteristics, such as stand-alone cache misses, for example.

In the past, cache contention prediction methods have not always been described properly, introducing inconsistencies and ambiguities. In order to achieve a most precise definition of cache contention prediction methods, I specify an equation $p_{C_a,\iota_F i}$ for each method to determine the way a method predicts cache contention with mathematical definiteness. Each equation of $p_{C_a,\iota_F i}$ defines how the corresponding method predicts the penalty a set of co-scheduled applications $C_a = \{c_{a,1}, c_{a,2}, \ldots c_{a,|C_a|}\}$ introduces to an application $a \notin C_a$ in interval $\iota_F i$, i.e. when executing instructions $i \cdot F \ldots (i+1) \cdot F - 1$. Generally, higher values of $p_{C_a,\iota_F i}$ denote higher penalty, lower values denote lower penalty. Note that absolute values of $p_{C_a,\iota_F i}$ have a specific meaning such as 'additional cache misses' only in some special cases. Instead, values of $p_{C_a,\iota_F i}$ have to be compared in relation to one another in order to evaluate the predicted performance of one co-schedule to another. Comparing values of $p_{C_a,\iota_F i}$ to one another is meaningful only if they were calculated by the same method and the same variation. In order to *compare various techniques to one another*, $p_{C_a,\iota_F i}$ has to be used as input to the evaluation processes presented in chapter 3. In order to make the evaluation of *timing performance* replicable, I specify for each pre-

diction $p_{C_a,\iota_{F\,i}}$ the *predictors* I apply to calculate the prediction. *Predictors* are calculated *in advance* to the prediction and do not contribute to the amount of time necessary to perform a prediction at runtime; the time to perform a prediction, however, is significant for timing performance. With this partitioning, it can easily be extracted which calculations are performed *before* runtime and are encapsulated in predictors and therefore *do not* contribute to prediction time, and which calculations have to be performed at runtime and *do* contribute to execution time. Without that knowledge, evaluation results regarding timing performance would be of limited use.
In order to achieve a fair evaluation of timing performance, predictors are designed to include all calculations that can be performed *before* runtime, i.e. without any knowledge of the applications that are about to be co-scheduled.

I apply the following notation regarding 'methods' and 'variations': I refer to the *key idea* used for prediction as 'method' and apply term 'variation' to differ between different versions of a method, even if a version is identical to an original state-of-the-art method that has already been proposed by another author. At any time, however, I will make clear which variations represent state-of-the-art techniques, an which do not. As the variation name is used to describe special characteristics of a method, this makes it easy to remember implementation differences when comparing the results in chapter 3.

2.1 Stack distance histograms

In this section, I introduce so-called *stack distance* histograms; I apply stack distance histograms in order to define cache contention prediction methods.

Stack distance histograms have been proposed in 1970 by Mattson et al. in order to investigate virtual memory systems [Mattson et al., 1970]. Since that time, the concept has also been applied to analyze cache memory systems, cf. [Hill and Smith, 1989].

In this context, they summarize program memory access patterns by giving an overview of the amount of references to cache lines of a cache set that have most recently, second most recently, ... α most recently and $> \alpha$ most recently been accessed; α is the associativity of the cache. The term *stack distance* relates to the *distance* δ observed on the LRU *stack* of a cache set when accessing a cache line. See figure 40 in the appendix for an explanation of *cache set* or *cache line*.

Figure 5 a) shows an LRU stack ζ_s^S for a cache set s in case associativity $\alpha = 8$.

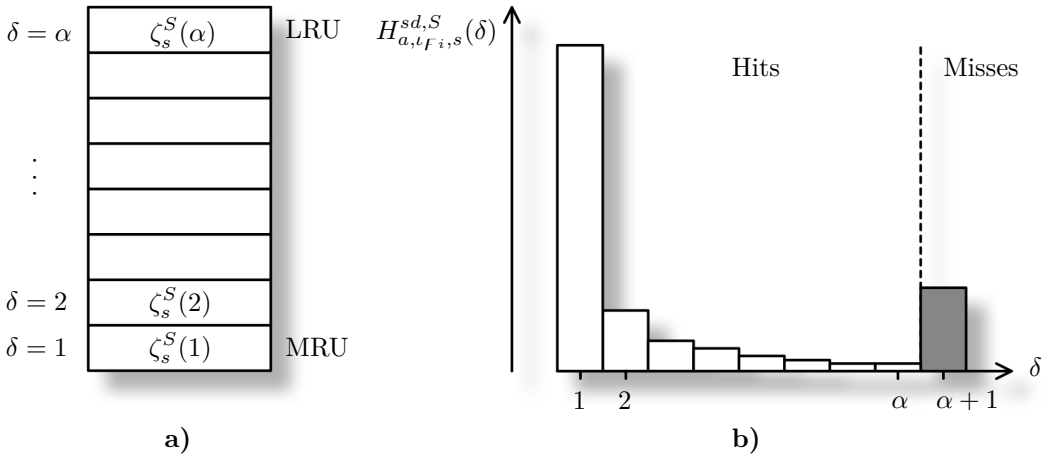


Figure 5: a) LRU stack ζ_s^S for cache set s; $\delta = 1$ depicts the most recently used (MRU) element, and $\delta = \alpha = 8$ the least recently used (LRU) element; **b)** stack distance histogram $H^{sd,S}_{a,\iota_{Fi},s}$ for application a, cache set s, execution interval ι_{Fi}. Note that $H^{sd,S}_{a,\iota_{Fi},s}$ denotes a stack distance histogram for a specific cache set s.

Let $S = \{s_1, s_2, \ldots s_{|S|}\}$ be the set of cache sets of a processor cache and let $|S|$ be the number of cache sets. Generally, a cache with $|S|$ cache sets and associativity α applies $|S|$ LRU stacks $\zeta^S = \{\zeta_1^S, \ldots \zeta_{|S|}^S\}$ of capacity α each in order to track cache line references and to determine which element to displace next.

Definition: Let ζ be a stack. If it is said that stack ζ is of capacity $\alpha, \alpha \in \mathbb{N}^+$, then ζ can hold up to α numbers $\in \mathbb{N}$. $\zeta(\delta)$, $1 \leq \delta \leq \alpha$, is the operation that references stack element at position δ (cf. figure 5 a)). $\zeta\{x\}$ is the operation that returns the stack position of element x according to

$$\zeta\{x\} = \begin{cases} \delta, & \text{if } \exists\ \delta, 1 \leq \delta \leq \alpha :\ \zeta(\delta) = x \\ \alpha + 1, & \text{if } \nexists\ \delta, 1 \leq \delta \leq \alpha :\ \zeta(\delta) = x \end{cases} \tag{1}$$

If there is *more than one* δ with $\zeta(\delta) = x$, the least valued δ is returned. □

Figure 5 b) shows a per-cache-set stack distance histogram $H^{sd,S}_{a,\iota_{\digamma i},s}$ for application a, interval $\iota_{\digamma i}$, cache set s, associativity $\alpha = 8$. $H^{sd,S}_{a,\iota_{\digamma i},s}(\delta)$ refers to element δ in stack distance histogram $H^{sd,S}_{a,\iota_{\digamma i},s}$ and represents the number of references of application a, execution interval $\iota_{\digamma i}$ that map to cache set s and address the δ most recently referenced cache line. As figure 5 b) suggest, $H^{sd,S}_{a,\iota_{\digamma i},s}(\alpha+1)$ represents the number of cache misses of application a in intervall $\iota_{\digamma i}$, cache set s. For application a in interval $\iota_{\digamma i}$, the number of cache hits in cache set s calculates to $\sum_{\delta=1}^{\alpha} H^{sd,S}_{a,\iota_{\digamma i}}(\delta)$, and the total number of references to cache set s calculates to $\sum_{\delta=1}^{\alpha+1} H^{sd,S}_{a,\iota_{\digamma i},s}(\delta)$. Contrary to per-cache-set histograms $H^{sd,S}_{a,\iota_{\digamma i},s}$, stack distance histogram $H^{sd}_{a,\iota_{\digamma i}}$ summarizes the amount of distances of *all* cache sets, i.e.

$$H^{sd}_{a,\iota_{\digamma i}}(\delta) = \sum_{s=1}^{|S|} H^{sd,S}_{a,\iota_{\digamma i},s}(\delta)\ , \quad 1 \leq \delta \leq \alpha + 1, \tag{2}$$

and the number of cache hits, cache misses, and cache accesses calculate accordingly.

Algorithm 1 shows how to calculate both the set of per-cache-set stack distance histograms $H^{sd,S}_{a,\iota_{\digamma i}} = \left\{H^{sd,S}_{a,\iota_{\digamma i},1}, \ldots H^{sd,S}_{a,\iota_{\digamma i},|S|}\right\}$ and the combined histogram $H^{sd}_{a,\iota_{\digamma i}}$ from a given tuple of memory references $M_{a,\iota_{\digamma i}} = (m_{a,\iota_{\digamma i},1}, \ldots m_{a,\iota_{\digamma i},|M_{a,\iota_{\digamma i}}|})$ that an application a references when it executes instructions $\digamma \cdot i \ldots \digamma \cdot (i+1) - 1$. $|M_{a,\iota_{\digamma i}}|$ is the number of references in $\iota_{\digamma i}$. Each reference $m_{a,\iota_{\digamma i},j} \in M_{a,\iota_{\digamma i}}$ is a natural number of range $0 \leq m_{a,\iota_{\digamma i},j} < 2^{32}$.

In order to *extract the cache set address* of a memory reference m, I apply operation $\varsigma(m)$. Given a cache of way size $|w|$ and line size $|\lambda|$ (cf. figure 40 in the appendix), then $\varsigma(m) = ((m/|\lambda|)\ mod\ (|w|/|\lambda|)) + 1$; note that $1 \leq \varsigma(m) \leq |S|$; $|w|$ and $|\lambda|$ are measured in units of byte.

To *extract the key address* of a memory reference m, I apply operation $\kappa(m)$. Given a cache of way size $|w|$, then $\kappa(m) = m/|w|$ and $\forall_m : \kappa(m) \geq 0$.

Algorithm 1 Generating stack distance histograms $H^{sd,S}_{a,\iota_{Fi},s}$ and $H^{sd}_{a,\iota_{Fi}}$ for interval ι_{Fi} of an application a and an α way set associative cache with $|S|$ cache sets.

1: # —— Initialization ——
2: **for** $\delta \leftarrow 1$ to $\alpha + 1$ **do**
3: $H^{sd}_{a,\iota_{Fi}}(\delta) \leftarrow 0$
4: **for** $s \leftarrow 1$ to $|S|$ **do**
5: $H^{sd,S}_{a,\iota_{Fi},s}(\delta) \leftarrow 0$
6: **if** $\delta \leq \alpha$ **then**
7: $\zeta^S_s(\delta) \leftarrow -1$
8: **end if**
9: **end for**
10: **end for**

11: # —— Process all references in $M_{a,\iota_{Fi}}$ ——
12: **for** $j \leftarrow 1$ to $|M_{a,\iota_{Fi}}|$ **do**
13: $s \leftarrow \varsigma(m_{a,\iota_{Fi},j})$
14: $\delta \leftarrow \zeta^S_s\{\kappa(m_{a,\iota_{Fi},j})\}$
15: $H^{sd}_{a,\iota_{Fi}}(\delta) \leftarrow H^{sd}_{a,\iota_{Fi}}(\delta) + 1$
16: $H^{sd,S}_{a,\iota_{Fi},s}(\delta) \leftarrow H^{sd,S}_{a,\iota_{Fi},s}(\delta) + 1$
17: **if** $\delta > \alpha$ **then**
18: $\delta \leftarrow \alpha$
19: **end if**

20: # —— Adjust LRU stack ——
21: **while** $\delta > 1$ **do**
22: $\zeta^S_s(\delta) \leftarrow \zeta^S_s(\delta - 1)$
23: $\delta \leftarrow \delta - 1$
24: **end while**
25: $\zeta^S_s(1) \leftarrow \kappa(m_{a,\iota_{Fi},j})$
26: **end for**

2.2 The FOA Method

The FOA (frequency of access) method is one of two heuristic cache contention prediction methods proposed in [Chandra et al., 2005]. It is based on the idea that a thread that gets co-scheduled with other threads, and competes for shared cache space, receives only a subset of the whole cache space, the so-called *effective cache space* for that thread. The ratio of a thread's *effective cache space* to its *working set size* determines the impact cache sharing has on the thread's performance: The lower the *effective cache space*, compared to the *working set size*, the more the thread suffers from cache sharing. Given an α way set associative cache and a stack distance histogram $H^{sd}_{a,\iota_{\mathcal{F}}i}$, then the lower *effective cache space* results in lowering the hit $\leftrightarrow$ miss barrier from associativity α to a virtual effective associativity α', while way size and line size of the cache remain unchanged. Figure 6 shows how Chandra et al.'s *FOA* method assumes that the cache space is reduced.

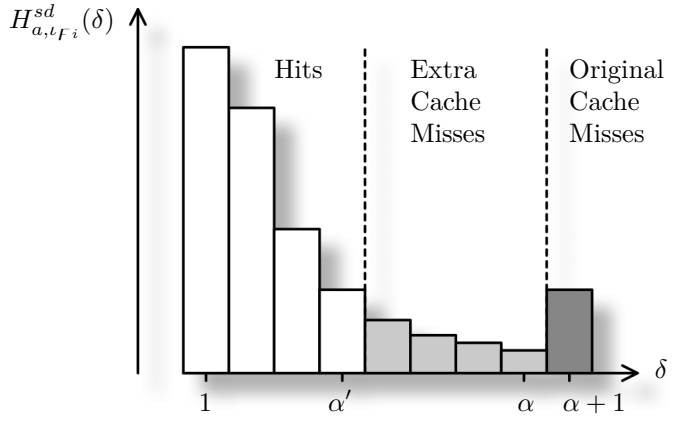


Figure 6: Extra cache misses from cache sharing.

If an application a is executed stand-alone without sharing the cache with any other application, the number of misses μ of a in execution interval $\iota_{\mathcal{F}}i$ for an α way set associative cache is determined by

$$\mu_{a,\iota_{\mathcal{F}}i} = H^{sd}_{a,\iota_{\mathcal{F}}i}(\alpha+1). \tag{3}$$

If, in contrast, for an application a, sharing the cache with a set of applications C_a in interval $\iota_{\mathcal{F}}i$ would reduce α to an effective $\alpha'_{C_a,\iota_{\mathcal{F}}i} \in \mathbb{N}$, additional misses would have to

be considered and the total number of misses then calculates to

$$\mu_{a,\iota_{Fi}} = \sum_{\delta=\alpha'_{C_a,\iota_{Fi}}+1}^{\alpha+1} H^{sd}_{a,\iota_{Fi}}(\delta). \tag{4}$$

When calculating additional misses from α', Chandra et al. note that they apply "linear interpolation whenever necessary" [Chandra et al., 2005]. Therefore, if $\alpha'_{C_a,\iota_{Fi}} \in \mathbb{R}$, I calculate μ by

$$\mu_{a,\iota_{Fi}} = \left(\lceil \alpha'_{C_a,\iota_{Fi}} \rceil - \alpha'_{C_a,\iota_{Fi}}\right) \cdot H^{sd}_{a,\iota_{Fi}}(\lfloor \alpha'_{C_a,\iota_{Fi}} \rfloor + 1) + \sum_{\delta=\lceil \alpha'_{C_a,\iota_{Fi}} \rceil+1}^{\alpha+1} H^{sd}_{a,\iota_{Fi}}(\delta) \tag{5}$$

With the *FOA* method, Chandra et al. propose a technique to *estimate the reduction of* α *to* α'. They assume that a thread's effective cache space is proportional to its access frequency, as threads with high access frequency are likely to bring in more data into the cache and retain it [Chandra et al., 2005].

In the following, I present three variations of the FOA method.

Variation 'one'

Variation 'one' represents the original FOA method as it has been proposed in [Chandra et al., 2005]. Hereby, 'one' means that there is exactly *one* histogram $H^{sd}_{a,\iota_F i}$ for each execution interval $\iota_F i$ of an application a. Let a be an application and let C_a be the set of applications that are co-scheduled with a and compete for a shared cache of associativity α and let $H^{sd}_{a',\iota_F i}$, $a' \in \{a\} \cup C_a$ be the stack distance histograms of capacity $\alpha + 1$ that are calculated from stand-alone execution of applications a' in interval $\iota_F i$ as presented in algorithm 1. The effective associativity $\alpha'_{C_a,\iota_F i}$ for application a, interval $\iota_F i$ calculates to

$$\alpha'_{C_a,\iota_F i} = \frac{\sum_{\delta=1}^{\alpha+1} H^{sd}_{a,\iota_F i}(\delta)}{\sum_{a' \in \{a\} \cup C_a} \sum_{\delta=1}^{\alpha+1} H^{sd}_{a',\iota_F i}(\delta)} \cdot \alpha. \tag{6}$$

Similar to equation 5, Chandra et al. calculate predictor $p_{C_a,\iota_F i}$ that predicts *additional* misses introduced to application a in interval $\iota_F i$ due to reduced associativity $\alpha'_{C_a,\iota_F i}$ by

$$p_{C_a,\iota_F i} = \left(\lceil \alpha'_{C_a,\iota_F i} \rceil - \alpha'_{C_a,\iota_F i}\right) \cdot H^{sd}_{a,\iota_F i}(\lceil \alpha'_{C_a,\iota_F i} \rceil) + \sum_{\delta=\lceil \alpha'_{C_a,\iota_F i} \rceil + 1}^{\alpha} H^{sd}_{a,\iota_F i}(\delta). \tag{7}$$

Let $l = 4$ be the word length of a processor, measured in units of byte. I apply the following predictors:

Predictor	For all $a' \in$	Size (each a')
$H^{sd}_{a',\iota_F i}$	$\{a\}$	$(\alpha + 1) \cdot l$
$\sum_{\delta=1}^{\alpha+1} H^{sd}_{a',\iota_F i}(\delta)$	$\{a\} \cup C_a$	l

Variation 'set'

Calculating α' by equation 6 as it has been proposed in [Chandra et al., 2005] implicitly assumes that memory references are uniformly distributed in the set of cache sets S. However, memory references might show spatial locality, referencing only a small subset of the available cache sets [Settle et al., 2004]. Therefore, if access frequency really had a significant effect on α', as it has been stated by Chandra et al., then α' actually would have to be calculated on a per-cache-set basis. Therefore, I present variation 'set' that calculates a separate $\alpha'^{S}_{C_a,\iota_{Fi},s}$ for each cache set according to

$$\alpha'^{S}_{C_a,\iota_{Fi},s} = \frac{\sum_{\delta=1}^{\alpha+1} H^{sd,S}_{a,\iota_{Fi},s}(\delta)}{\sum_{a'\in\{a\}\cup C_a}\sum_{\delta=1}^{\alpha+1} H^{sd,S}_{a',\iota_{Fi},s}(\delta)}, \tag{8}$$

where $H^{sd,S}_{a',\iota_{Fi},s}$, $a' \in \{a\} \cup C_a$ is the *per cache set* stack distance histogram of application a', interval ι_{Fi}, cache set s that can be calculated as shown in algorithm 1.
Given $\alpha'^{S}_{C_a,\iota_{Fi},s}$ and $H^{sd,S}_{a,\iota_{Fi},s}$ for every $s \in S$, then, in variation 'set', $p_{C_a,\iota_{Fi}}$ that predicts additional cache misses introduced to application a, interval ι_{Fi} due to cache contention with applications C_a calculates to

$$p_{C_a,\iota_{Fi}} = \sum_{s=1}^{|S|}\Bigg(\left(\lceil \alpha'^{S}_{C_a,\iota_{Fi},s}\rceil - \alpha'^{S}_{C_a,\iota_{Fi},s}\right)\cdot H^{sd,S}_{a,\iota_{Fi},s}(\lceil \alpha'^{S}_{C_a,\iota_{Fi},s}\rceil) + \sum_{\delta=\lceil \alpha'^{S}_{C_a,\iota_{Fi},s}\rceil+1}^{\alpha} H^{sd,S}_{a,\iota_{Fi},s}(\delta)\Bigg) \tag{9}$$

For my evaluations, I calculate $p_{C_a,\iota_{Fi}}$ from the following predictors:

Predictor	For all $s \in S$ and $a' \in$	Size (each a')
$H^{sd,S}_{a',\iota_{Fi},s}$	$\{a\}$	$\lvert S\rvert \cdot (\alpha+1)\cdot l$
$\sum_{\delta=1}^{\alpha+1} H^{sd,S}_{a',\iota_{Fi},s}(\delta)$	$\{a\}\cup C_a$	$\lvert S\rvert \cdot l$

Variation 'set, masking'

In variation 'set, masking', I account for those cache sets only to contribute to $p_{C_a,\iota_{Fi}}$ that are able to introduce contention misses from memory references that are *hits* in stand-alone execution. This might be a significant enhancement, *if* cache contention is primarily determined by stack distance histogram entries $1 \dots \alpha$, as it is assumed by the FOA method and nearly all other state-of-the art prediction methods as well (e.g. [Chandra et al., 2005]). Therefore, I calculate a mask $\xi^S_{C_a,\iota_{Fi},s}$ for each cache set by

$$\xi^S_{C_a,\iota_{Fi},s}\{x\} = \begin{cases} 1, & \text{if } \sum_{a' \in \{a\} \cup C_a} \delta^{\max,S}_{a',\iota_{Fi},s} > \alpha \\ 0, & \text{if } \sum_{a' \in \{a\} \cup C_a} \delta^{\max,S}_{a',\iota_{Fi},s} \leq \alpha \end{cases} \tag{10}$$

where $\delta^{\max,S}_{a',\iota_{Fi},s}$, $a' \in \{a\} \cup C_a$, represents the largest δ with $\delta \leq \alpha$, that yields $H^{sd,S}_{a',\iota_{Fi},s}(\delta) \neq 0$ (cf. section 2.10). Given a cache of way size $|w|$, line size $|\lambda|$ and associativity α, I calculate $\delta^{\max,S}_{a',\iota_{Fi}}$ as shown in algorithm 2.

Algorithm 2 Calculating predictor $\delta^{\max,S}_{a,\iota_{Fi},s}$ from $H^{sd,S}_{a,\iota_{Fi},s}$.

```
1: for s ← 1 to |S| do
2:    δ^{max,S}_{a,ι_F i,s} ← α
3:    while δ^{max,S}_{a,ι_F i,s} > 0 ∧ H^{sd,S}_{a,ι_F i,s}(δ^{max,S}_{a,ι_F i,s}) = 0 do
4:       δ^{max,S}_{a,ι_F i,s} ← δ^{max,S}_{a,ι_F i,s} − 1
5:    end while
6: end for
```

To exclusively account for those $\alpha'^S_{C_a,\iota_{Fi},s}$ that belong to cache sets that have the potential to introduce cache contention misses from stand-alone cache hits only, I integrate $\xi^S_{C_a,\iota_{Fi},s}$ in equation 9 to mask out all other cache sets and calculate $p_{C_a,\iota_{Fi}}$ according to

$$p_{C_a,\iota_{Fi}} = \sum_{s=1}^{|S|} \Bigg(\xi_{C_a,\iota_{Fi},s} \cdot \Big(\big(\lceil \alpha'^S_{C_a,\iota_{Fi},s} \rceil - \alpha'^S_{C_a,\iota_{Fi},s} \big) \cdot H^{sd,S}_{a,\iota_{Fi},s}(\lceil \alpha'^S_{C_a,\iota_{Fi},s} \rceil) + \sum_{\delta = \lceil \alpha'^S_{C_a,\iota_{Fi},s} \rceil + 1}^{\alpha} H^{sd,S}_{a,\iota_{Fi},s}(\delta) \Big) \Bigg), \tag{11}$$

where $\alpha'^S_{C_a,\iota_{Fi},s}$ is calculated as shown in equation 8.

I calculate $p_{C_a,\iota_{F\,i}}$ from the following predictors:

Predictor	For all $s \in S$ and $a' \in$	Size (each a')
$H^{sd,S}_{a',\iota_{F\,i},s}$	$\{a\}$	$\lvert S\rvert \cdot (\alpha+1) \cdot l$
$\sum_{\delta=1}^{\alpha+1} H^{sd,S}_{a',\iota_{F\,i},s}(\delta)$	$\{a\} \cup C_a$	$\lvert S\rvert \cdot l$
$\delta^{\max,S}_{a',\iota_{F\,i},s}$	$\{a\} \cup C_a$	$\lvert S\rvert \cdot l$

However, assuming that a thread's effective cache space is proportional to its access frequency is a very coarse model. In fact, it is rather a *different distribution* of stack distance frequencies that will affect a thread's ability to keep its data in cache, as for example a more concentrated stack distance profile (i.e. $\forall\delta, 1 \leq \delta < \alpha : H^{sd}_{a,\iota_{F\,i}}(\delta+1) \neq 0 \Rightarrow H^{sd}_{a,\iota_{F\,i}}(\delta) \gg H^{sd}_{a,\iota_{F\,i}}(\delta+1)$) corresponds to a high temporal locality, making cache lines less likely to be replaced, which in turn increases the effective cache space. Chandra et al. try to address this observation by their *SDC* (stack distance competition) and *Prob* (probabilistic) methods, while I address this observation by my *Width* and *DMax* methods.

2.3 The SDC Method

With the SDC (stack distance competition) model, Chandra et al. propose an alternative to the FOA method to calculate effective α'. The model merges stack distance entries $H^{sd}_{a',\iota_{Fi}}$ of competing threads $a' \in \{a\} \cup C_a$ to build up a new stack distance histogram $H^{sd,merged}_{\{a\}\cup C_a,\iota_{Fi}}$ of capacity α by applying a greedy strategy. Then, the *effective cache space* for each application $a' \in \{a\} \cup C_a$ is computed proportionally to the number of entries in $H^{sd,merged}_{\{a\}\cup C_a,\iota_{Fi}}$ that originate from a', i.e. $0 \dots \alpha$.

Variation 'one'

Just like variation 'one' in the FOA method, variation 'one' in the SDC model corresponds to the original method proposed in [Chandra et al., 2005] and calculates $\alpha'_{C_a,\iota_{Fi}}$ and $p_{C_a,\iota_{Fi}}$ from only a single stack distance histogram $H^{sd}_{a',\iota_{Fi}}$ for each application $a' \in \{a\} \cup C_a$ and execution interval ι_{Fi}.

Let $a \mapsto b$ be the operation that replaces, in an algorithm, element a by element b. In SDC variation 'one', $\alpha'_{C_a,\iota_{Fi}}$ is calculated as presented by algorithm 3, but with $|S| \mapsto 1$, $H^{sd,S}_{a',\iota_{Fi},s} \mapsto H^{sd}_{a',\iota_{Fi}}$ and $\alpha'^{S}_{C_a,\iota_{Fi},s} \mapsto \alpha'_{C_a,\iota_{Fi}}$; note that pure algorithm 3 calculates α' on a per-cache-set basis. Contrary to the FOA method, the SDC algorithm to calculate $\alpha'_{C_a,\iota_{Fi}}$ results in values of $\alpha'_{C_a,\iota_{Fi}} \in \mathbb{N}_0^+$. Therefore, no interpolation has to be performed (contrary to equation 7) and prediction $p_{C_a,\iota_{Fi}}$ calculates to

$$p_{C_a,\iota_{Fi}} = \sum_{\delta=\alpha'_{C_a,\iota_{Fi}}+1}^{\alpha} H^{sd}_{a,\iota_{Fi}}(\delta). \tag{12}$$

Compared to the FOA method, $\sum_{\delta=1}^{\alpha+1} H^{sd}_{a',\iota_{Fi}}$ for each $a' \in \{a\} \cup C_a$ can be omitted; the predictors necessary for SDC variation 'one' is as follows.

Predictor	For each $a' \in$	Size (each a')
$H^{sd}_{a',\iota_{Fi}}$	$\{a\} \cup C_a$	$(\alpha+1) \cdot l$

Variation 'set'

Variation *set* tries to exploit spatial locality as presented in [Settle et al., 2004]. Therefore, α' calculates on a per-set basis according to algorithm 3. Given $\alpha'^{S}_{C_a,\iota_F i,s}$ for each cache set $s \in S$, prediction $p_{C_a,\iota_F i}$ calculates to

$$p_{C_a,\iota_F i} = \sum_{s=1}^{|S|} \sum_{\delta=\alpha'^{S}_{C_a,\iota_F i,s}+1}^{\alpha} H^{sd,S}_{a,\iota_F i,s}(\delta), \tag{13}$$

applying a stack distance histogram for each cache set as predictor.

Predictor	For all $s \in S$ and $a' \in$	Size (each a')
$H^{sd,S}_{a',\iota_F i,s}$	$\{a\} \cup C_a$	$\lvert S\rvert \cdot (\alpha+1) \cdot l$

In contrast to the FOA method (cf. FOA, variation 'set, masking'), the SDC method has masking integrated automatically.

Proof: If a merged stack distance histogram $H^{sd,merged,S}_{\{a\}\cup C_a,\iota_F i,s}$ contains *all* stack distance entries $H^{sd,S}_{a',\iota_F i,s}(\delta) \neq 0,\ 1 \leq \delta \leq \alpha$ of all co-scheduled applications $a' \in \{a\} \cup C_a$, then algorithm 3 calculates $\alpha'^{S}_{C_a,\iota_F i,s}$ as the number of all entries in $H^{sd,S}_{a,\iota_F i,s}(\delta),\ 1 \leq \delta \leq \alpha$ that are $\neq 0$, i.e.

$$\sum_{a' \in \{a\}\cup C_a} \delta^{\max,S}_{a',\iota_F i,s} \leq \alpha \quad \Rightarrow \quad \alpha'^{S}_{C_a,\iota_F i,s} = \delta^{\max,S}_{a,\iota_F i,s}. \tag{14}$$

Given effective associativity $\alpha'^{S}_{C_a,\iota_F i,s}$ as calculated by equation 14. Then, in equation 13, $\sum_{\delta=\alpha'^{S}_{C_a,\iota_F i,s}+1}^{\alpha} H^{sd,S}_{a,\iota_F i,s}(\delta) = 0$, because all δ with $\alpha'^{S}_{C_a,\iota_F i,s} + 1 \leq \delta \leq \alpha$ can address only those entries in $H^{sd,S}_{a,\iota_F i,s}$ that are 0, as $\alpha'^{S}_{C_a,\iota_F i,s} + 1 \leq \delta$ implies $\delta > \delta^{\max,S}_{a,\iota_F i,s}$. □

As masking is implicitly integrated in SDC variation 'set', there is no need to investigate a separate variation 'set, masking'.

Algorithm 3 SDC model to calculate reduced associativity $\alpha'^{S}_{C_a,\iota_{Fi},s}$.

```
# —— Iterate over all cache sets ——
for s ← 1 to |S| do
  # —— Initialize pointers to next element of each histogram ——
  for all a' ∈ {a} ∪ C_a do
    δ_a' ← 1
  end for
  # —— Calculate contribution of a when merging histograms ——
  for δ ← 1 to α do
    # —— Get application with highest histogram value ——
    H ← 0
    a'' ← a
    for all a' ∈ {a} ∪ C_a do
      if H^{sd,S}_{a',ι_Fi,s}(δ_a') > H then
        H ← H^{sd,S}_{a',ι_Fi,s}(δ_a')
        a'' ← a'
      end if
    end for
    # —— Increase contribution ——
    if H > 0 then
      δ_a'' ← δ_a'' + 1
    end if
  end for
  # —— δ_a − 1 corresponds to α'^S_{C_a,ι_Fi,s} ——
  α'^S_{C_a,ι_Fi,s} ← δ_a − 1
end for
```

Variation ‘lru set group’

In this variation, I combine Chandra et al.’s SDC method with a technique presented by Suh et al. for cache partitioning [Suh et al., 2002].

In [Suh et al., 2002], Suh et al. introduce a method to partition a cache among multiple applications in order to maximize overall cache performance. The method is based on the *miss rate versus cache size curve* an application shows when executed in absence of any other application, i.e. on stand-alone execution. Figure 7 shows a miss rate curve similar to the one presented in [Suh et al., 2002] for the gcc benchmark.

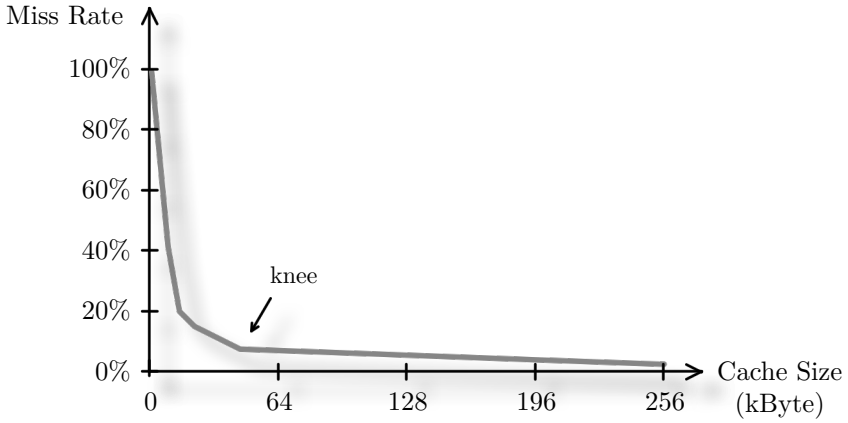


Figure 7: Miss rate curve for an application as it has similarly been presented in [Suh et al., 2002].

For small cache sizes, the curve quickly drops off and then settles down, forming a knee that is typical for many applications [Suh et al., 2002]. If an application has less cache space available than determined by the abscissa of its knee, cache performance heavily degrades. For larger cache sizes however, only a small gain in cache performance can be achieved. By reason of this observation, Suh et al. suggest to assign at least as many cache resources to an application as it is determined by the abscissa of the knee in the application’s miss rate vs. cache size diagram — for stand-alone execution as well as for execution in co-schedule with other applications where each application is assigned a predefined partition of the cache. To determine the partition sizes for the applications, Suh et al. apply a greedy strategy on the derivative of the miss rate curve that has been multiplied by the number of memory references: they assign to each application at least

as many cache blocks as determined by the abscissa of the knee in the miss rate curve — just the way Chandra et al. merge stack distance histogram entries of co-scheduled applications to a new stack distance histogram when calculating α' in algorithm 3. Since the multiplication of the number of memory references with the derivative of the miss rate curve results in a function that plots misses vs. cache size, i.e. a stack distance histogram for a fully associative cache, Suh et al. actually presented Chandra et al.'s SDC method (2005) as early as 2002.

Although Suh et al. consider *set associative* caches, they calculate their partitions from the miss rate curve of a *fully associative* cache; to obtain that curve, however, they apply data gathered from performance counters of a *set associative* cache. As a cache with associativity α results in only α different entries in a stack distance histogram (Suh et al. only regared cache *hits* $\Rightarrow$ entry $\alpha + 1$ is not used), Suh et al. not only use LRU information from cache ways, but also LRU information from *cache set accesses* to obtain stack distance histograms with more than just α entries: They partition $|S|$ cache sets into $|G|$ equally sized groups $G = \{g_1, \ldots, g_{|G|}\}$ (Suh et al.: $|G| \in \{8, 16\}$) and monitor, if the currently accessed cache set is member of the group that has *least recently*, *2nd least recently* ... or $|G|$ least recently been accessed, applying an LRU algorithm. To combine way LRU and set LRU information, Suh et al. create the set of $|G|$ histograms $H^{sd,G}_{a,\iota_F i} = \{H^{sd,G}_{a,\iota_F i,1}, \ldots, H^{sd,G}_{a,\iota_F i,|G|}\}$ of capacity α each. $H^{sd,G}_{a,\iota_F i,\delta_G}(\delta_S)$ refers to the number of accesses to set groups of LRU stack position δ_G and to cache ways of LRU stack position δ_S. To transform histograms $H^{sd,G}_{a,\iota_F i}$ into *one* stack distance histogram $H^{sd,ext}_{a,\iota_F i}$ of extended size, Suh et al. take all $\alpha \cdot |G|$ elements stored in $H^{sd,G}_{a,\iota_F i}$, sort them by value and create a new histogram $H^{sd,ext}_{a,\iota_F i}$ of capacity $\alpha \cdot |G|$ that holds all those values.

Given a set associative cache of associativity α and $|S|$ cache sets $S = \{s_1, \ldots, s_{|S|}\}$ and an application a with memory references $M_{a,\iota_F i}$ in execution interval $\iota_F i$.

Let $\zeta^S = \{\zeta^S_1, \ldots, \zeta^S_{|S|}\}$ be the set of stacks of capacity α that hold LRU information for the lines of each cache set, let ζ^G be a stack of capacity $|G|$ that holds LRU information for $|G|$ set groups, and let $\gamma(s) = \big((s-1) \cdot |G|/|S|\big) + 1$ be the operation that extracts the set group from set number s and let $1 \leq \gamma(s) \leq |G|$. Then, in variation 'lru set group', stack distance histogram $H^{sd,ext}_{a,\iota_F i}$ of capacity $\alpha \cdot |G|$ for application a, interval $\iota_F i$ is calculated as shown in algorithm 4. For my evaluation, I apply $|G| = 16$.

Given stack distance histograms $H^{sd,ext}_{a',\iota_F i}$ for all applications $a' \in \{a\} \cup C_a$, I calculate $\alpha'_{C_a,\iota_F i}$ according to algorithm 3 with

- $|S| \mapsto 1$,
- $H^{sd,S}_{a,\iota_F i,s} \mapsto H^{sd,ext}_{a,\iota_F i}$,
- $\alpha \mapsto \alpha \cdot |G|$ and
- $\alpha'^{S}_{C_a,\iota_F i,s} \mapsto \alpha'_{C_a,\iota_F i}$.

Given $H^{sd,ext}_{a,\iota_F i}$ and $\alpha'_{C_a,\iota_F i}$, I calculate prediction $p_{C_a,\iota_F i}$ for SDC variation 'lru set group' according to

$$p_{C_a,\iota_F i} = \sum_{\delta=\alpha'_{C_a,\iota_F i}+1}^{\alpha\cdot|G|} H^{sd,ext}_{a,\iota_F i}(\delta). \tag{15}$$

For this calculation, I apply predictors as follows.

Predictor	For all $a' \in$	Size (each a')
$H^{sd,ext}_{a',\iota_F i}$	$\{a\} \cup C_a$	$\|G\| \cdot \alpha \cdot l$

Algorithm 4 Generate stack distance histogram $H^{sd,ext}_{a,\iota_{Fi}}$ of capacity $\alpha \cdot |G|$.

1: # —— Initialize stacks and histograms ——
2: **for** $s \leftarrow 1$ **to** $|S|$ **do**
3: **for** $\delta_S \leftarrow 1$ **to** α **do**
4: $\zeta^S_s(\delta_S) \leftarrow -1$
5: **end for**
6: **end for**
7: **for** $\delta_G \leftarrow 1$ to $|G|$ **do**
8: $\zeta^G(\delta_G) \leftarrow -1$
9: **for** $\delta_S \leftarrow 1$ to α **do**
10: $H^{sd,G}_{a,\iota_{Fi},\delta_G}(\delta_S) \leftarrow 0$
11: **end for**
12: **end for**

13: # —— Calculate all $|G|$ histograms $H^{sd,G}_{a,\iota_{Fi},g},\ g \in \{1,\ldots,|G|\}$ ——
14: **for** $j \leftarrow 1$ to $|M_{a,\iota_{Fi}}|$ **do**

15: # —— Update histogram ——
16: $s \leftarrow \varsigma(m_{a,\iota_{Fi},j})$
17: $\delta_G \leftarrow \zeta^G\{\gamma(s)\}$
18: $\delta_S \leftarrow \zeta^S_s\{\kappa(m_{a,\iota_{Fi},j})\}$
19: **if** $\delta_S \leq \alpha$ **then**
20: $H^{sd,G}_{a,\iota_{Fi},\delta_G}(\delta_S) \leftarrow H^{sd,G}_{a,\iota_{Fi},\delta_G}(\delta_S) + 1$
21: **else**
22: $\delta_S \leftarrow \alpha$
23: **end if**

24: # —— Adjust line LRU stack of cache set s ——
25: **while** $\delta_S > 1$ **do**
26: $\zeta^S_s(\delta_S) \leftarrow \zeta^S_s(\delta_S - 1)$
27: $\delta_S \leftarrow \delta_S - 1$
28: **end while**
29: $\zeta^S_s(1) \leftarrow \kappa(m_{a,\iota_{Fi},j})$

30: # —— Adjust set group LRU stack ——
31: **if** $\delta_G = -1$ **then**
32: $\delta_G \leftarrow |G|$
33: **end if**
34: **while** $\delta_G > 1$ **do**
35: $\zeta^G(\delta_G) \leftarrow \zeta^G(\delta_G - 1)$
36: $\delta_G \leftarrow \delta_G - 1$
37: **end while**
38: $\zeta^G(1) \leftarrow \gamma(s)$
39: **end for**

40: # —— Merge all $|G|$ histograms $H^{sd,G}_{a,\iota_F i,g},\ g \in \{1,\ldots,|G|\}$
to a single sorted histogram $H^{sd,ext}_{a,\iota_F i}$ of extended size ——
41: **for** $\delta \leftarrow 1$ **to** $\alpha \cdot |G|$ **do**
42: $H \leftarrow 0$
43: $\delta_S \leftarrow 0$
44: $\delta_G \leftarrow 0$
45: **for** $\delta'_G \leftarrow 1$ **to** $|G|$ **do**
46: **for** $\delta'_S \leftarrow 1$ **to** α **do**
47: **if** $H^{sd,G}_{a,\iota_F i,\delta'_G}(\delta'_S) > H$ **then**
48: $H \leftarrow H^{sd,G}_{a,\iota_F i,\delta'_G}(\delta'_S)$
49: $\delta_S \leftarrow \delta'_S$
50: $\delta_G \leftarrow \delta'_G$
51: **end if**
52: **end for**
53: **end for**
54: $H^{sd,G}_{a,\iota_F i,\delta_G}(\delta_S) \leftarrow 0$
55: $H^{sd,ext}_{a,\iota_F i}(\delta) \leftarrow H$
56: **end for**

2.4 The Prob Method

With their heuristic methods FOA and SDC, Chandra et al. calculate additional cache misses introduced from cache contention by means of a reduced cache size, reflected in an effective associativity α' for set associative caches (cf. figure 6). With their analytical, inductive probability based *Prob* method, Chandra et al. calculate additional cache misses introduced from cache contention by the probability that a hit in stand-alone execution turns into a miss when sharing the cache [Chandra et al., 2005]. Given this probability, they predict additional cache misses introduced from a set of applications C_a to an application a in execution interval ι_{Fi} by

$$p_{C_a,\iota_{Fi}} = \sum_{c_a \in C_a} \sum_{\delta=1}^{\alpha} H^{sd}_{a,\iota_{Fi}}(\delta) \cdot P^{miss}_{c_a,\iota_{Fi}}(\delta). \tag{16}$$

$H^{sd}_{a,\iota_{Fi}}(\delta)$ refers to element δ in stack distance histogram $H^{sd}_{a,\iota_{Fi}}$ (cf. algorithm 1) and denotes the frequency of memory references with LRU stack distance δ in application a, interval ι_{Fi}. $P^{miss}_{c_a,\iota_{Fi}}(\delta)$, $1 \leq \delta \leq \alpha$, is the probability that a memory reference of application a, interval ι_{Fi} that accesses the LRU stack with a distance δ will turn into a miss, if a is co-scheduled with c_a. Figure 8 exemplarily shows how these extra misses are located in a stack distance histogram $H^{sd}_{a,\iota_{Fi}}$; note the difference to figure 6.

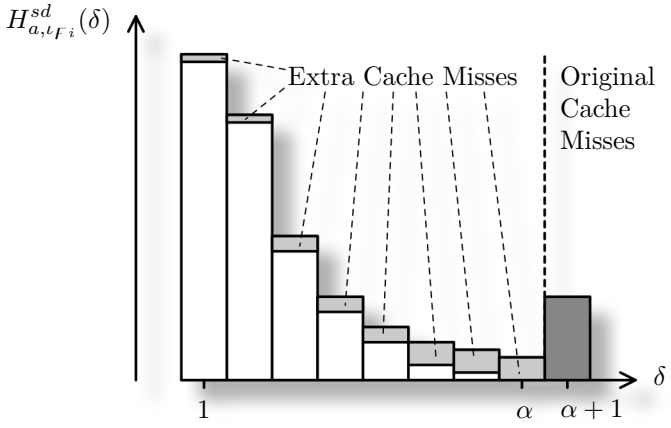


Figure 8: Effective cache space Prob model.

To turn a memory reference m_a of application a to cache set s with LRU stack distance δ, $1 \leq \delta \leq \alpha$ and key $\kappa(m_a)$ into a miss, a co-scheduled application c_a has to provide memory sequences to $\delta_{c_a} \geq \alpha - \delta + 1$ different cache lines in cache set s.

Chandra et al. calculate $P^{miss}_{c_a,\iota_{\digamma}i}(\delta)$ by

$$P^{miss}_{c_a,\iota_{\digamma}i}(\delta) = \sum_{\delta_{c_a}=\alpha-\delta+1}^{E(\nu_{c_a,\iota_{\digamma}i}(\delta))} P^{seq}_{c_a,\iota_{\digamma}i}(\delta_{c_a}, E(\nu_{c_a,\iota_{\digamma}i}(\delta))), \tag{17}$$

where $E(\nu_{c_a,\iota_{\digamma}i}(\delta))$ is the expected number of memory addresses that application c_a references in that time that application a, on average, takes to refer to δ different cache lines. $P^{seq}_{c_a,\iota_{\digamma}i}(\delta,\nu)$ is the probability that within a *seq*uence of ν memory references of application c_a in interval $\iota_{\digamma}i$ that map to the same cache set, δ different cache lines are used. Chandra et al. refer to such sequences as $seq(\delta,\nu)$. To calculate $P^{seq}_{c_a,\iota_{\digamma}i}(\delta,\nu)$, Chandra et al. recursively apply

$$P^{seq}_{c_a,\iota_{\digamma}i}(\delta,\nu) = \begin{cases} \left(P^{dist}_{c_a,\iota_{\digamma}i}(1)\right)^{\nu-1}, & \text{if } \nu \geq \delta = 1 \\ P^{seq}_{c_a,\iota_{\digamma}i}(\delta-1,\delta-1)\cdot\sum_{\delta'=\delta}^{\alpha+1} P^{dist}_{c_a,\iota_{\digamma}i}(\delta'), & \text{if } \nu = \delta > 1 \\ P^{seq}_{c_a,\iota_{\digamma}i}(\delta,\nu-1)\cdot\sum_{\delta'=1}^{\delta} P^{dist}_{c_a,\iota_{\digamma}i}(\delta') + & \\ \quad P^{seq}_{c_a,\iota_{\digamma}i}(\delta-1,\nu-1)\cdot\sum_{\delta'=\delta}^{\alpha+1} P^{dist}_{c_a,\iota_{\digamma}i}(\delta'), & \text{if } \nu > \delta > 1 \end{cases} \tag{18}$$

where $P^{dist}_{c_a,\iota_{\digamma}i}(\delta),\ 1 \leq \delta \leq \alpha+1$ is the probability that a memory reference of application c_a in interval $\iota_{\digamma}i$ has a stack distance of δ. Given a stack distance histogram $H^{sd}_{c_a,\iota_{\digamma}i}(\delta)$, $P^{dist}_{c_a,\iota_{\digamma}i}(\delta)$ can be calculated by

$$P^{dist}_{c_a,\iota_{\digamma}i}(\delta) = \frac{H^{sd}_{c_a,\iota_{\digamma}i}(\delta)}{\sum_{\delta'=1}^{\alpha+1} H^{sd}_{c_a,\iota_{\digamma}i}(\delta')} \tag{19}$$

and

$$\sum_{\delta=x}^{y} P^{dist}_{c_a,\iota_{\digamma}i}(\delta) = \frac{\sum_{\delta'=x}^{y} H^{sd}_{c_a,\iota_{\digamma}i}(\delta')}{\sum_{\delta'=1}^{\alpha+1} H^{sd}_{c_a,\iota_{\digamma}i}(\delta')}. \tag{20}$$

Note that in equation 17, $E(\nu_{c_a,\iota_{\digamma}i}(\delta))$ might be larger than $\alpha+1$. This generally poses a problem, as the calculation of $P^{seq}_{c_a,\iota_{\digamma}i}(\delta_{c_a}, E(\nu_{c_a,\iota_{\digamma}i}(\delta)))$ according to equation 18 calculates $P^{dist}_{c_a,\iota_{\digamma}i}(\delta_{c_a})$ by equation 19, which is not defined for values of $\delta_{c_a} > \alpha+1$, as $H^{sd}_{c_a,\iota_{\digamma}i}(\delta_{c_a})$ is

defined for $1 \leq \delta_{c_a} \leq \alpha + 1$ only. This problem is not addressed in [Chandra et al., 2005]. However, I found out that $\delta_{c_a} > \alpha + 1 \Rightarrow P^{seq}_{c_a,\iota_{F\,i}}(\delta_{c_a}, E(\nu_{c_a,\iota_{F\,i}}(\delta))) = 0$ independently of $P^{dist}_{c_a,\iota_{F\,i}}(\delta_{c_a})$.

Proof:

- In equation 18, assume to calculate $P^{seq}_{c_a,\iota_{F\,i}}(\alpha+j, \alpha+j)$ with $j \in \mathbb{N}, j > 1$. Then, due to $\sum_{\delta'=\alpha+j}^{\alpha+1} P^{dist}_{c_a,\iota_{F\,i}}(\delta') = 0$ (line 2), $P^{seq}_{c_a,\iota_{F\,i}}(\alpha + j, \alpha + j) = 0$.
- In equation 18, assume to calculate $P^{seq}_{c_a,\iota_{F\,i}}(\alpha + j, \alpha + k)$, $j, k \in \mathbb{N}$, $1 < j < k$.
  - In line 4, $\sum_{\delta'=\alpha+j}^{\alpha+1} P^{dist}_{c_a,\iota_{F\,i}}(\delta') = 0$, because $\alpha + j > \alpha + 1$. Applying recursion in line 4, line 4 will turn either into $0 \cdot (\text{line } 3 + \text{line } 4)$, or into $0 \cdot \text{line } 1$. Therefore, line 4 will always be 0 for $\delta_{c_a} > \alpha + 1$.
  - Applying recursion in line 3, line 3 will $k - j - 1$ times turn to $\left(\sum_{\delta'=1}^{\delta_{c_a}} P^{dist}_{c_a,\iota_{F\,i}}(\delta')\right) \cdot (\text{line } 3 + \text{line } 4)$, before it turns to $\left(\sum_{\delta'=1}^{\delta} P^{dist}_{c_a,\iota_{F\,i}}(\delta')\right) \cdot \text{line } 2$. As line 4 turns to 0 for $\delta_{c_a} > \alpha + 1$ however, line 3 actually turns $k - j - 1$ times to $\left(\sum_{\delta'=1}^{\delta} P^{dist}_{c_a,\iota_{F\,i}}(\delta')\right) \cdot \text{line } 3$ before it turns to $\left(\sum_{\delta'=1}^{\delta_{c_a}} P^{dist}_{c_a,\iota_{F\,i}}(\delta')\right) \cdot \text{line } 2$. Therefore, calculating $P^{seq}_{c_a,\iota_{F\,i}}(\alpha + j, \alpha + k)$ with $j, k \in \mathbb{N}$ and $1 < j < k$, line 3 turns to $\left[\prod_{n=1}^{k-j} \sum_{\delta'=1}^{\delta_{c_a}} P^{dist}_{c_a,\iota_{F\,i}}(\delta')\right] \cdot \text{line } 2$. As line 2 is 0 for $\delta_{c_a} > \alpha + 1$, line 3 also is 0.
  - As both line 3 and 4 turn to 0 for $\delta_{c_a} > \alpha + 1$, $P^{seq}_{c_a,\iota_{F\,i}}(\alpha + j, \alpha + k) = 0$ if $j, k \in \mathbb{N}$, $1 < j < k$.
- As both $P^{seq}_{c_a,\iota_{F\,i}}(\alpha + j, \alpha + j) = 0$ with $j \in \mathbb{N}, j > 1$ and $P^{seq}_{c_a,\iota_{F\,i}}(\alpha + j, \alpha + k) = 0$ with $j, k \in \mathbb{N}$, $1 < j < k$, $P^{seq}_{c_a,\iota_{F\,i}}(\delta_{c_a}, E(\nu_{c_a,\iota_{F\,i}}(\delta))) = 0$ for $\delta_{c_a} > \alpha + 1$. □

As mentioned before, $E(\nu_{c_a,\iota_{\mathcal{F}}i}(\delta)$ is the expected number of memory accesses that application c_a performs in that time that application a, on average, takes to refer to δ different cache lines. Still the question remains how to calculate $E(\nu_{c_a,\iota_{\mathcal{F}}i}(\delta))$.
Chandra et al. estimate $E(\nu_{c_a,\iota_{\mathcal{F}}i}(\delta))$ from $\overline{\nu}_{a,\iota_{\mathcal{F}}i}(\delta)$, the average number of memory references of application a in interval $\iota_{\mathcal{F}i}$ that occur when a accesses δ different cache lines per set: They scale $\overline{\nu}_{a,\iota_{\mathcal{F}}i}(\delta)$ by the ratio of all memory references in a and c_a according to

$$E(\nu_{c_a,\iota_{\mathcal{F}}i}(\delta)) = \overline{\nu}_{a,\iota_{\mathcal{F}}i}(\delta) \cdot \frac{\sum_{\delta'=1}^{\alpha+1} H^{sd}_{c_a,\iota_{\mathcal{F}}i}(\delta')}{\sum_{\delta'=1}^{\alpha+1} H^{sd}_{a,\iota_{\mathcal{F}}i}(\delta')}. \tag{21}$$

Thereto, they calculate $\overline{\nu}_{a,\iota_{\mathcal{F}}i}(\delta)$ by

$$\overline{\nu}_{a,\iota_{\mathcal{F}}i}(\delta) = \frac{\sum_{\nu=\delta+1}^{\nu_{max}} H^{cseq}_{a,\iota_{\mathcal{F}}i}(\delta,\nu) \cdot \nu}{\sum_{\nu=\delta+1}^{\nu_{max}} H^{cseq}_{a,\iota_{\mathcal{F}}i}(\delta,\nu)}. \tag{22}$$

Note that I apply $\nu_{max} = 128$, as it is proposed in [Chandra et al., 2005].
$H^{cseq}_{a,\iota_{\mathcal{F}}i}$ is a three dimensional histogram of capacity $\alpha \times (\nu_{max} - 1)$ for application a, interval $\iota_{\mathcal{F}i}$. $H^{cseq}_{a,\iota_{\mathcal{F}}i}(\delta,\nu)$ with $1 \leq \delta \leq \alpha$, $\delta \in \mathbb{N}^+$ and $2 \leq \nu \leq \nu_{max}$, $\nu \in \mathbb{N}^+$ refers to the histogram entry that tracks the frequency of so-called *circular sequences* $cseq_s(\delta,\nu)$ that occur in *any* cache set $s \in S$ of application a in interval $\iota_{\mathcal{F}i}$. A circular sequence $cseq_s(\delta,\nu)$ is a sequence of ν memory references to cache set s, where the first and the last reference map to the same cache line λ' and the references in between refer to $\delta - 1$ different cache lines $\{\lambda_1, \ldots, \lambda_{\delta-1}\} = \Lambda$ and $\lambda' \notin \Lambda$. Figure 9, that can similarly be found in [Chandra et al., 2005], demonstrates the difference between *sequences* $seq(\delta,\nu)$ and *circular sequences* $cseq(\delta,\nu)$.

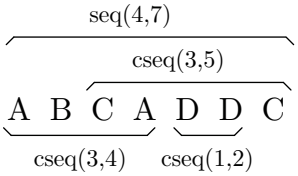


Figure 9: Difference between *sequences* $seq(\delta,\nu)$ and *circular sequences* $cseq(\delta,\nu)$, as it has similarly been presented in [Chandra et al., 2005].

Let $\zeta^{d,S} = \{\zeta_1^{d,S}, \ldots, \zeta_{|S|}^{d,S}\}$ be the set of stacks of capacity α that are used to implement LRU replacement, and let $\zeta^{n,S} = \{\zeta_1^{n,S}, \ldots, \zeta_{|S|}^{n,S}\}$ be a set of $|S|$ stacks of capacity α and let element δ of stack $\zeta_s^{n,S}$, referenced by $\zeta_s^{n,S}(\delta)$, be the element that is used to track the number of memory references to cache set s that occur between the last memory reference to cache set s and the δ most recently used cache line in cache set s.
For an application a, interval $\iota_{\digamma i}$, $H_{a,\iota_{\digamma i}}^{cseq}$ calculates as presented in algorithm 5.

Algorithm 5 Generation of a circular sequence histogram $H_{a,\iota_{\digamma i}}^{cseq}$ for interval $\iota_{\digamma i}$ of application a, applying an α way set associative cache with $|S|$ cache sets.

1: # —— Initialize stacks and histograms ——
2: **for** $\delta \leftarrow 1$ **to** α **do**
3: **for** $s \leftarrow 1$ **to** $|S|$ **do**
4: $\nu_{s,\delta} \leftarrow 0$
5: $\zeta_s^{d,S}(\delta) \leftarrow -1$
6: $\zeta_s^{n,S}(\delta) \leftarrow 0$
7: **end for**
8: **for** $\nu \leftarrow 2$ **to** ν_{max} **do**
9: $H_{a,\iota_{\digamma i}}^{cseq}(\delta, \nu) \leftarrow 0$
10: **end for**
11: **end for**

12: # —— Calculate $H_{a,\iota_{\digamma i}}^{cseq}$ ——
13: **for** $j \leftarrow 1$ **to** $|M_{a,\iota_{\digamma i}}|$ **do**
14: $s \leftarrow \varsigma(m_{a,\iota_{\digamma i},j})$
15: **for** $\delta \leftarrow 1$ **to** α **do**
16: $\zeta_s^{n,S}(\delta) \leftarrow \zeta_s^{n,S}(\delta) + 1$
17: **end for**
18: $\delta \leftarrow \zeta_s^{d,S}\{\kappa(m_{a,\iota_{\digamma i},j})\}$

19: **if** $\delta \leq \alpha$ **then**
20: $\nu \leftarrow \zeta_s^{n,S}(\delta)$
21: **if** $\nu > \nu_{max}$ **then**
22: $\nu \leftarrow \nu_{max}$
23: **end if**
24: $H^{cseq}_{a,\iota_F i}(\delta,\nu) \leftarrow H^{cseq}_{a,\iota_F i}(\delta,\nu) + 1$
25: **end if**

26: # —— Adjust stacks ——
27: **if** $\delta > \alpha$ **then**
28: $\delta \leftarrow \alpha$
29: **end if**
30: **while** $\delta > 1$ **do**
31: $\zeta_s^{d,S}(\delta) \leftarrow \zeta_s^{d,S}(\delta - 1)$
32: $\zeta_s^{n,S}(\delta) \leftarrow \zeta_s^{n,S}(\delta - 1)$
33: **end while**
34: $\zeta_s^{d,S}(1) \leftarrow \kappa(m_{a,\iota_F i,j})$
35: $\zeta_s^{n,S}(1) \leftarrow 1$
36: **end for**

I apply the following predictors when calculating $p_{C_a,\iota_F i}$:

Predictor	For each $a' \in$	Size (each a')
$H^{sd}_{a',\iota_F i}$	$\{a\} \cup C_a$	$(\alpha+1) \cdot l$
$\sum_{\delta=1}^{\alpha+1} H^{sd}_{a',\iota_F i}$	$\{a\} \cup C_a$	l
$\overline{\nu}_{a',\iota_F i}$	$\{a\}$	$\alpha \cdot l$

For performance reasons, I calculate equation 18 only once, then store $P^{seq}_{c_a,\iota_F i}(\delta,\nu)$ for each reasonable combination of δ and ν in a table and then apply them to equation 17.

While Chandra et al. calculate additional misses introduced to application a for each application $c_a \in C_a$ separately, Chen and Aamodt propose a more advanced variation of the Prob model [Chen and Aamodt, 2009] that incorporates effects of all co-scheduled applications at the same time. They adapt equation 16 to

$$p_{C_a,\iota_{F\,i}} = \sum_{\delta=1}^{\alpha} H^{sd}_{a,\iota_{F\,i}}(\delta) \cdot P^{miss}_{C_a,\iota_{F\,i}}(\delta) \tag{23}$$

and calculate $P^{miss}_{C_a,\iota_{F\,i}}(\delta)$ by

$$P^{miss}_{C_a,\iota_{F\,i}}(\delta) = 1 - \sum_{\delta_{c_{a,1}}+\delta_{c_{a,2}}+\cdots+\delta_{c_{a,|C_a|}} \leq \alpha-\delta} \prod_{c_{a'} \in C_a} P^{seq}_{c_{a'},\iota_{F\,i}}(\delta_{c_{a'}}, E(\nu_{c_{a'},\iota_{F\,i}}(\delta))). \tag{24}$$

However, as my simulations showed that calculating co-scheduling penalty by equation 17 is already computationally too slow to predict cache contention effects in a reasonable amount of time (similarly noted by Zhuravlev et al.: "only two of them (FOA and SDC) are computationally fast enough to be used" [Zhuravlev et al., 2010]), and equation 24 will calculate much slower than equation 17 as $P^{seq}_{c_a,\iota_{F\,i}}$ is calculated over far more combinations of input parameters, I desist from investigating Chen and Aamodt extension any further.

While Chandra et al. restricted themselves to threads exploiting private data only, Song et al. extended Chandra's work in [Song et al., 2007] by supporting not only threads with *private*, but also with *shared* data, considering effects such as prefetching of shared data as well. However, as this work focuses on cache contention of different processes not sharing any address space, the adaptions proposed by Song et al. are not discussed any further.

2.5 The Width Method

With the *Width* method, I investigate two assumptions on stack distance histograms $H^{sd}_{a,\iota_{\digamma}i}$ and $H^{sd,S}_{a,\iota_{\digamma}i,s}$ respectively for their ability to predict cache contention:

- Application intervals featuring more concentrated stack distance histograms (e.g. $0.25 \cdot \alpha \leq \delta \leq \alpha \Rightarrow H^{sd}_{a,\iota_{\digamma}i}(\delta) = 0$) are less likely to account for cache contention than application intervals that are less concentrated (e.g. $\delta \leq 0.75 \cdot \alpha \Rightarrow H^{sd}_{a,\iota_{\digamma}i}(\delta) \neq 0$). This seems to be a valid statement, as a co-scheduled application has to provide many references to different cache lines within a short amount of time to displace cache lines of an application with a very concentrated stack distance histogram — and many references to many different cache lines are less likeley to occur than many references to only a small amount of different cache lines due to program locality. On the contrary, it takes only a few references to different cache lines (which is more likely to occur) to displace cache lines of a co-scheduled application that feature a high distance δ.
- For a given δ, an application a is more likely to have cache lines replaced by a co-scheduled application if $H^{sd}_{a,\iota_{\digamma}i}(\delta)$ has a high value rather than a small value. This seems to be a valid statement as a high value of $H^{sd}_{a,\iota_{\digamma}i}(\delta)$ means that there are many references with distance δ, which in turn makes it more likely for a co-scheduled application to displace one, as if there were just a few references with distance δ.

Combining both assumptions, I investigate if multiplying δ by $H^{sd}_{a,\iota_{\digamma}i}(\delta)$ and δ by $H^{sd,S}_{a,\iota_{\digamma}i,s}(\delta)$ respectively would achieve any promising results in predicting cache contention.

Variation ‘one’

With variation ‘one’, I calculate predictor $p_{C_a,\iota_{\digamma}i}$ from *one* stack distance histogram per execution interval $\iota_{\digamma}i$ per application a according to

$$p_{C_a,\iota_{\digamma}i} = \sum_{a' \in \{a\} \cup C_a} \sum_{\delta=1}^{\alpha} H^{sd}_{a',\iota_{\digamma}i}(\delta) \cdot \delta. \tag{25}$$

Note that I apply $\sum_{\delta=1}^{\alpha} H^{sd}_{a',\iota_{F\,i}}(\delta)\cdot\delta$ as predictor in order to perform only those calculations at runtime that cannot be calculated in advance; this makes timing performance of the method more comparable.

Predictor	For all $a' \in$	Size (each a')
$\sum_{\delta=1}^{\alpha} H^{sd}_{a',\iota_{F\,i}}(\delta)\cdot\delta$	$\{a\}\cup C_a$	l

Variation 'set, mask'

With variation 'set, mask', I calculate predictors on a per-cache-set basis, multiplying each δ, $1 \le \delta \le \alpha$, with the stack distance histograms $H^{sd,S}_{a,\iota_{F\,i},s}(\delta)$ of the various cache sets. Additionally, I add *masking* that I already proposed to be incorporated in the FOA method to account for those cache sets in S only that introduce contention misses even from stand-alone cache hits. I calculate $p_{C_a,\iota_{F\,i}}$ by

$$p_{C_a,\iota_{F\,i}} = \sum_{s=1}^{|S|}\Big(\xi^{S}_{C_a,\iota_{F\,i},s}\cdot \sum_{a'\in\{a\}\cup C_a}\sum_{\delta=1}^{\alpha} H^{sd,S}_{a',\iota_{F\,i},s}(\delta)\cdot\delta\Big), \tag{26}$$

where $\xi^{S}_{a,\iota_{F\,i},s}$ is calculated as presented in equation 10.
For my calculations, I apply the following predictors:

Predictor	For all $s \in S$ and $a' \in$	Size (each a')
$\sum_{\delta=1}^{\alpha} H^{sd,S}_{a',\iota_{F\,i},s}(\delta)\cdot\delta$	$\{a\}\cup C_a$	$\lvert S\rvert\cdot l$
$\delta^{max,S}_{a',\iota_{F\,i},s}$	$\{a\}\cup C_a$	$\lvert S\rvert\cdot l$

Variation 'set, mask, exp delta'

With variation 'set, mask, exp delta', I investigate my assumption that a higher δ might contribute much more to cache contention than a higher $H^{sd}(\delta)$, where $H^{sd}(\delta)$ is short for any $H^{sd}_{a',\iota_{F\,i}}(\delta)$ with $a' \in \{a\}\cup C_a$ (or $H^{sd,S}_{a',\iota_{F\,i},s}(\delta)$ with $a' \in \{a\}\cup C_a$ and $s \in S$ respectively). In variation 'one', scaling δ by a factor f has the same effect as scaling $H^{sd}(\delta)$ by f. However, as I observed that stack distance histograms are often highly concentrated and

$H^{sd}(1)$ often exceeds $H^{sd}(2)$ by several orders of magnitude, variation 'one' primarily uses $H^{sd}(1)$ as a measure for cache contention, which I think is not an optimal choice: $H^{sd}(1)$ represents the number of references with stack distance $\delta = 1$. Although there are many of these, it is unlikely that any reference in $H^{sd}(1)$ will become a miss if the number of cores is small compared to associativity α, as it is just the high number of references $H^{sd}(1)$ that imply that the next reference will go to the same cache line and a single entry on the LRU stack will satisfy a huge number (i.e. $H^{sd}(1)$) of references. Even if this LRU stack entry is pushed back some stack positions by references of other applications, it is the high number of $H^{sd}(1)$ which makes it very likely that this LRU stack entry will shortly be moved to LRU position 1 and other applications will not be able to displace it from the LRU stack. References that correspond to entries $H^{sd}(\delta)$ with a high number of δ however will very likely result in misses: First, they have a unfavorable LRU position even on stand-alone execution and it takes only a few references from other applications to displace them from the LRU stack. But another aspect might be more important: Due to the high stack distance histogram concentration, references to higher LRU stack distance positions occur less often than references to lower LRU stack positions, which means that there is a lot of time available for other applications to displace an entry from the LRU stack. Therefore, references with high δ seem to be much more likely to generate contention misses than references with low δ. However, as $H^{sd}(\delta)$ is very low for high values of δ (remember that $H^{sd}(\delta) \gg H^{sd}(\delta+1)$), even a high percental change in $H^{sd}(\delta)$ will hardly have any effect on the predictor, as changes in $H^{sd}(\delta)$ for high values of δ are generally small compared to the absolute value of $H^{sd}(1)$. To boost up effects of higher values of δ, I potentiate δ applying an exponential function and calculate the predictor for this variation by

$$p_{C_a,\iota_F i} = \sum_{s=1}^{|S|}\Big(\xi^S_{C_a,\iota_F i,s} \cdot \sum_{a' \in \{a\}\cup C_a} \sum_{\delta=1}^{\alpha} H^{sd,S}_{a',\iota_F i,s}(\delta) \cdot \beta^{\delta-1}\Big). \tag{27}$$

For my simulations I use $\beta = 10$. Note that this value was chosen without any further investigation. I apply predictors as follows:

Predictor	For all $s \in S$ and $a' \in$	Size (each a')
$\sum_{\delta=1}^{\alpha} H^{sd,S}_{a',\iota_F i,s}(\delta) \cdot \beta^{\delta-1}$	$\{a\} \cup C_a$	$\lvert S\rvert \cdot 2 \cdot l$
$\delta^{max,S}_{a',\iota_F i,s}$	$\{a\} \cup C_a$	$\lvert S\rvert \cdot l$

2.6 The Pain Method

In [Zhuravlev et al., 2010] and [Fedorova et al., 2010], Sergey Zhuravlev, Alexandra Fedorova and Sergey Blagodurov present their so-called *Pain* method that estimates co-scheduling penalties by *cache sensitivity* $\chi^{sens}_{a,\iota_{F}i}$ and *cache intensity* $\chi^{ins}_{a,\iota_{F}i}$ of applications. *Cache sensitivity* is the extent an application is sensitive to cache contention when sharing the processor cache with another application. To determine cache sensitivity, the authors multiply each cache hit by its probability to become a miss under cache sharing. To estimate hit probability, Zhuravlev et al. employ the information, how recently the corresponding cache line has been referenced: If a memory access references a cache line that resides on top of the LRU stack, i.e. $\delta = 1$, it is very unlikely that this reference will turn into a miss under cache sharing. In contrast, memory accesses that reference cache lines that have not been referenced for a long time and reside near the bottom of the LRU stack (e.g. $\delta = \alpha$) are very likely to be replaced by a co-scheduled application. Generally, the authors assume a linear loss probability and calculate the probability that an access to a cache line of LRU stack position δ will become a miss in an α way set associative cache by $\delta/(1+\alpha)$. Zhuravlev et al. define sensitivity $\chi^{sens}_{a,\iota_{F}i}$ of an application a in interval $\iota_{F}i$ as

$$\chi^{sens}_{a,\iota_{F}i} = \sum_{\delta=1}^{\alpha} \frac{\delta}{1+\alpha} \cdot H^{sd}_{a,\iota_{F}i}(\delta). \tag{28}$$

Contrary to cache sensitivity, *cache intensity* is a measure of the intensity an application uses cache: Memory intensive applications are more likely to replace cache lines of a co-scheduled application than applications with low memory usage. Zhuravlev et al. define cache intensity as the number of cache accesses for a fixed amount of instruction. As each interval $\iota_{F}i$ represents a fixed amount of instruction and as I compare only prediction results of the same window size F to one another, I calculate intensity $\chi^{int}_{a,\iota_{F}i}$ by

$$\chi^{int}_{a,\iota_{F}i} = \sum_{\delta=1}^{\alpha+1} H^{sd}_{a,\iota_{F}i}(\delta). \tag{29}$$

Given cache sensitivity and cache intensity, the authors estimate the so-called *Pain* introduced to an application a by a co-scheduled application c_a by

$$Pain = \chi^{sens}_{a,\iota_{F}i} \cdot \chi^{int}_{c_a,\iota_{F}i}. \tag{30}$$

Variation 'one'

With variation 'one', I present the original method as it has been proposed in [Zhuravlev et al., 2010]. As the variations' name suggests, this variation employs only a single sensitivity/intensity descriptor per interval ι_{Fi} for each application and prediction $p_{C_a,\iota_{F}i}$ calculates to

$$p_{C_a,\iota_{F}i} = \chi^{sens}_{a,\iota_{F}i} \cdot \sum_{c_a \in C_a} \chi^{int}_{c_a,\iota_{F}i}. \tag{31}$$

For my calculations, I apply the following predictors:

Predictor	For all $a' \in$	Size (each a')
$\chi^{sens}_{a',\iota_{F}i}$	$\{a\}$	l
$\chi^{int}_{a',\iota_{F}i}$	C_a	l

Variation 'one, sens38'

With Pain variation 'one, sens38', I pick up my ideas presented in variation 'set, mask, exp delta' of the 'Width' method regarding highly concentrated stack distance histograms, i.e. $H^{sd}(\delta) \gg H^{sd}(\delta+1)$ if $H^{sd}(\delta+1) \neq 0$. Instead of introducing an exponential function to make high values of δ more effective as presented before, I simply omit this time $H^{sd}(\delta)$ values with $\delta = 1$ and $\delta = 2$, i.e. I calculate application sensitivity only by stack distance entries $3 \ldots \alpha$. Given $\alpha = 8$, I calculate $\chi^{sens38}_{a,\iota_{F}i}$ by

$$\chi^{sens38}_{a,\iota_{F}i} = \sum_{\delta=3}^{\alpha} \frac{\delta}{1+\alpha} H^{sd}_{a,\iota_{F}i}(\delta) \tag{32}$$

and calculate prediction $p_{C_a,\iota_{F}i}$ by

$$p_{C_a,\iota_{F}i} = \chi^{sens38}_{a,\iota_{F}i} \cdot \sum_{c_a \in C_a} \chi^{int}_{c_a,\iota_{F}i}. \tag{33}$$

I apply the following predictors:

Predictor	For all $a' \in$	Size (each a')
$\chi^{sens38}_{a',\iota_{F}i}$	$\{a\}$	l
$\chi^{int}_{a',\iota_{F}i}$	C_a	l

Variation 'one, misses'

In [Zhuravlev et al., 2010], the authors examine the impact that co-scheduled applications $c_a \in C_a$ have on an application a by means of the number of references of each application $c_a \in C_a$; they refer to these references as *intensity* (cf. equation 29). Regarding the distribution of stack distances in a stack distance histogram, their intensity measure is highly dominated by $H^{sd}_{c_a,\iota_{F\,i}}(1)$. However, references with $H^{sd}_{c_a,\iota_{F\,i}}(1)$ are very unlikely to displace a reference of an application a from the LRU stack, as they are very *unlikely not to reside on the LRU stack*: Consider a memory reference m_a of an application a with (stand-alone) LRU distance $\delta = 1$ that, for example, occupies position $\delta = 4$ on a (shared) LRU stack. Then, referencing m_a moves the corresponding cache key from LRU stack position $\delta = 4$ to LRU stack position $\delta = 1$, repositioning entries $1 \ldots 3$ to new LRU stack positions $2 \ldots 4$. This example shows that referencing an already cached element *will not displace any element* from the LRU stack, but only reposition elements with a better LRU stack position by 1 towards position $\delta = \alpha$. To kick out elements from the LRU stack (a prerequisite to generate contention misses), however, an element has to be fetched that *does not currently reside on the shared cache.* This is more likely for (stand-alone) references with high δ, but definite for references with $\delta > \alpha$, i.e. *references that are misses even on stand-alone execution.* Therefore, references with a stand-alone LRU stack distance of $\delta = \alpha + 1$ seem to be a much better selection for *intensity* than simply taking the number of memory references, as it has been done in [Zhuravlev et al., 2010]. Further note that there are often far more references with (stand-alone) LRU stack distance $\delta > \alpha$ than there are references with $\delta = 3$, or sometimes even $\delta = 2$. In order to verify my considerations, I apply the number of misses $H^{sd}_{c_a,\iota_{F\,i}}(\alpha+1)$, $c_a \in C_a$, as intensity measure and calculate prediction $p_{C_a,\iota_{F\,i}}$ for 'Pain' variation 'one, misses' according to

$$p_{C_a,\iota_{F\,i}} = \chi^{sens}_{a,\iota_{F\,i}} \cdot \sum_{c_a \in C_a} H^{sd}_{c_a,\iota_{F\,i}}(\alpha+1). \tag{34}$$

I apply the following predictors for this variation of the 'Pain' method:

Predictor	For all $a' \in$	Size (each a')
$\chi^{sens}_{a',\iota_{F\,i}}$	$\{a\}$	l
$H^{sd}_{a',\iota_{F\,i}}(\alpha+1)$	C_a	l

Variation 'one, sens38, misses'

With 'Pain' variation 'one, sens38, misses', I combine my considerations regarding variations 'one, sens38' and 'one, misses' and calculate prediction $p_{C_a,\iota_F i}$ according to

$$p_{C_a,\iota_F i} = \chi^{sens38}_{a,\iota_F i} \cdot \sum_{c_a \in C_a} H^{sd}_{c_a,\iota_F i}(\alpha+1). \tag{35}$$

I apply the following predictors:

Predictor	For all $a' \in$	Size (each a')
$\chi^{sens38}_{a',\iota_F i}$	$\{a\}$	l
$H^{sd}_{a',\iota_F i}(\alpha+1)$	C_a	l

Variation 'set'

With variation 'set', I investigate the maximum improvement that can be achieved with Zhuravlev et al.'s method when additionally accounting for spatial locality. Therefore, variation 'set' calculates sensitivity and intensity on a per-cache-set basis according to

$$\chi^{sens,S}_{a,\iota_F i,s} = \sum_{\delta=1}^{\alpha} \frac{\delta}{1+\alpha} \cdot H^{sd,S}_{a,\iota_F i,s}(\delta) \tag{36}$$

and

$$\chi^{int,S}_{a,\iota_F i,s} = \sum_{\delta=1}^{\alpha+1} H^{sd,S}_{a,\iota_F i,s}(\delta). \tag{37}$$

Given $\chi^{sens,S}_{a,\iota_F i,s}$ and $\chi^{int,S}_{c_a,\iota_F i,s}$ for each $c_a \in C_a$ for all $s \in S$, I calculate $p_{C_a,\iota_F i}$ according to

$$p_{C_a,\iota_F i} = \sum_{s=1}^{|S|} \big(\chi^{sens,S}_{a,\iota_F i,s} \cdot \sum_{c_a \in C_a} \chi^{int,S}_{c_a,\iota_F i,s}\big). \tag{38}$$

I apply the following predictors:

Predictor	For all $a' \in$	Size (each a')
$\chi^{sens,S}_{a',\iota_F i,s}$	$\{a\}$	$\lvert S\rvert \cdot l$
$\chi^{int,S}_{a',\iota_F i,s}$	C_a	$\lvert S\rvert \cdot l$

Variation ‘set, misses’

With variation ‘set, misses’, I enhance variation ‘one, misses’ by considering spatial locality given by the various cache sets and determine predictor $p_{C_a,\iota_F\,i}$ by

$$p_{C_a,\iota_F\,i} = \sum_{s=1}^{|S|} \big(\chi^{sens,S}_{a,\iota_F\,i,s} \cdot \sum_{c_a \in C_a} H^{sd,S}_{c_a,\iota_F\,i,s}(\alpha+1)\big), \tag{39}$$

applying predictors as follows.

Predictor	For all $a' \in$	Size (each a')
$\chi^{sens,S}_{a',\iota_F\,i,s}$	$\{a\}$	$\lvert S\rvert \cdot l$
$H^{sd,S}_{c_a,\iota_F\,i,s}(\alpha+1)$	C_a	$\lvert S\rvert \cdot l$

Variation ‘set, sens38, misses’

With variation ‘set, sens38, misses’, I combine spatial locality with the considerations presented in variations ‘one, sens38’ and ‘one, misses’. Therefore, I calculate a per-set-version of sensitivity $\chi^{sens38}_{a,\iota_F\,i}$ by

$$\chi^{sens38,S}_{a,\iota_F\,i,s} = \sum_{\delta=3}^{\alpha} \frac{\delta}{1+\alpha} H^{sd,S}_{a,\iota_F\,i,s}(\delta) \tag{40}$$

and calculate predictor $p_{C_a,\iota_F\,i}$ by

$$p_{C_a,\iota_F\,i} = \sum_{s=1}^{|S|} \left(\chi^{sens,S}_{a,\iota_F\,i,s} \cdot \sum_{c_a \in C_a} H^{sd,S}_{c_a,\iota_F\,i}(\alpha+1)\right). \tag{41}$$

I apply the following predictors:

Predictor	For all $a' \in$	Size (each a')
$\chi^{sens38,S}_{a',\iota_F\,i,s}$	$\{a\}$	$\lvert S\rvert \cdot l$
$H^{sd,S}_{a',\iota_F\,i,s}(\alpha+1)$	C_a	$\lvert S\rvert \cdot l$

2.7 The Misses Method

In [Knauerhase et al., 2008], the authors investigate methods to improve scheduling and load balancing on multicore processors. To improve caching performance and reduce cache contention, they sample performance counters such as `INVALID_L2_RQSTS` (L2 cache misses) and `L2_RQSTS` (L2 cache references) on a per-thread basis and try to use this data to improve co-scheduling. They discovered that the number of cache misses per cycle performs best as indicator of cache interference. However, this does not seem to be a surprise, as the number of L2 cache misses observed have been measured *at runtime*, when the L2 cache *is already shared* among instructions; therefore, gathering values from `INVALID_L2_RQSTS` is a *measurement*, and not a *prediction*. For Knauerhase et al.'s intent to improve scheduling performance from runtime measures, this approach seems to be a good solution. But for cache contention *prediction*, you would have to run each candidate set of co-scheduled applications in parallel, measure cache contention and predict cache contention of future execution intervals from previous intervals. This does not seem to be a reasonable approach, in particular for larger prediction intervals, as program behavior changes over time [Sherwood et al., 2003, Zwick et al., 2009b, Zwick, 2010b].

In order to use cache misses as a *predictor*, it seems to be worth investigating if the number of *stand-alone* cache misses can be applied to predict cache contention of shared applications. This seems to be a promising approach, as stack distance histograms are often highly concentrated, i.e. $H^{sd}_{a,\iota_{F\,i}}(\delta) \gg H^{sd}_{a,\iota_{F\,i}}(\delta+1)$ for small values of δ, and $H^{sd}_{a,\iota_{F\,i}}(\delta) = 0$ for most other values of δ up to α. $H^{sd}_{a,\iota_{F\,i}}(\alpha+1)$ then often has a value in the range of about $H^{sd}_{a,\iota_{F\,i}}(2)$. See figure 12 for an example of a typical shape of such a stack distance histogram; assume that the ordinate is of logarithmic scale. If stack distance concentration is that high that, for a given cache set $s \in S$, $\sum_{a' \in \{a\} \cup C_a} \delta^{\max,S}_{a',\iota_{F\,i},s} \leq \alpha$, then it is not possible that any stand-alone cache *hit* of an application $c_a \in C_a$ will render any stand-alone cache hit of application $a \notin C_a$ into a miss. In this case, cache contention introduced to application a can soleley originate from memory references being stand-alone cache *misses*.

Proof: Let s be the considered cache set. Assume each $a' \in \{a\} \cup C_a$ to have a specific $\delta^{\max,S}_{a',\iota_{F\,i},s}$ and let $\forall_{a' \in \{a\} \cup C_a} : H^{sd,S}_{a',\iota_{F\,i},s}(\alpha+1) = 0$. Then, the worst possible LRU stack position a reference of $a' \in \{a\} \cup C_a$ can earn is $\sum_{a' \in \{a\} \cup C_a} \delta^{\max,S}_{a',\iota_{F\,i},s}$, which is $\leq \alpha$,

as assumed, and therefore a cache hit. The only way, for such values of $\delta^{\max,S}_{a',\iota_{F}i,s}$, $a' \in \{a\} \cup C_a$, to turn stand-alone cache hits of application a into contention misses is with stand-alone cache misses. Therefore, if $\sum_{a' \in \{a\} \cup C_a} \delta^{\max,S}_{a,\iota_{F}i,s} \leq \alpha$, which is likely due to the high concentration of stack distance histogram values, contention misses can only be introduced from stand-alone cache misses. □

Note the difference to variation 'one, misses' of the Pain method: Applications $c_a \in C_a$ are considered by their amount of stand-alone cache misses $H^{sd}_{c_a,\iota_{F}i}(\alpha+1)$ in both methods Pain and Misses. Application a however is considered differently: In Pain variation 'one, misses', a is considered by the *sensitivity* measure obtained from elements $H^{sd}_{a,\iota_{F}i}(\delta)$, $1 \leq \delta \leq \alpha$. In the Misses method however, I consider application a by $H^{sd}_{a,\iota_{F}i}(\alpha+1)$, i.e. just the same way as applications $c_a \in C_a$. This seems to be a valid approach, however, as stand-alone misses of application a are related to elements $H^{sd}_{a,\iota_{F}i}(\delta)$, $2 \leq \delta \leq \alpha$: Each $H^{sd}_{a,\iota_{F}i}(\alpha+1)$ repositions LRU stack entries of the corresponding cache set by 1 towards the LRU position and therefore also accounts for stack distance histogram positions $H^{sd}_{a,\iota_{F}i}(\delta)$ with $\delta \geq 2$. To $H^{sd}_{a,\iota_{F}i}(1)$, entries $H^{sd}_{a,\iota_{F}i}(\alpha+1)$ do not contribute, which seems to be advantageous for cache contention prediction, as entries $H^{sd}_{a,\iota_{F}i}(1)$ are unlikely to turn into contention misses (cf. discussion about variation 'set, mask, exp delta' of the Width method).

Variation 'one'

With variation 'one', I apply the number of stand-alone cache misses obtained from applications $\{a\} \cup C_a$ as a predictor for cache contention and calculate $p_{C_a,\iota_{F}i}$ by

$$p_{C_a,\iota_{F}i} = \sum_{a' \in \{a\} \cup C_a} H^{sd}_{a,\iota_{F}i}(\alpha+1). \tag{42}$$

I apply the following predictor:

Predictor	For all $a' \in$	Size (each a')
$H^{sd}_{a',\iota_{F}i}(\alpha+1)$	$\{a\} \cup C_a$	l

Variation 'set, mask'

With variation 'set, mask', I investigate whether or not a better prediction accuracy can be achieved by considering only misses of those cache sets s that are able to introduce contention misses even from stand-alone cache *hits*, i.e. $\sum_{a' \in \{a\} \cup C_a} \delta^{max,S}_{a',\iota_F i,s} > \alpha$. If prediction gets worse compared to variation 'one', then cache contention will be caused rather by stand-alone misses than by stand-alone hits. For this investigation, I calculate prediction $p_{C_a,\iota_F i}$ by

$$p_{C_a,\iota_F i} = \sum_{s=1}^{|S|} \xi^{S}_{C_a,\iota_F i,s} \cdot \sum_{a' \in \{a\} \cup C_a} H^{sd,S}_{a',\iota_F i,s}(\alpha+1), \tag{43}$$

where $\xi^{S}_{C_a,\iota_F i,s}$ is calculated as presented in equation 10. I apply the following predictors:

Predictor	For all $s \in S$ and $a' \in$	Size (each a')
$H^{sd,S}_{a',\iota_F i,s}(\alpha+1)$	$\{a\} \cup C_a$	$\lvert S\rvert \cdot l$
$\delta^{max,S}_{a',\iota_F i,s}$	$\{a\} \cup C_a$	$\lvert S\rvert \cdot l$

2.8 The Miss Rate Method

In [Fedorova et al., 2010], the authors compare their 'Pain' method (cf. section 2.6) to the results achieved when predicting cache contention by means of stand-alone cache miss rates of the co-scheduled applications. They did so despite the "stronger evidence in favor of the memory-reuse approach" [Fedorova et al., 2010] they applied for their 'Pain' method. Their evaluation showed, however, that *stand-alone* miss rate is a surprisingly good method to predict contention of a *shared* cache.

In order to compare the miss rate based approach to the other techniques presented in this thesis, I introduce the 'Miss Rate' method that calculates its prediction as follows:

$$p_{C_a,\iota_F i} = \sum_{a' \in \{a\} \cup C_a} \frac{H^{sd}_{a',\iota_F i}(\alpha+1)}{\sum_{\delta=1}^{\alpha+1} H^{sd}_{a',\iota_F i}(\delta)}, \tag{44}$$

where I calculate the fraction in advance to the prediction and use it as predictor.

Predictor	For all $a' \in$	Size (each a')
$\frac{H^{sd}_{a',\iota_F i}(\alpha+1)}{\sum_{\delta=1}^{\alpha+1} H^{sd}_{a',\iota_F i}(\delta)}$	$\{a\} \cup C_a$	l

2.9 The Activity Vector Method

In [Kihm and Connors, 2004], [Kihm et al., 2005] and [Settle et al., 2004], the authors propose a method to predict L2 cache contention they call *Activity Vector*. The method exploits the observation that a program's cache utilization does not only vary in time, but also in space: Some parts of the cache show high activity, while others show low activity, as it is exemplarily presented in figure 10. Generally, activity vectors are bit vectors that indicate, for a given execution interval, if a group of cache sets shows *high activity* (1) or *low activity* (0).

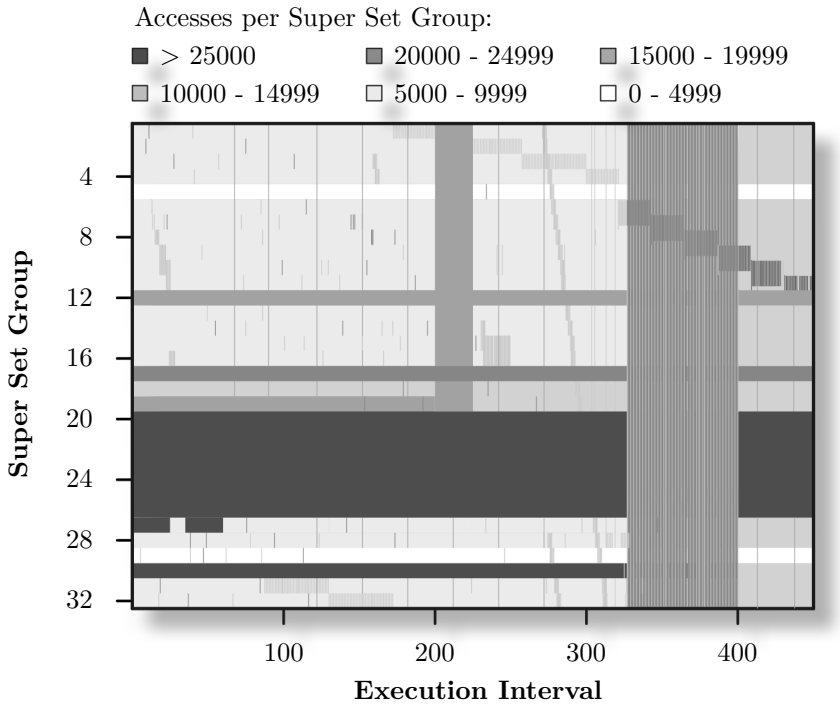


Figure 10: Activity map as it is similarly presented in [Settle et al., 2004] to demonstrate spatial cache utilization (horizontal stripes of similar color). The *abscissa* represents instruction intervals, i.e. timing information; the *color* indicates *cache access activity* (dark color = high activity; bright color = low activity). The *ordinate* represents 32 so-called *super sets*, a series of contiguous groups of cache sets that represent space/location information.

To generate activity vectors, the authors track the *number of accesses* to groups $G = \{g_1, \ldots, g_{|G|}\}$ of contiguous cache sets, called *super sets*. Further, they track the *number of cache misses* per group. If, for an application a, the *accesses* to group g in execution interval ι_{Fi} exceed threshold $\Omega_F^{acc,G}$, then element g in the so-called *access activity vector* $\aleph_{a,\iota_{Fi}}^{acc,G} = \left[\aleph_{a,\iota_{Fi},1}^{acc,G} \ \ldots \ \aleph_{a,\iota_{Fi},|G|}^{acc,G}\right]$ is set to one, i.e. $\aleph_{a,\iota_{Fi},g}^{acc,G} = 1$; otherwise, $\aleph_{a,\iota_{Fi},g}^{acc,G} = 0$. If the

misses of application a in interval $\iota_{F}i$ that occur in group g exceed threshold $\Omega_{F}^{miss,G}$, then element g in the so-called *miss activity vector* $\aleph_{a,\iota_{F}i}^{miss,G} = \left[\aleph_{a,\iota_{F}i,1}^{miss,G} \ \ldots \ \aleph_{a,\iota_{F}i,|G|}^{miss,G}\right]$ is set to one, i.e. $\aleph_{a,\iota_{F}i,g}^{miss,G} = 1$, otherwise $\aleph_{a,\iota_{F}i,g}^{miss,G} = 0$. The activity vector that contains both accesses and misses is defined by $\aleph_{a,\iota_{F}i}^{G} = \left[\aleph_{a,\iota_{F}i}^{acc,G} \ \aleph_{a,\iota_{F}i}^{miss,G}\right]$. Kihm and Connors analyze various decision points to differentiate between *high* and *low* activity; they discover that good results can be achieved when applying the third quartile as offset [Kihm and Connors, 2004].

Let $\Omega = \frac{3}{4}$, let $|G| = 32$ be the number of groups, let $|M_a|$ be the total number of memory references of application a in the set of intervals $\iota_{F} = \{\iota_{F}1, \ldots \iota_{F|\iota_{F}|}\}$, let $A = \{$astar, bzip2, gcc, gobmk, h264ref, hmmer, lbm, mcf, milc, povray$\}$ be the set of applications I apply for my simulations, and let $|A|$ be the number of applications in A. I define thresholds $\Omega_{F}^{acc,G}$ and $\Omega_{F}^{miss,G}$ by

$$\Omega_{F}^{acc,G} = \frac{\Omega}{|A| \cdot |G|} \sum_{a \in A} (|M_a|/F) \tag{45}$$

and

$$\Omega_{F}^{miss,G} = \frac{\Omega}{|A| \cdot |G|} \cdot \sum_{a \in A} \frac{F}{|M_a|} \sum_{\iota_{F}i \in \iota_{F}} H_{a,\iota_{F}i}^{sd}(\alpha + 1). \tag{46}$$

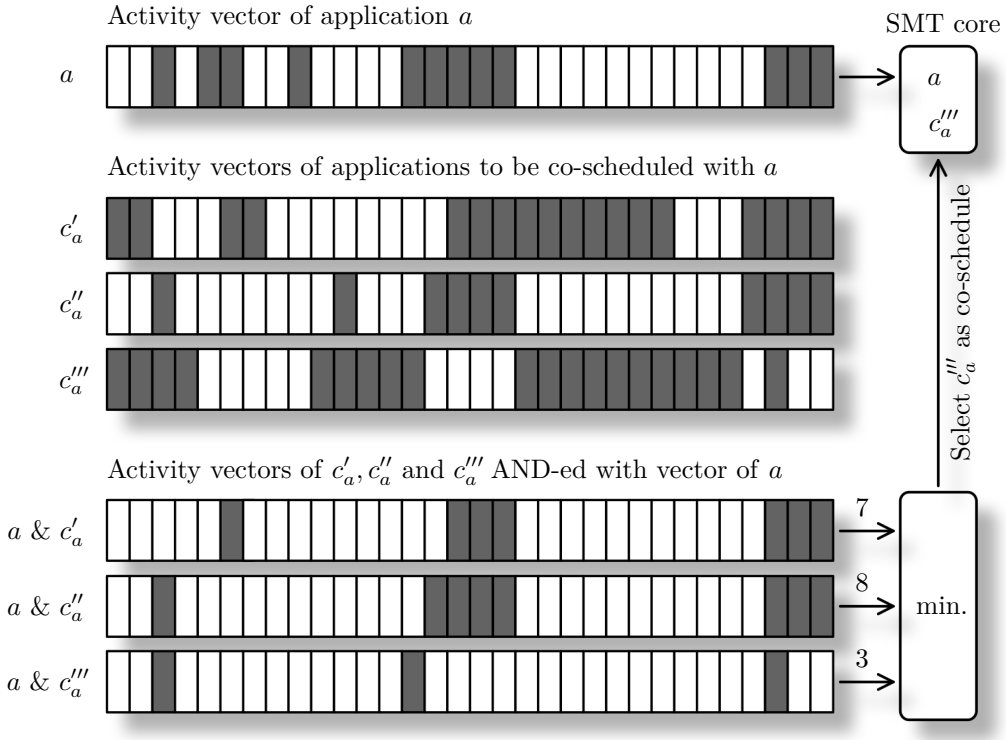


Figure 11: AND-ing (&) activitiy vectors to select the best co-schedule for a thread, as it is similarly shown in [Settle et al., 2004]. Bits filled with grey color indicate bits set to 1; bits filled with white color indicate bits set to 0.

Figure 11 shows how activity vectors are applied to select the best co-schedule for an application a from a set of candidate co-schedules $\{c'_a, c''_a, c'''_a\}$ to minimize overall cache contention. To determine the best co-schedule, the activity vector of a is bitwise logically AND-ed with the activity vectors of the candidate co-schedules. After AND-ing the vectors, the bits set to 1 in the resulting vector are counted and the thread that yields the resulting vector with the least number of bits set to 1 is chosen as co-schedule. Note that in figure 11, activity vectors have dimension 32 for demonstration purposes only; Settle et al. apply activity vectors of dimension 64, as they compose an activity vector of both an *access* activity vector of dimension 32 and a *miss* activity vector of dimension 32. In the following, I present three variations of the 'Activity Vector' method.

Variation 'superset'

Variation 'superset' is the original method as it is proposed in [Settle et al., 2004]. In this variation, activity vectors $\aleph^G_{a,\iota_{Fi}}$ are calculated as presented in algorithm 6.
Given $\aleph^G_{a,\iota_{Fi}}$ for each application $\{a\} \cup C_a$, and let variable $mask =$ **false**, prediction $p_{C_a,\iota_{Fi}}$ is calculated as shown in algorithm 7.

Algorithm 6 Calculation of activity vectors $\aleph^G_{a,\iota_{Fi}}$ from stack distance histogram $H^{sd,S}_{a,\iota_{Fi}}$.

1: # —— Init ——
2: **for** $g \leftarrow 1$ **to** $|G|$ **do**
3: $\quad \aleph^{acc,G}_{a,\iota_{Fi},g} \leftarrow 0$
4: $\quad \aleph^{miss,G}_{a,\iota_{Fi},g} \leftarrow 0$
5: **end for**
6: # —— Calculate group misses/accesses ——
7: **for** $s \leftarrow 1$ **to** $|S|$ **do**
8: $\quad g \leftarrow \gamma(s)$
9: $\quad$ **for** $\delta \leftarrow 1$ **to** $\alpha + 1$ **do**
10: $\quad\quad \aleph^{acc,G}_{a,\iota_{Fi},g} \leftarrow \aleph^{acc,G}_{a,\iota_{Fi},g} + H^{sd,S}_{a,\iota_{Fi},s}(\delta)$
11: $\quad$ **end for**
12: $\quad \aleph^{miss,G}_{a,\iota_{Fi},g} \leftarrow \aleph^{miss,G}_{a,\iota_{Fi},g} + H^{sd,S}_{a,\iota_{Fi},s}(\alpha + 1)$
13: **end for**

for $g = 1$ **to** $|G|$ **do**
  if $\aleph^{acc,G}_{a,\iota_F i,g} > \Omega^{acc,G}_{F}$ **then**
    $\aleph^{acc,G}_{a,\iota_F i,g} \leftarrow 1$
  else
    $\aleph^{acc,G}_{a,\iota_F i,g} \leftarrow 0$
  end if
  if $\aleph^{miss,G}_{a,\iota_F i,g} > \Omega^{miss,G}_{F}$ **then**
    $\aleph^{miss,G}_{a,\iota_F i,g} \leftarrow 1$
  else
    $\aleph^{miss,G}_{a,\iota_F i,g} \leftarrow 0$
  end if
end for
$\aleph^{G}_{a,\iota_F i} \leftarrow \left[\aleph^{acc,G}_{a,\iota_F i} \quad \aleph^{miss,G}_{a,\iota_F i}\right]$

Algorithm 7 Calculation of $p_{C_a,\iota_F i}$ from $\aleph^{G}_{a,\iota_F i}$ according to the Activity Vector method.

$p_{C_a,\iota_F i} \leftarrow 0$
for $g = 1$ **to** $|G|$ **do**
  $\aleph \leftarrow \sum_{a' \in \{a\} \cup C_a} \aleph^{G}_{a',\iota_F i,g}$
  if $mask =$ **false** $\vee$ $\left(mask =$ **true** $\wedge\ \xi^{S}_{C_a,\iota_F i,s} = 1\right)$ **then**
    if $|C_a| = 1$ **then**
      if $\aleph == 2$ **then**
        $p_{C_a,\iota_F i} \leftarrow p_{C_a,\iota_F i} + 1$
      end if
    else if $C_a > 1$ **then**
      if $\aleph \geq \Omega \cdot \left(|C_a| + 1\right)$ **then**
        $p_{C_a,\iota_F i} \leftarrow p_{C_a,\iota_F i} + 1$
      end if
    end if
  end if
end for

To calculate $p_{C_a,\iota_{Fi}}$ in variation 'superset', I apply the following predictor for each application $a' \in \{a\} \cup C_a$ and execution interval ι_{Fi}:

Predictor	For all $a' \in$	Size (each a')
$\aleph^G_{a',\iota_{Fi}} = \left[\aleph^{acc,G}_{a',\iota_{Fi}} \;\; \aleph^{miss,G}_{a',\iota_{Fi}}\right]$	$\{a\} \cup C_a$	$2 \cdot \lvert G\rvert \cdot l$

Note that I do not store the predictors as $2 \cdot |G|$ bits, but as $2 \cdot |G|$ integers in order to avoid additional operations to extract single bits from a word. As the size of a predictor, even it it is stored as $2 \cdot |G|$ integers, is much smaller than a memory page, there should be no additional memory transaction penalty using integers instead of bits.

Variation 'set'

With variation 'set', I investigate if a much better accuracy of the activity vector method can be achieved when applying groups of cache set granularity in order to exploit maximum spatial locality. For variation 'set', I calculate predictor $\aleph^S_{a,\iota_{Fi}}$ and prediction $p_{C_a,\iota_{Fi}}$ according to algorithms 6 and 7, but with the following modifications:

- $G \mapsto S$
- $g \mapsto s$
- $\gamma(s) \mapsto s$; alternatively, line 8 in algorithm 6 can be omitted.

Similar to equations 45 and 46 I calculate per-set offsets according to

$$\Omega^{acc,S}_F = \frac{\Omega}{|A| \cdot |S|} \sum_{a \in A} (|M_a|/F) \tag{47}$$

and

$$\Omega^{miss,S}_F = \frac{\Omega}{|A| \cdot |S|} \cdot \sum_{a \in A} \frac{F}{|M_a|} \sum_{\iota_{Fi} \in \iota_F} H^{sd}_{a,\iota_{Fi}}(\alpha + 1). \tag{48}$$

I apply the following predictors:

Predictor	For all $a' \in$	Size (each a')
$\aleph^S_{a',\iota_{Fi}} = \left[\aleph^{acc,S}_{a',\iota_{Fi}} \;\; \aleph^{miss,S}_{a',\iota_{Fi}}\right]$	$\{a\} \cup C_a$	$2 \cdot \lvert S\rvert \cdot l$

Variation ‘set, mask’

In variation ‘set, mask’, I additionally add masking to variation ‘set’ to consider the activity of those cache sets s only with $\xi^{S}_{C_a,\iota_{\boldsymbol{F}}i,s} = 1$. I calculate variation ‘set, mask’ the same way as variation ‘set’, but with variable *mask* = **true** and $\xi^{S}_{C_a,\iota_{\boldsymbol{F}}i,s}$ as calculated by equation 10 and algorithm 2. I apply the following predictors for each application $a' \in \{a\} \cup C_a$ and execution interval $\iota_{\boldsymbol{F}i}$.

Predictor	For all $a' \in$	Size (each a’)
$\aleph^{S}_{a',\iota_{\boldsymbol{F}}i} = \left[\aleph^{acc,S}_{a',\iota_{\boldsymbol{F}}i} \quad \aleph^{miss,S}_{a',\iota_{\boldsymbol{F}}i}\right]$	$\{a\} \cup C_a$	$2 \cdot \lvert S\rvert \cdot l$
$\delta^{max,S}_{a',\iota_{\boldsymbol{F}}i,s}$	$\{a\} \cup C_a$	$\lvert S\rvert \cdot l$

2.10 The DMax Method

In [Zwick et al., 2010] and [Zwick, 2011], I proposed the *Setvector* method to predict cache contention. The Setvector method is a heuristic method that combines the ideas of *stack distances* (temporal locality) and *activity vectors* (spatial locality). In this section, I present the *DMax* method to further investigate ideas integrated in the *Setvector* method. Analyzing stack distance histograms, I discovered that they are often highly concentrated, i.e. $H^{sd}_{a,\iota_{\mathcal{F}}i}(\delta) \gg H^{sd}_{a,\iota_{\mathcal{F}}i}(\delta+1)$ for small values of δ and that larger values of δ often result in $H^{sd}_{a,\iota_{\mathcal{F}}i}(\delta) = 0$. This observation conforms to the property of high temporal program locality. For example, consider interval $\iota_{\mathcal{F}i}$ of application a to achieve a hitrate of 98% on a 2 way set associative level 1 cache. If the same interval $\iota_{\mathcal{F}i}$ of application a would be executed on a processor with an 8 way set associative cache, then $H^{sd}_{a,\iota_{\mathcal{F}}i}(1)$ and $H^{sd}_{a,\iota_{\mathcal{F}}i}(2)$ would hold 98% of all memory references, while $H^{sd}_{a,\iota_{\mathcal{F}}i}(3) \ldots H^{sd}_{a,\iota_{\mathcal{F}}i}(9)$ would hold only 2% of the references. As a consequence, in stack distance histograms for caches with high associativity, there are often only the first few entries different from 0, as it is exemplarily shown in figure 12. If there is one stack distance histogram for each cache set s to exploit spatial locality, each histogram $H^{sd,S}_{a,\iota_{\mathcal{F}}i,s}$ holds only a small part of $H^{sd}_{a,\iota_{\mathcal{F}}i}$ and, depending on the distribution of cache set references, many histograms $H^{sd,S}_{a,\iota_{\mathcal{F}}i,s}$ will have an even lower number of non-zero histogram entries than $H^{sd}_{a,\iota_{\mathcal{F}}i}$ has.

In the following, I refer to the number of non-zero elements in stack distance histograms $H^{sd}_{a,\iota_{\mathcal{F}}i}$ and $H^{sd,S}_{a,\iota_{\mathcal{F}}i,s}$ as $\delta^{\max}_{a,\iota_{\mathcal{F}}i}$ (cf. figure 12) and $\delta^{\max,S}_{a,\iota_{\mathcal{F}}i,s}$ respectively. With algorithm 2, I already presented the way to calculate $\delta^{max,S}_{a,\iota_{\mathcal{F}}i,s}$ from a stack distance histogram $H^{sd,S}_{a,\iota_{\mathcal{F}}i,s}$ ($H^{sd,S}_{a,\iota_{\mathcal{F}}i,s}$ can be calculated as presented in. Given a stack distance histogram $H^{sd}_{a,\iota_{\mathcal{F}}i}$, algorithm 8 shows how to calculate $\delta^{\max}_{a,\iota_{\mathcal{F}}i}$. Note that $\delta^{\max}_{a,\iota_{\mathcal{F}}i} = max(\delta^{\max,S}_{\iota_{\mathcal{F}}i,1}, \delta^{\max,S}_{\iota_{\mathcal{F}}i,2}, \ldots \delta^{\max,S}_{\iota_{\mathcal{F}}i,|S|})$, as $H^{sd}_{a,\iota_{\mathcal{F}}i}(\delta) = \sum_{s=1}^{|S|} H^{sd,S}_{a,\iota_{\mathcal{F}}i,s}(\delta)$.

Algorithm 8 Calculating predictor $\delta^{\max}_{a,\iota_{\mathcal{F}}i}$ from $H^{sd}_{a,\iota_{\mathcal{F}}i}$.

1: $\delta^{\max}_{a,\iota_{\mathcal{F}}i} \leftarrow \alpha$
2: **while** $\delta^{\max}_{a,\iota_{\mathcal{F}}i} > 0 \wedge H^{sd}_{\iota_{\mathcal{F}}i}(\delta^{\max}_{a,\iota_{\mathcal{F}}i}) = 0$ **do**
3: $\quad \delta^{\max}_{a,\iota_{\mathcal{F}}i} \leftarrow \delta^{\max}_{a,\iota_{\mathcal{F}}i} - 1$
4: **end while**

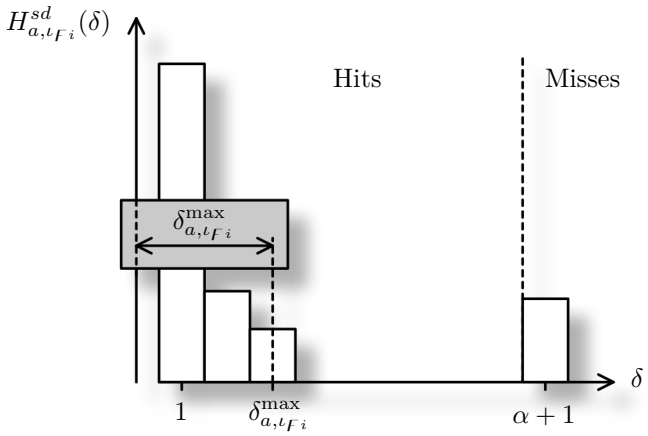


Figure 12: In many stack distance histograms $H^{sd}_{a,\iota_{\mathcal{F}}i}$, only the first few entries are different from 0.

Regarding stack distance histograms of multiple applications $\{a\} \cup C_a$ that get co-scheduled sharing a common cache, it seems to be suggested that low values of $\delta^{\max}_{a',\iota_{\mathcal{F}}i}$ or $\delta^{\max,S}_{a',\iota_{\mathcal{F}}i,s}$, $a' \in \{a\} \cup C_a$, will result in lower cache contention than high values of $\delta^{\max}_{a',\iota_{\mathcal{F}}i}$ or $\delta^{\max,S}_{a',\iota_{\mathcal{F}}i,s}$: Given a cache set s and an application a that achieves a specific $\delta^{\max,S}_{a,\iota_{\mathcal{F}}i,s}$ in interval $\iota_{\mathcal{F}}i$, then cache set s can host accesses of other applications $a' \in C_a$ that (together) reference up to $\alpha - \delta^{\max,S}_{a,\iota_{\mathcal{F}}i,s}$ additional cache lines in s with arbitrary access pattern without rendering any cache hit of application a into a miss. Therefore, larger values of $\delta^{\max,S}_{a,\iota_{\mathcal{F}}i,s}$ decrease the number of additional cache lines that can be accessed from any $a' \in C_a$ using an arbitrary access pattern without introducing additional cache misses to a.

In the following, I present several ways I investigate to predict cache contention with $\delta^{\max}_{a,\iota_{\mathcal{F}}i}$ and $\delta^{\max,S}_{a,\iota_{\mathcal{F}}i,s}$ respectively. Note that solely applying $\delta^{\max}_{a,\iota_{\mathcal{F}}i}$ and $\delta^{\max,S}_{a,\iota_{\mathcal{F}}i,s}$ as predictor implies that contention misses introduced to an application a due to co-scheduling a with C_a are predicted by means of stand-alone cache *hits* of the applications in C_a only. Memory references in C_a that are cache *misses* even on stand-alone execution are *not* considered in this case.

Variation 'one'

With variation 'one', I investigate the effectiveness of summing up $\delta^{\max}_{a',\iota_{Fi}}$ of all co-scheduled applications $a' \in \{a\} \cup C_a$ as a measure to predict cache contention.

Given the set of applications $\{a\} \cup C_a$, application a still has enough cache space available in order to not suffer from cache contention misses introduced from memory references in C_a that correspond to stand-alone cache hits if $\sum_{a' \in \{a\} \cup C_a} \delta^{\max}_{a',\iota_{Fi}} \leq \alpha$. This characteristic is obvious, as $\sum_{a' \in \{a\} \cup C_a} \delta^{\max}_{a',\iota_{Fi}} \leq \alpha \Rightarrow \forall s \in S : \sum_{a' \in \{a\} \cup C_a} \delta^{\max,S}_{a',\iota_{Fi},s} \leq \alpha$; this means, in turn, that references in C_a that correspond to stand-alone cache hits can shift an entry of a on the LRU stack of a cache set s by a maximum amount of $\delta = \sum_{a' \in C_a} \delta^{\max,S}_{a',\iota_{Fi},s}$ positions, which is not a sufficient distance to displace any entry of a from the LRU stack of cache set s.

If $\sum_{a' \in \{a\} \cup C_a} \delta^{\max}_{a',\iota_{Fi}} > \alpha$, it is still *possible* that stand-alone hits of applications in C_a do not introduce any contention misses to a: First of all, not all cache sets $s \in S$ do have the same $\delta^{\max,S}_{a',\iota_{Fi}}$ and therefore $\delta^{\max}_{a_1,\iota_{Fi}} + \delta^{\max}_{a_2,\iota_{Fi}} > \alpha$ does not necessarily mean that $\exists s \in S : \delta^{\max,S}_{a_1,\iota_{Fi},s} + \delta^{\max,S}_{a_2,\iota_{Fi},s} > \alpha$.

Secondly, accesses with distance $\delta^{\max,S}_{a',\iota_{Fi},s}$ might be distributed differently in ι_{Fi} for applications $a' \in \{a\} \cup C_a$. As an example, figure 13 shows a distribution of stack distances of two applications a and c_a in interval ι_{Fi} for a cache set s. Although $\delta^{\max,S}_{a,\iota_{Fi},s} + \delta^{\max,S}_{c_a,\iota_{Fi},s} = 3+3 = 6$,

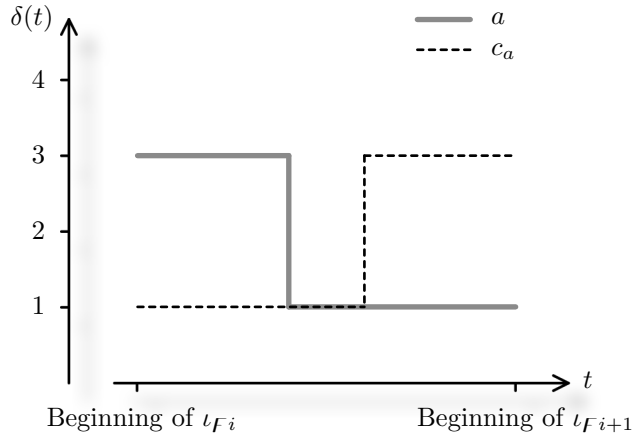


Figure 13: Example of maximum LRU stack distances an application references over time.

it is obvious that a cache with associativity $\alpha = 4$ would be sufficient to avoid any conflict misses in cache set s arising from contention of memory references that represent stand-alone cache hits of a and c_a in interval ι_{Fi}. However, assuming stack distance histograms with comparable distributions (over time) and $\sum_{a' \in \{a\} \cup C_a} \delta^{\max,S}_{a',\iota_F i,s} > \alpha$, then larger values of $\sum_{a' \in \{a\} \cup C_a} \delta^{\max,S}_{a',\iota_F i,s}$ seem more likely to correspond to a high probability of conflict misses introduced from cache hits than lower values do. Since the highest value of $\delta^{\max,S}_{a,\iota_F i,s}$ for all $s \in S$ might dominate cache contention, I investigate the ability of $\sum_{a' \in \{a\} \cup C_a} \delta^{\max}_{a',\iota_F i}$ to predict cache contention and calculate prediction $p_{C_a,\iota_F i}$ for this variation according to

$$p_{C_a,\iota_F i} = \sum_{a' \in \{a\} \cup C_a} \delta^{\max}_{a',\iota_F i}. \tag{49}$$

I apply the following predictors:

Predictor	For all $a' \in$	Size (each a')
$\delta^{max}_{a',\iota_F i}$	$\{a\} \cup C_a$	l

Variation 'one, set'

With variation 'one, set', I investigate if predicting cache contention by the number of non-zero entries in stack distance histograms achieves significantly better results applying cache set granularity. Therefore, I calculate prediction $p_{C_a,\iota_F i}$ by

$$p_{C_a,\iota_F i} = \sum_{a' \in \{a\} \cup C_a} \sum_{s=1}^{|S|} \delta^{\max,S}_{a',\iota_F i,s}. \tag{50}$$

Note that $\sum_{s=1}^{|S|} \delta^{\max,S}_{a',\iota_F i,s}$ can be calculated before runtime and I apply the following predictors for my evaluation:

Predictor	For all $a' \in$	Size (each a')
$\sum_{s=1}^{\lvert S\rvert} \delta^{\max,S}_{a',\iota_F i,s}$	$\{a\} \cup C_a$	l

Variation 'one, set, inf'

With variations 'one' and 'one, set', I investigate if the number of non-zero entries $\delta^{\max}_{a,\iota_F i}$ (and $\delta^{\max,S}_{a,\iota_F i,s}$ respectively) in a stack distance histogram can be used to predict cache contention. A given value of $\delta^{\max}_{a,\iota_F i}$ might suggest that an application a in interval $\iota_F i$ references only such memory addresses that belong to cache lines that have either

- been used before and are re-referenced, while there are at most $\delta^{\max}_{a,\iota_F i} - 1$ other cache lines of a set that have been referenced in the meantime or
- have never been used before (compulsory misses).

However, although there might be entries $H^{sd}_{a,\iota_F i}(\delta) = 0$, especially for values of δ approaching associativity α, this does not necessarily mean that cache lines that are re-referenced have a maximum distance of $\delta^{\max}_{a,\iota_F i}$: A specific value of $\delta^{\max}_{a,\iota_F i}$ only means that the maximum position on the LRU stack of size α that has been referenced is $\delta^{\max}_{a,\iota_F i}$. If the LRU stack however would have a size greater than α, $\delta^{\max}_{a,\iota_F i}$ might also have a value greater than α, even if there are values of δ with $\delta \leq \alpha$ that result in $H^{\mathrm{sd}}_{a,\iota_F i} = 0$.
With variation 'one, set, inf', I investigate on per-cache-set basis if such values are significant regarding prediction performance. Therefore, I assume LRU stacks ζ^S_s of infinite capacity and calculate $\delta^{\max,\inf,\mathrm{S}}_{a,\iota_F i,s}$ according to algorithm 9.
Given $\delta^{\max,\inf,\mathrm{S}}_{a,\iota_F i,s}$ for each $s \in S$, I determine $\delta^{\max,\inf}_{a,\iota_F i}$ by

$$\delta^{\max,\inf}_{a,\iota_F i} = \sum_{s=1}^{|S|} \delta^{\max,\inf,\mathrm{S}}_{a,\iota_F i,s} \tag{51}$$

and calculate prediction $p_{C_a,\iota_F i}$ according to

$$p_{C_a,\iota_F i} = \sum_{a' \in \{a\} \cup C_a} \delta^{\max,\inf}_{a',\iota_F i}, \tag{52}$$

applying $\delta^{\max,\inf}_{a',\iota_F i}$, $a' \in \{a\} \cup C_a$, as predictor:

Predictor	For all $a' \in$	Size (each a')
$\delta^{\max,\inf}_{a',\iota_F i}$	$\{a\} \cup C_a$	$2 \cdot l$

Algorithm 9 Calculation of $\delta^{\max,\inf,\mathrm{S}}_{a,\iota_{F}i,s}$.

1: # —— Initialization ——
2: **for** $s \leftarrow 1$ to $|S|$ **do**
3: **for** $j \leftarrow 1$ to $|M_{a,\iota_{F}i}|$ **do**
4: $\zeta^{S}_{s}(j) \leftarrow -1$
5: **end for**
6: $\delta^{\max,\inf,S}_{a,\iota_{F}i,s}(s) \leftarrow 0$
7: **end for**
8: # —— Iterate over all memory references ——
9: **for** $j \leftarrow 1$ to $|M_{a,\iota_{F}i}|$ **do**
10: $s \leftarrow \varsigma(m_{a,\iota_{F}i,j})$
11: $\delta \leftarrow \zeta^{S}_{s}\{\kappa(m_{a,\iota_{F}i,j})\}$
12: **if** $\delta \leq |M_{a,\iota_{F}i}|$ **then**
13: # —— If the address has ever been referenced before ——
14: **if** $\delta > \delta^{\max,\inf,\mathrm{S}}_{a,\iota_{F}i,s}$ **then**
15: $\delta^{\max,\inf,\mathrm{S}}_{a,\iota_{F}i,s} \leftarrow \delta$
16: **end if**
17: **else**
18: # —— Compulsory miss ——
19: $\delta \leftarrow \delta - 1$
20: **end if**
21: # —— Adjust LRU stack ——
22: **while** $\delta > 1$ **do**
23: $\zeta^{S}_{s}(\delta) \leftarrow \zeta^{S}_{s}(\delta - 1)$
24: $\delta \leftarrow \delta - 1$
25: **end while**
26: $\zeta^{S}_{s}(1) \leftarrow \kappa(m_{a,\iota_{F}i,j})$
27: **end for**

Variation 'set, mask'

With variation 'set, mask', I apply the same concept as for variation 'one, set' but enhanced by masking out cache sets s with $\sum_{a' \in \{a\} \cup C_a} \delta^{\max,S}_{a',\iota_F i,s} \leq \alpha$ according to

$$p_{C_a,\iota_F i} = \sum_{s=1}^{|S|} \Big(\xi^S_{C_a,\iota_F i,s} \cdot \sum_{a' \in \{a\} \cup C_a} \delta^{\max,S}_{a',\iota_F i,s} \Big), \tag{53}$$

where $\xi^S_{C_a,\iota_F i,s}$ is calculated as presented in equation 10. With masking, only those cache sets that are able to introduce contention misses even from stand-alone cache *hits* are incorporated in the calculation of $p_{C_a,\iota_F i}$. This might be a significant improvement, *if* cache contention is primarily determined by stack distance histogram entries $1 \ldots \alpha$, as it is assumed by nearly all state-of-the art prediction methods (e.g. FOA, SDC, ...) and the 'DMax' method (not variations '..., inf, ...') in particular. More precisely, masking will improve prediction accuracy, *if* the displacement of elements from the LRU stack is primarily caused by stand-alone cache hits of co-scheduled applications. In this case, masking avoids that cache sets with values $\sum_{a' \in \{a\} \cup C_a} \delta^{\max,S}_{a',\iota_F i} \leq \alpha$, but near to α, hide distances of cache sets with $\sum_{a' \in \{a\} \cup C_a} \delta^{\max,S}_{a',\iota_F i} > \alpha$.

As an example, consider a cache of associativity $\alpha = 8$ that has only two cache sets s_1 and s_2; further consider four applications a_1, a_2, a_3 and a_4.

Let $\sum_{a' \in \{a_1,a_2\}} \delta^{\max,S}_{a',\iota_F i,s_1} = 8$, $\sum_{a' \in \{a_1,a_2\}} \delta^{\max,S}_{a',\iota_F i,s_2} = 9$ and $\sum_{a' \in \{a_3,a_4\}} \delta^{\max,S}_{a',\iota_F i,s_1} = 4$, $\sum_{a' \in \{a_3,a_4\}} \delta^{\max,S}_{a',\iota_F i,s_2} = 12$. Applying equation 49 results in a predictor of $8 + 9 = 17$ for combination $\{a_1, a_2\}$ and a predictor of $4 + 12 = 16$ for combination $\{a_3, a_4\}$.

If cache contention primarily originates from memory references that are stand-alone cache hits, then combination $\{a_1, a_2\}$ will probably suffer from cache misses to a much lower extent than combination $\{a_3, a_4\}$, although the predictors indicate the opposite. Applying masking however as it is presented in equation 53, cache contention in combination $\{a_1, a_2\}$ is predicted by a value of $0 + 9 = 9$ and contention in combination $\{a_3, a_4\}$ is predicted by a value of $0 + 12$, which will probably be the better estimate, if cache contention primarily originates from stand-alone cache hits that replace other stand-alone cache hits.

As the calculation of $\xi^S_{C_a,\iota_F i,s}$ requires knowledge of the candidate co-schedules and can therefore not be calculated in advance, the per-set-version of $\delta^{\max}_{a,\iota_F i}$, i.e. $\delta^{\max,\mathrm{S}}_{a,\iota_F i,s}$, $s \in S$, has to be applied as predictor.

Predictor	For all $s \in S$ and $a' \in$	Size (each a')
$\delta^{max,S}_{a',\iota_{Fi},s}$	$\{a\} \cup C_a$	$\lvert S\rvert \cdot l$

Variation 'one, set, acc'

Applying $\delta^{\max}_{a',\iota_{Fi}}$ with $a' \in \{a\} \cup C_a$ as the only predictor as it is done in variations 'one' and 'one, set' yields identical values of $p_{C_a,\iota_{Fi}}$ for identical $\delta^{\max}_{a',\iota_{Fi}}$. However, $\delta^{\max}_{a',\iota_{Fi}}$ does not account for the number of references with distance $\delta^{\max}_{a',\iota_{Fi}}$, although it seems very likely that applications that provide a high number of references with $\delta^{\max}_{a',\iota_{Fi}}$ do account more for cache contention than applications with identical $\delta^{\max}_{a',\iota_{Fi}}$, but much less references with distance $\delta^{\max}_{a',\iota_{Fi}}$. Therefore, in variation 'one, set, acc', I extend variation 'one' and 'one, set' to additionally incorporate the number of references with distance $\delta^{\max}_{a',\iota_{Fi}}$, or rather $\delta^{\max,S}_{a,\iota_{Fi},s}$ (cf. figure 14).

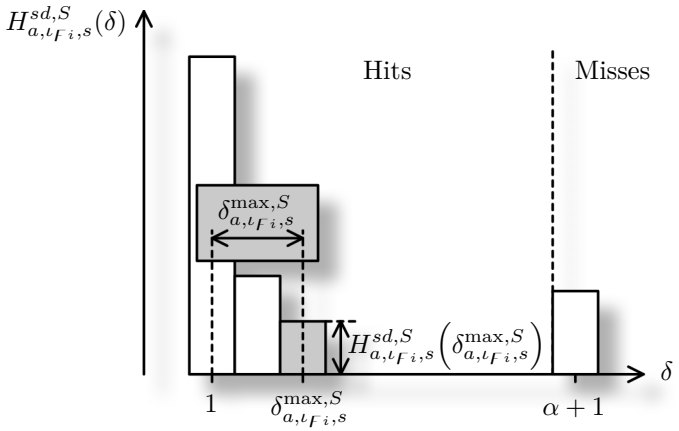


Figure 14: Distance $\delta^{max,S}_{a',\iota_{Fi},s}$ and the number of references $H^{sd,S}_{a',\iota_{Fi},s}(\delta^{max,S}_{a',\iota_{Fi},s})$ with that distance.

I calculate predictor $\delta^{\max,\text{acc}}_{a,\iota_{Fi}}$ by

$$\delta^{\max,\text{acc}}_{a,\iota_{Fi}} = \sum_{s=1}^{|S|} \delta^{\max,S}_{a,\iota_{Fi},s} \cdot H^{sd,S}_{a,\iota_{Fi},s}(\delta^{\max,S}_{a,\iota_{Fi},s}). \tag{54}$$

Note that I define $H^{sd,S}_{a,\iota_{Fi},s}(0) := 0$ in order to account for $\delta^{\max,S}_{a,\iota_{Fi},s} = 0$. Given $\delta^{\max,\text{acc}}_{a,\iota_{Fi}}$, I calculate $p_{C_a,\iota_{Fi}}$ according to

$$p_{C_a,\iota_{Fi}} = \sum_{a' \in \{a\} \cup C_a} \delta^{\max,\text{acc}}_{a',\iota_{Fi}}. \tag{55}$$

I apply the following predictors:

Predictor	For all $a' \in$	Size (each a')
$\delta^{\max,\mathrm{acc}}_{a',\iota_F i}$	$\{a\} \cup C_a$	l

Variation 'set, acc, mask'

With variation 'set, acc, mask', I extend variation 'one, acc' to include masking as it has been described in variation 'set, mask'. I define predictors

$$\delta^{\max,\mathrm{acc},S}_{a,\iota_F i,s} = \delta^{\max,S}_{a,\iota_F i,s} \cdot H^{sd,S}_{a,\iota_F i,s}(\delta^{\max,S}_{a,\iota_F i,s}) \tag{56}$$

and calculate prediction $p_{C_a,\iota_F i}$ to

$$p_{C_a,\iota_F i} = \sum_{s=1}^{|S|} \left(\xi^{S}_{C_a,\iota_F i,s} \cdot \sum_{a' \in \{a\} \cup C_a} \delta^{\max,\mathrm{acc},S}_{a',\iota_F i,s} \right), \tag{57}$$

applying predictors as follows.

Predictor	For all $s \in S$ and $a' \in$	Size (each a')
$\delta^{\max,S}_{a',\iota_F i,s}$	$\{a\} \cup C_a$	$\lvert S\rvert \cdot l$
$\delta^{\max,\mathrm{acc},S}_{a',\iota_F i,s}$	$\{a\} \cup C_a$	$\lvert S\rvert \cdot l$

Variation 'set, exp, acc, mask'

As stack distance histograms are often highly concentrated, i.e. $H^{sd}_{a,\iota_{Fi}}(\delta) \gg H^{sd}_{a,\iota_{Fi}}(\delta+1)$ for low values of δ and $H^{sd}_{a,\iota_{Fi}}(\delta) = 0$ for higher values of δ up to $\delta = \alpha$, variation 'set, acc, mask' often achieves much higher predictions for small values of $\delta^{\max,S}_{a,\iota_{Fi},s}$ than for higher values of $\delta^{\max,S}_{a,\iota_{Fi},s}$. However, higher values of $\delta^{\max,S}_{a,\iota_{Fi},s}$ will probably have a much more effect on cache contention. To properly weight high valued $\delta^{\max,S}_{a,\iota_{Fi},s}$, I account for $\delta^{\max,S}_{a,\iota_{Fi},s}$ exponentially by

$$\delta^{\max,\mathrm{acc},\exp,S}_{a,\iota_{Fi},s} = H^{sd,S}_{a,\iota_{Fi},s}\left(\delta^{\max,S}_{a,\iota_{Fi},s}\right) \cdot \beta^{\delta^{\max,S}_{a,\iota_{Fi},s}-1}, \tag{58}$$

applying base $\beta = 10$. The value of β has been chosen without any further investigation. Given $\delta^{\max,\mathrm{acc},\exp,S}_{a,\iota_{Fi},s}$, I calculate prediction $p_{C_a,\iota_{Fi}}$ by

$$p_{C_a,\iota_{Fi}} = \sum_{s=1}^{|S|}\left(\xi^{S}_{C_a,\iota_{Fi},s} \cdot \sum_{a'\in\{a\}\cup C_a} \delta^{\max,\mathrm{acc},\exp,S}_{a,\iota_{Fi},s}\right). \tag{59}$$

I apply the following predictors for my calculation:

Predictor	For all $s \in S$ and $a' \in$	Size (each a')
$\delta^{max,S}_{a',\iota_{Fi},s}$	$\{a\} \cup C_a$	$\lvert S\rvert \cdot l$
$\delta^{\max,\mathrm{acc},\exp,S}_{a,\iota_{Fi},s}$	$\{a\} \cup C_a$	$\lvert S\rvert \cdot 2 \cdot l$

2.11 The Diff method

Besides $\delta^{\max}_{a,\iota_{Fi}}$ and $\delta^{\max,S}_{a,\iota_{Fi},s}$, I also investigate if the *number of different keys that map to the same cache set* is an appropriate measure to predict cache contention.

Variation 'one'

To determine the number of different keys $\varkappa^{S}_{a,\iota_{Fi},s}$ of application a, interval ι_{Fi} that map (at least *once*) to cache set s, I apply a stack ζ^{S}_{s} of capacity $|M_{a,\iota_{Fi}}|$ for each cache set to track the keys that have already been used, as it is depicted in figure 15 a). See algorithm 10 for an in-depth description of how to calculate $\varkappa^{S}_{a,\iota_{Fi},s}$.

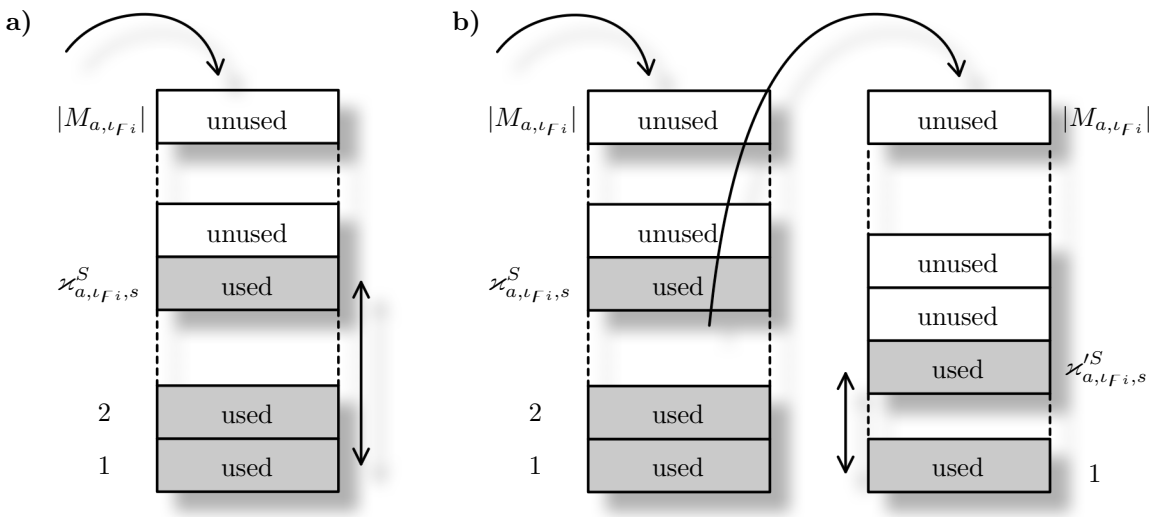


Figure 15: Applying stacks to determine the number of different keys that map to the same cache set. In a) a single stack is used for each cache set; b) follows a two-stack-approach to mask out cache accesses that occur only once (Diff variation 'one, two').

Given $\varkappa^{S}_{a,\iota_{Fi},s}$, I calculate $\varkappa_{a,\iota_{Fi}}$ by $\varkappa_{a,\iota_{Fi}} = \sum_{s=1}^{|S|} \varkappa^{S}_{a,\iota_{Fi},s}$ (cf. algorithm 10). Given $\varkappa_{a,\iota_{Fi}}$, I calculate predictor $p_{C_a,\iota_{Fi}}$ for variation 'one' according to

$$p_{C_a,\iota_{Fi}} = \sum_{a' \in \{a\} \cup C_a} \varkappa_{a',\iota_{Fi}}, \tag{60}$$

applying predictor $\varkappa_{a',\iota_{Fi}}$ for each application $a' \in \{a\} \cup C_a$:

Predictor	For all $a' \in$	Size (each a')
$\varkappa_{a',\iota_{Fi}}$	$\{a\} \cup C_a$	l

Algorithm 10 Calculating $\varkappa^S_{a,\iota_{F\,i},s}$, $\varkappa_{a,\iota_{F\,i}}$ and $\varkappa'^S_{a,\iota_{F\,i},s}$.

1: # —— Init ——
2: **for** $s \leftarrow 1$ **to** $|S|$ **do**
3: $\quad \varkappa^S_{a,\iota_{F\,i},s} \leftarrow 0$
4: $\quad$ **for** $\delta = 1$ **to** $|M_{a,\iota_{F\,i}}|$ **do**
5: $\quad\quad \zeta^S_s(\delta) \leftarrow -1$
6: $\quad\quad \zeta'^S_s(\delta) \leftarrow -1$
7: $\quad$ **end for**
8: **end for**

9: # —— Iterate over all memory references ——
10: **for** $j \leftarrow 1$ **to** $|M_{a,\iota_{F\,i}}|$ **do**
11: $\quad s \leftarrow \varsigma(m_{a,\iota_{F\,i},j})$
12: $\quad \delta \leftarrow \zeta^S_s\{\kappa(m_{a,\iota_{F\,i},j})\}$
13: $\quad$ **if** $\delta > |M_{a,\iota_{F\,i}}|$ **then**
14: $\quad\quad \zeta^S_s(\zeta^S_s\{-1\}) \leftarrow \kappa(m_{a,\iota_{F\,i},j})$
15: $\quad$ **else**
16: $\quad\quad \delta \leftarrow \zeta'^S_s\{\kappa(m_{a,\iota_{F\,i},j})\}$
17: $\quad\quad$ **if** $\delta > |M_{a,\iota_{F\,i}}|$ **then**
18: $\quad\quad\quad \zeta'^S_s(\zeta'^S_s\{-1\}) \leftarrow \kappa(m_{a,\iota_{F\,i},j})$
19: $\quad\quad$ **end if**
20: $\quad$ **end if**
21: **end for**
22: $\varkappa_{a,\iota_{F\,i}} \leftarrow 0$
23: $\varkappa'_{a,\iota_{F\,i}} \leftarrow 0$
24: **for** $s \leftarrow 1$ to $|S|$ **do**
25: $\quad \varkappa^S_{a,\iota_{F\,i},s} \leftarrow \zeta^S_s\{-1\} - 1$
26: $\quad \varkappa_{a,\iota_{F\,i}} \leftarrow \varkappa_{a,\iota_{F\,i}} + \varkappa^S_{a,\iota_{F\,i},s}$
27: $\quad \varkappa'_{a,\iota_{F\,i},s} \leftarrow \varkappa'_{a,\iota_{F\,i},s} + \zeta'^S_s\{-1\} - 1$
28: **end for**

Variation 'set, mask'

To prevent cache sets with $\sum_{a' \in \{a\} \cup C_a} \delta^{\max,S}_{a,\iota_F i,s} \leq \alpha$ to contribute to $p_{C_a,\iota_F i}$, I present variation 'set, mask' that combines $\varkappa^S_{a,\iota_F i,s}$ with masking according to

$$p_{C_a,\iota_F i} = \sum_{s=1}^{|S|} \xi^S_{C_a,\iota_F i,s} \cdot \sum_{a' \in \{a\} \cup C_a} \varkappa^S_{a',\iota_F i,s}, \tag{61}$$

where $\xi^S_{C_a,\iota_F i,s}$ is calculated as presented in equation 10.
I apply the following predictors for each application $a' \in \{a\} \cup C_a$:

Predictor	For all $s \in S$ and $a' \in$	Size (each a')
$\varkappa^S_{a',\iota_F i,s}$	$\{a\} \cup C_a$	$\lvert S\rvert \cdot l$
$\delta^{\max,S}_{a',\iota_F i,s}$	$\{a\} \cup C_a$	$\lvert S\rvert \cdot l$

Variation 'one, miss rate'

With variation 'one, miss rate', I investigate if a combination of both the number of different keys that map to the same cache set and the miss rate would achieve better prediction results than either of the methods alone. I define predictor $\chi^{\mu}_{a,\iota_F i}$ as follows

$$\chi^{\mu}_{a,\iota_F i} = \varkappa_{a',\iota_F i} \cdot \frac{H^{sd}_{a',\iota_F i}(\alpha+1)}{\sum_{\delta=1}^{\alpha+1} H^{sd}_{a',\iota_F i}(\delta)} \tag{62}$$

and calculate prediction $p_{C_a,\iota_F i}$ by

$$p_{C_a,\iota_F i} = \sum_{a' \in \{a\} \cup C_a} \chi^{\mu}_{a,\iota_F i}. \tag{63}$$

I apply the following predictors.

Predictor	For all $a' \in$	Size (each a')
$\chi^{\mu}_{a,\iota_F i}$	$\{a\} \cup C_a$	l

Variation 'one, set, acc'

With variation 'one, set, acc', I combine the number of keys that map to the same cache set and *cache activity*, as it has similarly been proposed in the *Activity Vector* method [Settle et al., 2004]. As a measure of activity, I apply cache access frequency and calculate predictor $\varkappa^{\text{acc}}_{a,\iota_{F}i}$ by

$$\varkappa^{\text{acc}}_{a,\iota_{F}i} = \sum_{s=1}^{|S|}\left(\varkappa^{S}_{a,\iota_{F}i,s} \cdot \sum_{\delta=1}^{\alpha+1} H^{sd,S}_{a,\iota_{F}i,s}(\delta)\right) \tag{64}$$

and prediction $p_{C_a,\iota_{F}i}$ by

$$p_{C_a,\iota_{F}i} = \sum_{a'\in\{a\}\cup C_a} \varkappa^{\text{acc}}_{a,\iota_{F}i}. \tag{65}$$

Applied predictor for each $a' \in \{a\} \cup C_a$:

Predictor	For all $a' \in$	Size (each a')
$\varkappa^{\text{acc}}_{a',\iota_{F}i}$	$\{a\} \cup C_a$	l

Variation 'one, two'

With variation 'one, two', I calculate the number of different keys that map to the same cache set by applying a two-step approach; in doing so, I account for those references only that are not compulsory misses. Figure 15 b) illustrates this concept: I apply two stacks of capacity $|M_{a,\iota_{F}i}|$ for each cache set. The first time a cache line is referenced, its address key is pushed to the first stack. When a cache line is referenced that is present on the first, but not on the second stack, I push the corresponding key to the second stack. This way, the second stack contains only those references to cache lines that are accessed at least twice. I refer to this number of different keys as $\varkappa'^{S}_{a,\iota_{F}i,s}$; see algorithm 10 for a formal description of the calculation of $\varkappa'^{S}_{a,\iota_{F}i,s}$. Given $\varkappa'^{S}_{a,\iota_{F}i,s}$ for each $s \in S$, I calculate $\varkappa'_{a,\iota_{F}i}$ by $\varkappa'_{a,\iota_{F}i} = \sum_{s=1}^{|S|} \varkappa'^{S}_{a,\iota_{F}i,s}$. Given $\varkappa'_{a,\iota_{F}i}$, I calculate predictor $p_{C_a,\iota_{F}i}$ by

$$p_{C_a,\iota_{F}i} = \sum_{a'\in\{a\}\cup C_a} \varkappa'_{a',\iota_{F}i}. \tag{66}$$

Predictor	For all $a' \in$	Size (each a')
$\varkappa'_{a',\iota_{F}i}$	$\{a\} \cup C_a$	l

2.12 The DMiss method

With the DMiss method, I introduce a new method that enhances the Misses method. One way to improve cache miss based methods is to exploit existing diversities in stand-alone cache misses. One such diversity is the *number of references between cache misses* I call *distance*; therefore the name *DMiss*, which means *distance between misses*. Figure 16 a) shows equally distributed memory references of applications $\{a\} \cup C_a$ that are stand-alone cache misses, i.e. contribute to $H^{sd}_{a',\iota_{F\,i}}(\alpha+1)$, $a' \in \{a\} \cup C_a$.

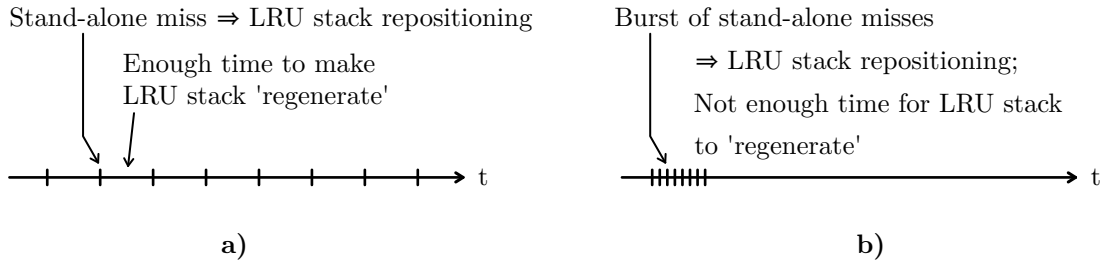


Figure 16: LRU stack regeneration as function of stand-alone cache miss distribution.

With each such reference, elements $\delta = 1$ to $\delta = \alpha - 1$ on the LRU stack get repositioned by one towards LRU position $\delta = \alpha$. If $\sum_{a' \in \{a\} \cup C_a} \delta^{\max,S}_{a',\iota_{F\,i},s} < \alpha$, which is likely for concentrated stack distance histograms, and if the distance between two references that are stand-alone misses is large enough, then the LRU stack might 'regenerate': If the cache line that has been fetched and pushed to LRU stack position $\delta = 1$ is not referenced again, the correpsonding key on the LRU stack is moved towards the LRU position ($\delta = \alpha$) and the LRU stack might be in the same state just as before the stand-alone miss reference.
In figure 16 b) however, a burst of misses occurs and there will be no time for the LRU stack to regenerate, which might introduce cache contention misses if displaced cache lines are referenced again in the future.
Regarding this scenario, it seems to be worth investigating if the distance between two consecutive misses to the same cache set is a good predictor for cache contention. Relating this method to the *Misses* method, the number of misses can be interpreted as a distance measure: On average, a higher number of misses implies a lower distance between misses; a lower number of misses implies a higher distance between misses.

I calculate distances $\delta^{miss}_{a,\iota_{Fi}}$ and per-set distances $\delta^{miss,S}_{a,\iota_{Fi},s}$ for an application a as shown in algorithm 11. Note that lower distances between misses result in higher values of $\delta^{miss}_{a,\iota_{Fi}}$ and $\delta^{miss,S}_{a,\iota_{Fi},s}$ respectively, as shown in the algorithm.

Variation 'one'

Given $\delta^{miss}_{a,\iota_{Fi}}$ as determined by algorithm 11, I calculate prediction $p_{C_a,\iota_{Fi}}$ by

$$p_{C_a,\iota_{Fi}} = \sum_{a' \in \{a\} \cup C_a} \delta^{miss}_{a,\iota_{Fi}}. \tag{67}$$

I apply the following predictors:

Predictor	For all $a' \in$	Size (each a')
$\delta^{miss}_{a',\iota_{Fi}}$	$\{a\} \cup C_a$	l

Variation 'one, sens38'

With variation 'one, sens38', I investigate if $\delta^{miss}_{c_a,\iota_{Fi}}$, $c_a \in C_a$ achieves good prediction results when applied as *intensity* measure, as it is done in the Pain method. Given *sensitivity* $\chi^{sens38}_{a,\iota_{Fi}}$ as calculated in equation 32, I calculate prediction $p_{C_a,\iota_{Fi}}$ as follows.

$$p_{C_a,\iota_{Fi}} = \chi^{sens38}_{a,\iota_{Fi}} \cdot \sum_{c_a \in C_a} \delta^{miss}_{c_a,\iota_{Fi}} \tag{68}$$

I apply the following predictors:

Predictor	For all $a' \in$	Size (each a')
$\chi^{sens38}_{a',\iota_{Fi}}$	$\{a\}$	l
$\delta^{miss}_{a',\iota_{Fi}}$	C_a	l

Variation ‘set, sens38’

With variation ‘set, sens38’, I investigate if a *per-cache-set* calculation of variation ‘one, sens38’ would achieve better results. I apply sensitivity $\chi^{sens38,S}_{a,\iota_{Fi},s}$ as defined in equation 40 and $\delta^{miss,S}_{a',\iota_{Fi},s}$ as determined by algorithm 11 and calculate prediction $p_{C_a,\iota_{Fi}}$ by

$$p_{C_a,\iota_{Fi}} = \sum_{s=1}^{|S|} \chi^{sens38,S}_{a,\iota_{Fi},s} \cdot \sum_{c_a \in C_a} \delta^{miss,S}_{c_a,\iota_{Fi},s}, \tag{69}$$

applying predictors as follows.

Predictor	For all $s \in S$ and $a' \in$	Size (each a')
$\chi^{sens38,S}_{a',\iota_{Fi},s}$	$\{a\}$	$\lvert S\rvert \cdot l$
$\delta^{miss,S}_{a',\iota_{Fi},s}$	C_a	$\lvert S\rvert \cdot l$

Algorithm 11 Calculation of $\delta^{miss}_{a,\iota_{Fi}}$ and $\delta^{miss,S}_{a,\iota_{Fi},s}$.

1: # —— Init ——
2: **for** $s \leftarrow 1$ to $|S|$ **do**
3: $j^S_s \leftarrow 0$
4: $j'^S_s \leftarrow 0$
5: **for** $j \leftarrow 1$ to α **do**
6: $\zeta^S_s(j) \leftarrow -1$
7: **end for**
8: **end for**

9: # —— Iterate over all memory references ——
10: **for** $j \leftarrow 1$ to $|M_{a,\iota_{Fi}}|$ **do**
11: $s \leftarrow \varsigma(m_{a,\iota_{Fi},j})$
12: $\delta \leftarrow \zeta^S_s\{\kappa(m_{a,\iota_{Fi},j})\}$
13: $j^S_s \leftarrow j^S_s + 1$

14: # —— If the reference is a miss ——
15: **if** $\delta > \alpha$ **then**
16: **if** $j_s'^S \neq 0$ **then**
17: $\delta' \leftarrow j_s^S - j_s'^S$
18: **if** $\delta' \leq \alpha$ **then**
19: $\delta_{a,\iota_{F}i}^{miss} \leftarrow \delta_{a,\iota_{F}i}^{miss} + (\alpha + 1 - \delta')$
20: $\delta_{a,\iota_{F}i,s}^{miss,S} \leftarrow \delta_{a,\iota_{F}i,s}^{miss,S} + (\alpha + 1 - \delta')$
21: **else**
22: $\delta_{a,\iota_{F}i}^{miss} \leftarrow \delta_{a,\iota_{F}i}^{miss} + \alpha^{-(\lfloor log_\alpha(\delta') \rfloor + 1)}$
23: $\delta_{a,\iota_{F}i,s}^{miss,S} \leftarrow \delta_{a,\iota_{F}i,s}^{miss,S} + \alpha^{-(\lfloor log_\alpha(\delta') \rfloor + 1)}$
24: **end if**
25: **end if**
26: $j_s'^S \leftarrow j_s^S$
27: $\delta \leftarrow \alpha$
28: **end if**
29: # —— Adjust the LRU stack ——
30: **while** $\delta > 1$ **do**
31: $\zeta_s^S(\delta) \leftarrow \zeta_s^S(\delta - 1)$
32: $\delta \leftarrow \delta - 1$
33: **end while**
34: $\zeta_s^S(1) \leftarrow \kappa(m_{a,\iota_{F}i,j})$
35: **end for**

3 Evaluating the Prediction of Cache Contention

In this chapter, I evaluate the techniques presented in chapter 2 by

- prediction *accuracy*; as evaluation measure I apply
  - *NMRD* (normalized mean ranking difference), which measures the ability of a prediction technique to rank a set of candidate co-schedules $\mathbf{C}_a = \{C_{a,1}, C_{a,2}, \dots\}$ by the amount of cache contention they introduce to an application a;
  - *MP* (mean penalty), which enhances NMRD by not evaluating ranking positions, but the penalty in time introduced from imperfectly predicted ranking positions;
  - *PPBAB* (penalty predicted best vs. actual best), which measures the amount of time the penalty of the co-schedule, which is *predicted* to be the best co-schedule, differs from the *actual* best co-schedule;
  - *PPBRS* (penalty predicted best vs. random selection), which measures the amount of time the penalty of the co-schedule, which is *predicted* to be the best co-schedule, differs from the *expected* penalty when the co-schedule is chosen randomly;
- *time* to perform a prediction, and a
- *gain vs. cost* analysis, which compares the amount of time that can be gained from proper co-scheduling (PPBRS) to the amount of time necessary to perform a prediction.

3.1 Evaluation framework

For my evaluation, I apply the framework presented in figure 17. As the figure suggests, the evaluation is partitioned into 5 steps. In step (A), I extract memory references of several applications and store them to tracefiles. In step (B), I apply those tracefiles to create predictors for the various methods described in chapter 2. In step (C), I apply the predictors to each prediction method described in chapter 2; I predict the penalty that candidate co-schedules $C_{a,j} \in \mathbf{C}_a = \{C_{a,1}, C_{a_2}, \dots\}$ introduce to a given application a and measure the time spent to perform the prediction. Then, I sort the predictions to determine predicted co-schedule rankings.

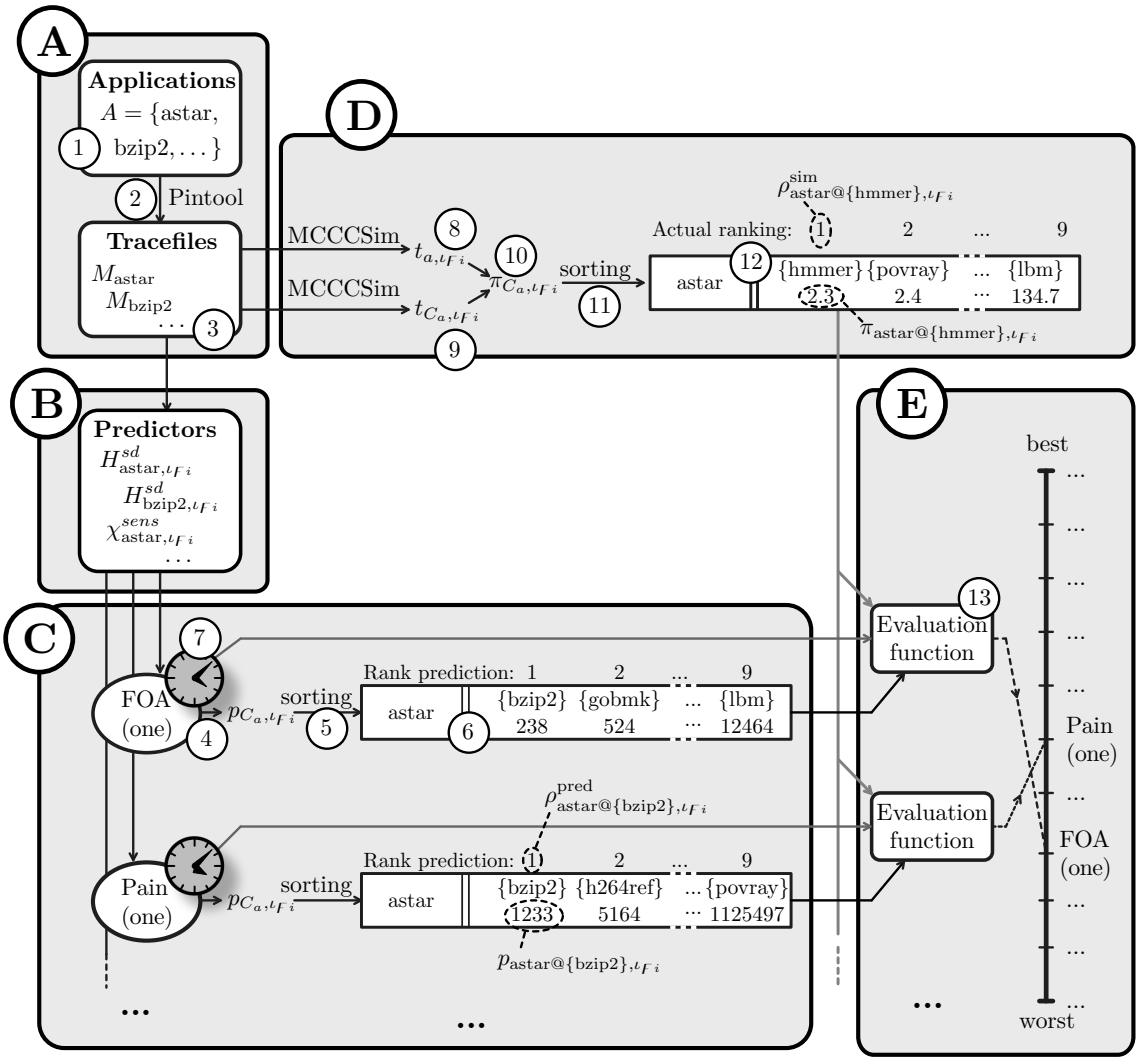


Figure 17: Evaluation framework I apply to evaluate cache contention prediction techniques.

In step (D), I calculate actual co-schedule penalties and actual rankings in order to generate a ground truth reference. In step (E), I apply several evaluation functions (NMRD, ...) to merge predicted rankings, actual rankings, actual penalties and prediction time to a single value; I use this value to compare various methods on a one-dimensional scale.
The remainder of this section describes steps (A) to (D) in more detail. Step (E), the evaluation by means of several evaluation functions, is accomplished in sections 3.2 to 3.5.

(A) Memory Reference Extraction with Pin

Let $A = \{\text{astar, bzip2, gcc, gobmk, h264ref, hmmer, lbm, mcf, milc, povray}\}$ be the set of SPEC 2006 test benchmarks I apply for my evaluation (see (1) in figure 17).
In order to extract memory references (2) from A, I implemented a pintool for the *Pin* toolkit [Luk et al., 2005], as it has similarly been done in [Huffmire and Sherwood, 2006] and [Zhuravlev et al., 2010]. The implemented pintool is a small program written in C/C++ applying the Pin API (application programming interface) and works as follows: Invoking the pintool with an application $a \in A$, the pintool interrupts application a whenever a performs a memory access; then, the pintool stores the corresponding address to a buffer. Whenever the pintool has buffered memory references of 2^{20} instructions, the pintool writes the buffer to a tracefile (3) and resets its buffer. Then, the pintool restarts to extract memory references of the next 2^{20} instructions and writes them to the same tracefile. All in all, this pocedure is performed 512 times for each $a \in A$, so memory references from $512 \cdot 2^{20}$ instructions are stored in each tracefile. Tuple $M_a = (m_{a,1}, m_{a,2}, \dots)$ refers to the memory references stored in the tracefile of application a.
When predicting cache contention in order to bring those applications together that best fit to one another, a major concern is the interval width $\mathcal{F}$ (measured in number of instructions) the prediction is performed on. If $\mathcal{F}$ is too small, then it might take more time to predict cache contention than a good application match will compensate for. If $\mathcal{F}$ gets too large, then applications are likely to change their so-called phase within an interval [Sherwood et al., 2003], i.e. they might show a completely different behavior, which might degrade prediction performance. In order to evaluate cache contention prediction techniques for several interval widths, I perform my evaluation on intervals of $\mathcal{F} \in \boldsymbol{\mathcal{F}} = \{2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}, 2^{25}, 2^{26}, 2^{27}, 2^{28}, 2^{29}\}$ instructions.

Therefore, I partition all M_a, $a \in A$, into memory reference intervals of $\mathcal{F}$ instructions each, i.e. $M_a \mapsto M_{a,\iota_\mathcal{F}} = (M_{a,\iota_\mathcal{F}1}, \dots M_{a,\iota_\mathcal{F}|M_{a,\iota_\mathcal{F}}|})$ and perform my evaluation on each $M_{a,\iota_\mathcal{F}i} \in M_{a,\iota_\mathcal{F}}$. Note that tuple $M_{a,\iota_\mathcal{F}i} = (m_{a,\iota_\mathcal{F}i,1}, \dots m_{a,\iota_\mathcal{F}i,|M_{a,\iota_\mathcal{F}i}|})$ refers to the memory addresses that are referenced when executing instructions $i \cdot \mathcal{F} \dots (i+1) \cdot \mathcal{F} - 1$. The amount of memory references of application a in interval $\iota_{\mathcal{F}i}$ is denoted by $|M_{a,\iota_\mathcal{F}i}|$.

(B) Predictor Calculation

I calculate predictors according to their definition in chapter 2. Note that calculation is performed for each (single) application $a \in A$, for all interval sizes $\mathcal{F} \in \boldsymbol{\mathcal{F}}$, and for each interval $\iota_{\mathcal{F}i} \in \iota_\mathcal{F}$. As predictors are calculated from memory references of solo applications only, they are identical for all cases of parallelism $\psi \in \boldsymbol{\psi} = \{2, 4, 8\}$.

For my evaluations, I assume predictors to be calculated at compile time; therefore, they do not account for the time required to calculate a prediction. The *size* of the predictors however might be of interest and can be extracted from the description in chapter 2.

(C) Calculation of Prediction Rankings

Let $\mathfrak{C}(A, \psi)$ be the operation that returns all possible combinations of ψ elements of set A, $\psi \leq |A|$, i.e. $\mathfrak{C}(A, \psi) = \Big\{\{a_1, a_2, \dots a_\psi\} \mid a_i \in A \backslash \{a_1, a_2, \dots a_{i-1}\}\Big\}$. On a processor architecture where ψ applications are executed in parallel and share the same cache, $\mathbf{C}_a^\psi = \{C_{a,1}^\psi, C_{a,2}^\psi, \dots\} = \mathfrak{C}(A \backslash \{a\}, \psi - 1)$ is the set of all possible candidate co-schedules for an application a that can be generated from a set of applications $A \backslash \{a\}$. The size of that set, i.e. the amount of all possible candidate co-schedules of application a, is $|\mathbf{C}_a^\psi|$. Note that I refer to this set as $\mathbf{C}_a^\psi = \{C_{a,1}^\psi, C_{a,2}^\psi, \dots\}$ and $\mathbf{C}_a = \{C_{a,1}, C_{a,2}, \dots\}$ interchangeably; I also abbreviate $C_{a,j}$ by C_a.

Prediction rankings are calculated as follows: For each prediction technique presented in chapter 2, for each $\psi \in \boldsymbol{\psi}$, for each $\mathcal{F} \in \boldsymbol{\mathcal{F}}$, each $\iota_{\mathcal{F}i} \in \iota_\mathcal{F}$ and each $a \in A$, I

- (4) *calculate prediction* $p_{C_a,\iota_\mathcal{F}i}$ for each $C_a \in \mathbf{C}_a^\psi$,
- (5) *sort* the resulting predictions $p_{C_a,\iota_\mathcal{F}i}$ by value and, in case of equal values, by a unique hash generated from C_a in order to ensure unambiguitiy, and
- (6) *determine ranking position* $\rho^{\text{pred}}_{C_a,\iota_\mathcal{F}i}$ for each candidate co-schedule $C_a \in \mathbf{C}_a^\psi$.

Sorting and ranking is performed as follows: Given predictions $\{p_{C_{a,1},\iota_{\mathcal{F}}i}, p_{C_{a,2},\iota_{\mathcal{F}}i}, \dots\}$ and a hash function $h(C_{a,j})$ that returns a unique hash value $h(C_{a,j}) \in \{1, 2, \dots |\mathbf{C}_a^\psi|\}$ for all $C_{a,j} \in \mathbf{C}_a^\psi$, i.e. $\forall_{j,k \in \{1,\dots|\mathbf{C}_a^\psi|\}} : j \neq k \Rightarrow h(C_{a,j}) \neq h(C_{a,k})$, I determine a unique ranking position $\rho^{\text{pred}}_{C_{a,j},\iota_{\mathcal{F}}i} \in \{1, 2 \dots |\mathbf{C}_a^\psi|\}$ of co-schedule $C_{a,j}$ in interval $\iota_{\mathcal{F}}i$ by

$$\begin{aligned} &\forall_{j,k} \in \{1, 2, \dots |\mathbf{C}_a^\psi|\} : \\ &\quad p_{C_{a,j},\iota_{\mathcal{F}}i} < p_{C_{a,k},\iota_{\mathcal{F}}i} \Rightarrow \rho^{\text{pred}}_{C_{a,j},\iota_{\mathcal{F}}i} < \rho^{\text{pred}}_{C_{a,k},\iota_{\mathcal{F}}i} \\ &\quad p_{C_{a,j},\iota_{\mathcal{F}}i} = p_{C_{a,k},\iota_{\mathcal{F}}i} \wedge h(C_{a,j}) < h(C_{a,k}) \Rightarrow \rho^{\text{pred}}_{C_{a,j},\iota_{\mathcal{F}}i} < \rho^{\text{pred}}_{C_{a,k},\iota_{\mathcal{F}}i}. \end{aligned} \tag{70}$$

See figure 18 for an example prediction ranking with $\psi = 2$ for $a =$ astar and $\mathbf{C}_a^\psi = \{\{\text{bzip2}\}, \{\text{gcc}\}, \{\text{gobmk}\}, \{\text{h264ref}\}, \{\text{hmmer}\}, \{\text{lbm}\}, \{\text{mcf}\}, \{\text{milc}\}, \{\text{povray}\}\}$: Co-schedule {bzip2} is predicted to introduce the least penalty (202) to application astar and therefore yields the best ranking position (1). Co-schedule {povray} is predicted to introduce maximum penalty (1071) to application astar and is therefore assigned the worst ranking position (9).

$\rho^{\text{pred}}_{\text{astar@}\{\text{bzip2}\},\iota_{\mathcal{F}}i} = \rho^{\text{pred}}_{C_{a,j},\iota_{\mathcal{F}}i}$ with $a =$ astar and $C_{a,j} = \{\text{bzip2}\}$; $\psi = 2$

Ranking:	(1)	2	3	4	5	6	7	8	9
astar	{bzip2}	{milc}	{gcc}	{gobmk}	{hmmer}	{h264ref}	{lbm}	{mcf}	{povray}
	202	252	300	623	652	662	808	1054	1071

$p_{\text{astar@}\{\text{bzip2}\},\iota_{\mathcal{F}}i} = p_{C_{a,j},\iota_{\mathcal{F}}i}$ with $a =$ astar and $C_{a,j} = \{\text{bzip2}\}$; $\psi = 2$

Figure 18: Ranking of predictions $p_{C_a^\psi,\iota_{\mathcal{F}}i}$ for $a =$ astar and $\psi = 2$.

When calculating predictions, I measure the time (7) a prediction method requires to calculate predictions $p_{C_a,\iota_{\mathcal{F}}i}$ for all $C_a \in \mathbf{C}_a^\psi$ and for all intervals $\iota_{\mathcal{F}}i \in \iota_{\mathcal{F}}$ by means of *user* time $\tau^{\text{user}}_{C_a}$, *system* time $\tau^{\text{syst}}_{C_a}$, and *elapsed* time $\tau^{\text{elap}}_{C_a}$. Note that $\tau^{\text{user}}_{C_a}$, $\tau^{\text{syst}}_{C_a}$ and $\tau^{\text{elap}}_{C_a}$ hold the time required for the prediction of *all* intervals $\iota_{\mathcal{F}}i \in \iota_{\mathcal{F}}$. To calculate the time *per prediction*, $\tau^{\text{user}}_{C_a}$, $\tau^{\text{syst}}_{C_a}$, and $\tau^{\text{elap}}_{C_a}$ have to be divided by the number of intervals $|\iota_{\mathcal{F}}|$, as prediction is performed on a per-interval basis. For no specific reason, this calculation is not performed in step (C), but within the evaluation function in step (E). With algorithm 12, I shortly outline my implementation of time measurement: I determine $\tau^{\text{elap}}_{C_a}$ by the `gettimeofday(2)` function and $\tau^{\text{user}}_{C_a}$ and $\tau^{\text{syst}}_{C_a}$ by the `getrusage(2)` function integrated in the OSX 10.6.4 operating system by default. Note that I include the time to read the predictors from disk in order to account for different predictor sizes. To ensure equal

Algorithm 12 Measuring time of a prediction cycle.

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

conditions, all prediction methods are implemented single threaded in the same programming language (C++), are compiled applying the same compiler (gcc 4.2.1) and executed on the same otherwise unloaded computer system running OS X 10.6.4 operating system. See the following table for a brief description of the computer system.

As data blocks on hard disk drive platters generally have a random position relative to read/write heads when read/write commands are invoked, seek time heavily varies. In order to keep varitions in disk access time low, the computer system I apply for time measurements features a random access solid state drive of uniform access time.

Computer Model	MacBook Pro
Computer model identifier	MacBookPro5.1
Processor	2.4 GHz Intel Core 2 Duo
L1 data cache	32 kB
L1 instruction cache	32 kB
L2 cache (shared)	3 MB
Main memory	4 GB DDR3
Hard disk drive	256 GB solid state disk
Bus speed	1.07 GHz

(D) Generation of Ground Truth Reference

For each $a \in A$, each $F \in \boldsymbol{F}$, and each $\iota_{Fi} \in \iota_F$, I

- (8) *apply the MCCCSim simulator* [Zwick et al., 2009a] to *calculate* $t_{a,\iota_F i}$, the amount of time application a spends on memory references in interval ι_{Fi} when a is executed stand-alone ($\psi = 1$).

Then, for each $\psi \in \boldsymbol{\psi}$, for each $a \in A$, each $C_a \in \mathbf{C}_a^\psi = \mathfrak{C}(A \backslash \{a\}, \psi - 1)$, each $F \in \boldsymbol{F}$, and each $\iota_{Fi} \in \iota_F$, I

- (9) *apply the MCCCSim simulator to calculate* t_{C_a,ι_F}, the amount of time application a spends on memory references in interval ι_{Fi} when a is co-scheduled with applications $c_{a,j} \in C_a$,
- (10) *calculate* $\pi_{C_a,\iota_F i} = t_{C_a,\iota_F i} - t_{a,\iota_F i}$, the penalty that is introduced to application a in interval ι_{Fi} when a is co-scheduled with C_a, and
- (11) *sort all* $\pi_{C_a,\iota_F i}$ according to their value and, if values are identical, additionally by a unique hash generated from C_a in order to achieve unambiguous rankings, and then
- (12) *determine ranking position* $\rho^{\text{sim}}_{C_a,\iota_F i} \in \{1, 2, \dots |\mathbf{C}_a^\psi|\}$ each candidate co-schedule $C_a \in \mathbf{C}_a^\psi$ achieves.

Similar to equation 70, I determine actual (and unique) ranking positions $\rho^{\text{sim}}_{C_{a,j},\iota_F i} \in \{1, 2, \dots |\mathbf{C}_a^\psi|\}$ for candidate co-schedules $C_{a,j} \in \mathbf{C}_a^\psi$ by

$$\begin{aligned} &\forall_{j,k} \in \{1, 2, \dots |\mathbf{C}_a^\psi|\} : \\ &\quad \pi_{C_{a,j},\iota_F i} < \pi_{C_{a,k},\iota_F i} \Rightarrow \rho^{\text{sim}}_{C_{a,j},\iota_F i} < \rho^{\text{sim}}_{C_{a,k},\iota_F i} \\ &\quad \pi_{C_{a,j},\iota_F i} = \pi_{C_{a,k},\iota_F i} \wedge h(C_{a,j}) < h(C_{a,k}) \Rightarrow \rho^{\text{sim}}_{C_{a,j},\iota_F i} < \rho^{\text{sim}}_{C_{a,k},\iota_F i}. \end{aligned} \tag{71}$$

As an example, figure 19 shows actual ranking positions for $\psi = 2$, $a =$ astar, and $\mathbf{C}_a^\psi = \big\{\{\text{bzip2}\}, \{\text{gcc}\}, \{\text{gobmk}\}, \{\text{h264ref}\}, \{\text{hmmer}\}, \{\text{lbm}\}, \{\text{mcf}\}, \{\text{milc}\}, \{\text{povray}\}\big\}$. Note that in some execution intervals ι_{Fi}, co-scheduling an application a with a set of applications C_a might result in a *lower* memory access time of application a than it is achieved when executing a stand-alone, i.e. $t_{C_a,\iota_F i} < t_{a,\iota_F i}$ and $\pi_{C_a,\iota_F i} < 0$. This statement seems to be a paradox, as any application $c_a \in C_a$ can only *displace* an address key of a from the LRU stack, but *never fetch* any data of a to a cache or a TLB, as application a

$\rho^{\text{sim}}_{\text{astar@\{hmmer\}},\iota_{\mathcal{F}i}} = \rho^{\text{sim}}_{C_{a,j},\iota_{\mathcal{F}i}}$ with $a = \text{astar}$ and $C_{a,j} = \{\text{hmmer}\};\ \psi = 2$

Ranking:	1	2	3	4	5	6	7	8	9
astar	{hmmer}	{povray}	{h264ref}	{gcc}	{bzip2}	{mcf}	{gobmk}	{milc}	{lbm}
	2.3	2.4	2.7	23.0	33.9	65.3	95.2	101.6	134.7

$\pi_{\text{astar@\{hmmer\}},\iota_{\mathcal{F}i}} = \pi_{C_{a,j},\iota_{\mathcal{F}i}}$ with $a = \text{astar}$ and $C_{a,j} = \{\text{hmmer}\};\ \psi = 2$

Figure 19: Ranking of actual penalties $\pi_{C_a,\iota_{\mathcal{F}i}}$ for $a = \text{astar}$ and $\psi = 2$.

does not share any data with any application $c_a \in C_a$ (cf. 'Limitations' in section 1.3). Simulating execution intervals with the MCCCSim simulator [Zwick et al., 2009a], however, I observed the following: There are cases where a co-scheduled application $c_a \in C_a$ displaces an address translation of a from the shared L1 TLB. Although this introduces an L1 TLB miss penalty to a, it also results in a better LRU stack position for this address translation in the L2 TLB. This better LRU stack position, in turn, might result in an L2 TLB hit of an address translation of a at a later time, while the same address translation might result in an L2 TLB miss if application a is executed stand-alone, due to an unfavorable TLB 2 LRU stack position.

Figure 20 shows the MCCCSim setup I apply to calculate $t_{a,\iota_{\mathcal{F}i}}$ and $t_{C_a,\iota_{\mathcal{F}i}}$. Note that the L2 cache is shared among L1 data caches only; L1 instruction caches and TLBs do not have access to the L2 cache in order to improve observability of L2 cache contention and exclude as many side effects as possible from my simulation. Therefore, L1 and L2 data caches also incorporate virtual addresses only and address translation (MMU, memory management unit) is performed on L2 data cache misses only.

Figure 21 shows the parameterization of the MCCCSim simulator.

In order to make my evaluation reproducible, I introduce algorithm 13 that formally specifies the way MCCCSim determines $t_{a,\iota_{\mathcal{F}i}}$ and $t_{C_a,\iota_{\mathcal{F}i}}$ from a given set of memory reference tuples $M_{a,\iota_{\mathcal{F}i}}$ and $M_{C_a,\iota_{\mathcal{F}i}}$, $C_a \in \mathbf{C}^{\psi}_a$. In order to calculate $t_{a,\iota_{\mathcal{F}i}}$, algorithm 13 has to be executed with $A' = \{a\}$, which implies $\psi = 1$; then, variable $t_{a,\iota_{\mathcal{F}i}}$ in algorithm 13 corresponds to the time application a spends on memory references when executing interval $\iota_{\mathcal{F}i}$. To calculate $t_{C_a,\iota_{\mathcal{F}i}}$, however, algorithm 13 has to be executed applying $A' = \{a\} \cup C_a$. Then, variable $t_{a,\iota_{\mathcal{F}i}}$ in algorithm 13 represents execution time under cache sharing for *each* application $\in A'$, depending on the selection of a in lines 38 to 45.

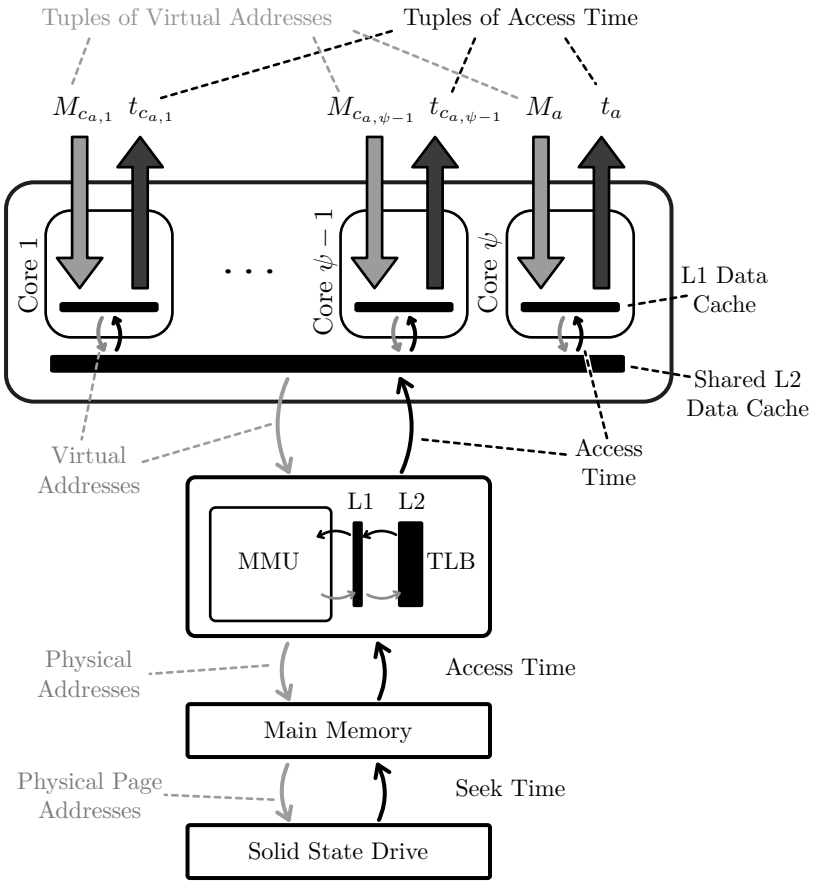


Figure 20: Setup of the MCCCSim simulator.

Parameter	L1 data cache	L2 data cache	L1 TLB	L2 TLB	MEM (Memory)	SSD (Solid State Disk)
Size	32 kB	2 MB	48 entries	512 entries	∞	∞
Line Size	128 Byte	128 Byte	-	-	-	-
Associativity	2	8	-	-	-	-
Hit Time	$t^{\mathrm{L1C}} = 1\,\mathrm{ns}$	$t^{\mathrm{L2C}} = 10\,\mathrm{ns}$	$t^{\mathrm{L1T}} = 1\,\mathrm{ns}$	$t^{\mathrm{L2T}} = 10\,\mathrm{ns}$	-	-
On Miss	L2 data cache access	L1 TLB & MEM access	L2 TLB access	MEM access (page table)	SSD access	-
Access Time	-	-	-	-	$t^{\mathrm{Mem}} = 100\,\mathrm{ns}$	$t^{\mathrm{Disk}} = 0.1\,\mathrm{ms}$
Replacement	LRU	LRU	LRU	LRU	-	-

Figure 21: MCCCSim setup parameters.

In algorithm 13, I apply symbols and notations as follows.

- A' is the set of applications that are executed in parallel on the architecture presented in figure 20. Note that each application $\in A'$ is executed on a separate core and the number of cores equals the number of applications, i.e. $\psi = |A'|$.

- $\zeta^{\mathrm{L1C,S}} = \left\{\zeta_{a_1}^{\mathrm{L1C,S}}, \ldots \zeta_{a_\psi}^{\mathrm{L1C,S}}\right\}$ is a set of sets of stacks where each
- $\zeta_{a_i}^{\mathrm{L1C},S} \in \zeta^{\mathrm{L1C},S}$ is the set of stacks that holds tag RAM information for the processor core executing application $a_i \in A'$.
- $\zeta_{a_i,s}^{\mathrm{L1C},S} \in \zeta_{a_i}^{\mathrm{L1C},S} = \left\{\zeta_{a_i,1}^{\mathrm{L1C},S}, \ldots \zeta_{a_i,|S^{\mathrm{L1C}}|}^{\mathrm{L1C},S}\right\}$ is the stack of capacity α^{L1C} that holds tag RAM information for cache set $s \in \{1, 2, \ldots |S^{\mathrm{L1C}}|\}$ of the processor core that executes application a_i, where
- $\alpha^{\mathrm{L1C}} = 2$ is the associativity of the L1 data caches (cf. figure 21), and
- $|S^{\mathrm{L1C}}| = 32\,\mathrm{k}/(2 \cdot 128) = 128$ is the number of cache sets of each L1 data cache.
- $\zeta^{\mathrm{L2C},S} = \left\{\zeta_1^{\mathrm{L2C},S}, \ldots \zeta_{|S^{\mathrm{L2C}}|}^{\mathrm{L2C},S}\right\}$ is the set of stacks that holds tag RAM information for the shared L2 cache, where each
- $\zeta_s^{\mathrm{L2C},S} \in \zeta^{\mathrm{L2C},S}$ is of capacity α^{L2C}.
- $\alpha^{\mathrm{L2C}} = 8$ is the associativity of the shared L2 cache (cf. figure 21).
- $|S^{\mathrm{L2C}}| = 2\,\mathrm{M}/(8 \cdot 128) = 2048$ is the number of cache sets of the L2 cache.

- ζ^{L1T} is the stack of capacity K^{L1T} that holds L1 TLB tags, where
- K^{L1T} is 48 (cf. figure 21).

- ζ^{L2T} is the stack of capacity K^{L2T} that holds L2 TLB tags, where
- K^{L2T} is 512 (cf. figure 21).

- ζ^{Mem} is the stack of capacity $\psi \cdot 2^{|m|-ld(|\varpi|)}$ that holds the set of unique page addresses that the applications in A' reference, where
- $|m| = 32$ is the word length of a memory address in bits, and

- $|\varpi| = 2^{12} = 4\,\mathrm{k}$ is the page size in byte.
- $\varpi_a(m)$ is the operation that transforms a memory reference m of application a into a *page address*. I assume a byte addressed memory architecture and define $\varpi_a(m) = (m \bmod |\varpi|) + h(a) \cdot 2^{|m|-ld(|\varpi|)}$, where $h(a) \in \{1, 2, \ldots \psi\}$ is the

operation that returns a unique hash for each $a \in A'$. Therefore, $\forall_{a_i,a_j \in A'} : a_i \neq a_j \Rightarrow \nexists_{m_i,m_j} : \varpi_{a_i}(m_i) = \varpi_{a_j}(m_j)$. Generally, $\forall_{a \in A', \lceil ld(\psi) \rceil < ld(|\varpi|)} : \varpi_a(m) \geq 0$.

- $\varsigma^{\mathrm{L1C}}(m)$ is the operation that extracts the *set* address for the L1 caches from a memory reference m. Given way size $|w| = 32\,\mathrm{k}/2 = 16\,\mathrm{k}$ byte and line size $|\lambda| = 128$ byte (cf. figure 21), then $\varsigma^{\mathrm{L1C}}(m) = ((m/|\lambda|) \bmod (|w|/|\lambda|)) + 1 = ((m/128) \bmod 128) + 1$. Note that $1 \leq \varsigma^{\mathrm{L1C}}(m) \leq |S^{\mathrm{L1C}}|$.
- $\varsigma^{\mathrm{L2C}}(m)$ is the operation that extracts the *set* address for the L2 cache from a memory reference m. Given way size $|w| = 2\,\mathrm{M}/8 = 256\,\mathrm{k}$ byte and line size $|\lambda| = 128$ byte (cf. figure 21), then $\varsigma^{\mathrm{L2C}}(m) = ((m/|\lambda|) \bmod (w/|\lambda|)) + 1 = ((m/128) \bmod 2048) + 1$. Note that $1 \leq \varsigma^{\mathrm{L2C}}(m) \leq |S^{\mathrm{L2C}}|$.

- $\kappa^{\mathrm{L1C}}(m)$ is the operation that extracts the *key* part of a memory address m for the L1 caches. Given a cache of way size $|w|$, then $\kappa^{\mathrm{L1C}}(m) = m/|w| = m/16\,\mathrm{k}$. Generally, $\kappa(m) \geq 0$.
- $\kappa_a^{\mathrm{L2C}}(m)$ is the operation that extracts the *key* part of a memory address m of an application a for the shared L2 cache. Given a cache of way size $|w|$, then $\kappa_a^{\mathrm{L2C}}(m) = m/|w| + h(a) \cdot 2^{|\lambda| \cdot \alpha^{\mathrm{L2C}} \cdot |S^{\mathrm{L2C}}|}$; $h(a) \in \{1, 2, \ldots \psi\}$ is the operation that returns a unique hash for each $a \in A'$. Therefore, $\forall_{a_i,a_j \in A'} : a_i \neq a_j \Rightarrow \nexists_{m_i,m_j} : \kappa_{a_i}^{\mathrm{L2C}}(m_i) = \kappa_{a_j}^{\mathrm{L2C}}(m_j)$. Generally, $\kappa_a^{\mathrm{L2C}}(m) \geq 0$.

- t^{L1C}, t^{L2C}, t^{L1T}, t^{L2T}, t^{Mem}, and t^{Disk} refer to hit/access times shown in figure 21.

Further, I define boolean variable

- $tlb =$ **true**,

and a *conditional value*

- 'a **op** b ? c : d' as follows: If a **op** b is **true**, then the value of the whole expression is c. Otherwise, it is d. Operator **op** stands for any operator that returns a boolean value, e.g. compare operator $=$, operators $<$, $\geq$, etc.

Algorithm 13 Calculating execution time with MCCCSim.

1: **for** $i \leftarrow 1$ **to** $|\iota_F|$ **do**
2: # ################## Initialization ##################
3: # ——— Init time and address index ———
4: **for all** a in A' **do**
5: $t_{a,\iota_F\,i} \leftarrow 0$
6: $j_a \leftarrow 0$
7: **end for**
8: **if** $i = 1$ **then**
9: # ——— Reset L1 cache ———
10: **for all** a in A' **do**
11: **for** $s \leftarrow 1$ **to** $|S^{\mathrm{L1C}}|$ **do**
12: **for** $\delta \leftarrow 1$ **to** α^{L1C} **do**
13: $\zeta_{a,s}^{\mathrm{L1C},S}(\delta) \leftarrow -1$
14: **end for**
15: **end for**
16: **end for**
17: # ——— Reset shared L2 cache ———
18: **for** $s \leftarrow 1$ **to** $|S^{\mathrm{L2C}}|$ **do**
19: **for** $\delta \leftarrow 1$ **to** α^{L2C} **do**
20: $\zeta_{s}^{\mathrm{L2C},S}(\delta) \leftarrow -1$
21: **end for**
22: **end for**
23: # ——— Reset L1 TLB ———
24: **for** $\delta \leftarrow 1$ **to** K^{L1T} **do**
25: $\zeta^{\mathrm{L1T}}(\delta) \leftarrow -1$
26: **end for**
27: # ——— Reset L2 TLB ———
28: **for** $\delta \leftarrow 1$ **to** K^{L2T} **do**
29: $\zeta^{\mathrm{L2T}}(\delta) \leftarrow -1$
30: **end for**

31: # ——— Reset Memory ———
32: **for** $\delta \leftarrow 1$ **to** $\psi \cdot 2^{|m|-ld(|\varpi|)}$ **do**
33: $\zeta^{\mathrm{Mem}}(\delta) \leftarrow -1$
34: **end for**
35: **end if**

36: # ############ Process all references in ι_{Fi} ############
37: **while true do**
38: # ——— Select application with least progress p ———
39: $p \leftarrow 1.0$
40: **for all** a' in A' **do**
41: **if** $(|M_{a',\iota_{Fi}}| = 0 \ ? \ 1.0 : j_{a'}/|M_{a',\iota_{Fi}}|) < p$ **then**
42: $a \leftarrow a'$
43: $p \leftarrow j_{a'}/|M_{a',\iota_{Fi}}|$
44: **end if**
45: **end for**
46: **break** if $p = 1.0$ # process next interval ι_{Fi+1}

47: # ——— L1 access ———
48: $s \leftarrow \varsigma^{\mathrm{L1C}}(m_{a,\iota_{Fi},j_a})$
49: $\delta \leftarrow \zeta_{a,s}^{\mathrm{L1C},S}\{\kappa^{\mathrm{L1C}}(m_{a,\iota_{Fi},j_a})\}$
50: $\delta' \leftarrow \delta > \alpha^{\mathrm{L1C}} \ ? \ \alpha^{\mathrm{L1C}} : \delta$
51: **while** $\delta' > 1$ **do**
52: $\zeta_{a,s}^{\mathrm{L1C},S}(\delta') \leftarrow \zeta_{a,s}^{\mathrm{L1C},S}(\delta' - 1)$
53: $\delta' \leftarrow \delta' - 1$
54: **end while**
55: $\zeta_{a,s}^{\mathrm{L1C},S}(1) \leftarrow \kappa^{\mathrm{L1C}}(m_{a,\iota_{Fi},j_a})$
56: **if** $\delta \leq \alpha^{\mathrm{L1C}}$ **then**
57: # ——— L1 Hit ———
58: $t_{a,\iota_{Fi}} \leftarrow t_{a,\iota_{Fi}} + t^{\mathrm{L1C}}$
59: **else**

60: # ——— L1 miss ⇒ L2 access (shared) ———
61: $s \leftarrow \varsigma^{\mathrm{L2C}}(m_{a,\iota_{F}i,j_a})$
62: $\delta \leftarrow \zeta_s^{\mathrm{L2C},S}\{\kappa_a^{\mathrm{L2C}}(m_{a,\iota_{F}i,j_a})\}$
63: $\delta' \leftarrow \delta > \alpha^{\mathrm{L2C}} \ ? \ \alpha^{\mathrm{L2C}} : \delta$
64: **while** $\delta' > 1$ **do**
65: $\zeta_s^{\mathrm{L2C},S}(\delta') \leftarrow \zeta_s^{\mathrm{L2C},S}(\delta' - 1)$
66: $\delta' \leftarrow \delta' - 1$
67: **end while**
68: $\zeta_s^{\mathrm{L2C},S}(1) \leftarrow \kappa_a^{\mathrm{L2C}}(m_{a,\iota_{F}i,j_a})$
69: **if** $\delta \leq \alpha^{\mathrm{L2C}}$ **then**
70: # —— L2 hit ——
71: $t_{a,\iota_{F}i} \leftarrow t_{a,\iota_{F}i} + t^{\mathrm{L2C}}$
72: **else**
73: # ——— L2 miss ⇒ TLB and Memory access ———
74: **if** *tlb* = **true then**
75: # ——— L1 TLB access ———
76: $\delta \leftarrow \zeta^{\mathrm{L1T}}\{\varpi_a(m_{a,\iota_{F}i,j_a})\}$
77: $\delta' \leftarrow \delta > K^{\mathrm{L1T}} \ ? \ K^{\mathrm{L1T}} : \delta$
78: **while** $\delta' > 1$ **do**
79: $\zeta^{\mathrm{L1T}}(\delta') \leftarrow \zeta^{\mathrm{L1T}}(\delta' - 1)$
80: $\delta' \leftarrow \delta' - 1$
81: **end while**
82: $\zeta^{\mathrm{L1T}}(1) \leftarrow \varpi_a(m_{a,\iota_{F}i,j_a})$
83: **if** $\delta \leq K^{\mathrm{L1T}}$ **then**
84: # ——— L1 TLB hit ———
85: $t_{a,\iota_{F}i} \leftarrow t_{a,\iota_{F}i} + t^{\mathrm{L1T}}$
86: **else**
87: # ——— L1 TLB miss ⇒ L2 TLB access ———
88: $\delta \leftarrow \zeta^{\mathrm{L2T}}\{\varpi_a(m_{a,\iota_{F}i,j_a})\}$
89: $\delta' \leftarrow \delta > K^{\mathrm{L2T}} \ ? \ K^{\mathrm{L2T}} : \delta$

90: **while** $\delta' > 1$ **do**
91: $\zeta^{\mathrm{L2T}}(\delta') \leftarrow \zeta^{\mathrm{L2T}}(\delta' - 1)$
92: $\delta' \leftarrow \delta' - 1$
93: **end while**
94: $\zeta^{\mathrm{L2T}}(1) \leftarrow \varpi_a(m_{a,\iota_{\digamma}i,j_a})$
95: **if** $\delta \leq K^{\mathrm{L2T}}$ **then**
96: # ——— L2 TLB hit ———
97: $t_{a,\iota_{\digamma}i} \leftarrow t_{a,\iota_{\digamma}i} + t^{\mathrm{L1T}} + t^{\mathrm{L2T}}$
98: **else**
99: # ——— L2 TLB miss ⇒ Read page address from memory ———
100: $t_{a,\iota_{\digamma}i} \leftarrow t_{a,\iota_{\digamma}i} + t^{\mathrm{L1T}} + t^{\mathrm{L2T}} + t^{\mathrm{Mem}}$
101: **end if**
102: **end if**
103: **end if**
104: # ——— Read page ———
105: $\delta \leftarrow \zeta^{\mathrm{Mem}}\{\varpi_a(m_{a,\iota_{\digamma}i,j_a})\}$
106: **if** $\delta \leq \psi \cdot 2^{|m|-ld(|\varpi|)}$ **then**
107: # ——— Page is in memory ———
108: $t_{a,\iota_{\digamma}i} \leftarrow t_{a,\iota_{\digamma}i} + t^{\mathrm{Mem}}$
109: **else**
110: # ——— Page is not in memory ⇒ fetch from disk ———
111: $\zeta^{\mathrm{Mem}}(\zeta^{\mathrm{Mem}}\{-1\}) \leftarrow \varpi_a(m_{a,\iota_{\digamma}i,j_a})$
112: $t_{a,\iota_{\digamma}i} \leftarrow t_{a,\iota_{\digamma}i} + t^{\mathrm{Mem}} + t^{\mathrm{Disk}}$
113: **end if**
114: **end if**
115: **end if**
116: $j_a \leftarrow j_a + 1$
117: **end while**
118: **end for**

3.2 General Ranking Performance

In this section, I evaluate the cache contention prediction techniques presented in chapter 2 by means of so-called *general ranking performance*. As general ranking performance, I define the ability of a cache contention prediction method to rank candidate co-schedules $C_a \in \mathbf{C}_a^\psi = \{C_{a,1}^\psi, C_{a,2}^\psi, \dots\}$ of an application a by the contention they introduce to a.

NMRD - Normalized Mean Ranking Difference

NMRD (normalized mean ranking difference) is one of the evaluation functions (13) referenced in figure 17. The NMRD evaluation function calculates the normalized mean ranking difference by comparing rankings $\rho^{\text{pred}}_{C_a,\iota_F i}$ that are estimated by a prediction method (cf. equation 70) to actual rankings $\rho^{\text{sim}}_{C_a,\iota_F i}$ (cf. equation 71) determined by MCCCSim.
Given both $\rho^{\text{pred}}_{C_a,\iota_F i}$ and $\rho^{\text{sim}}_{C_a,\iota_F i}$ for all $C_a \in \mathbf{C}_a^\psi = \mathfrak{C}(A\backslash\{a\}, \psi - 1)$, I calculate differences

$$\Delta\rho_{C_a,\iota_F i} = \left|\rho^{\text{pred}}_{C_a,\iota_F i} - \rho^{\text{sim}}_{C_a,\iota_F i}\right| \tag{72}$$

between predicted and actual ranking positions for each $C_a \in \mathbf{C}_a^\psi$. Then, I add all $\Delta\rho_{C_a,\iota_F i}$ and divide it by $|\mathbf{C}_a^\psi|$ to calculate so-called mean ranking difference MRD, i.e. the number of positions that predicted rankings differ from actual rankings on average:

$$MRD_{\mathbf{C}_a^\psi,\iota_F i} = \frac{1}{|\mathbf{C}_a^\psi|} \sum_{C_a \in \mathbf{C}_a^\psi} \Delta\rho_{C_a,\iota_F i} \tag{73}$$

As an example, figure 22 shows how *MRD* is calculated for $\psi = 2$ and $a =$ astar. As you might suggest, the MRD value depends on the number of candidate co-schedules $|\mathbf{C}_a^\psi|$, rendering it difficult to compare MRD values of various ψ to one another. To overcome this, I normalize MRD values by the maximum MRD value possible.
Without proof, I specify that the maximum MRD value can be observed at co-schedule rankings presented in figure 23 a) for an even number of co-schedules, and in figure 23 b) for an odd number of co-schedules. With this assumption, maximum *MRD* calculates as follows:

$$MRD^{\max}_{\mathbf{C}_a^\psi} = \frac{\left\lceil \frac{|\mathbf{C}_a^\psi|}{2} \right\rceil \cdot \left\lfloor \frac{|\mathbf{C}_a^\psi|}{2} \right\rfloor \cdot 2}{|\mathbf{C}_a^\psi|} \tag{74}$$

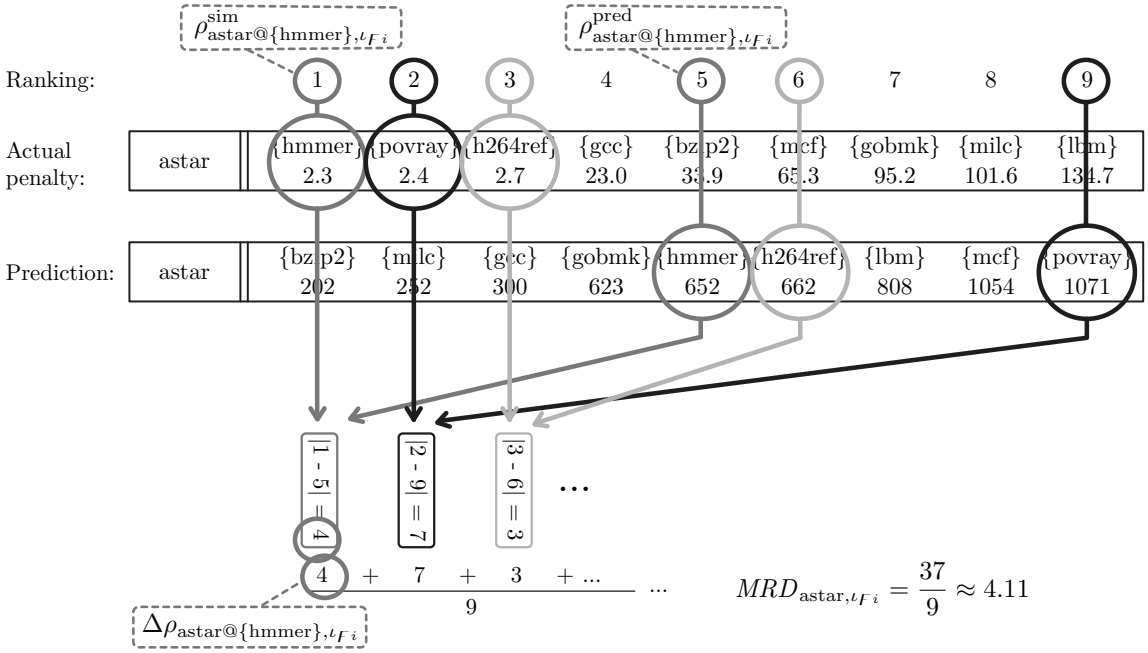


Figure 22: Determination of the MRD mean ranking distance value.

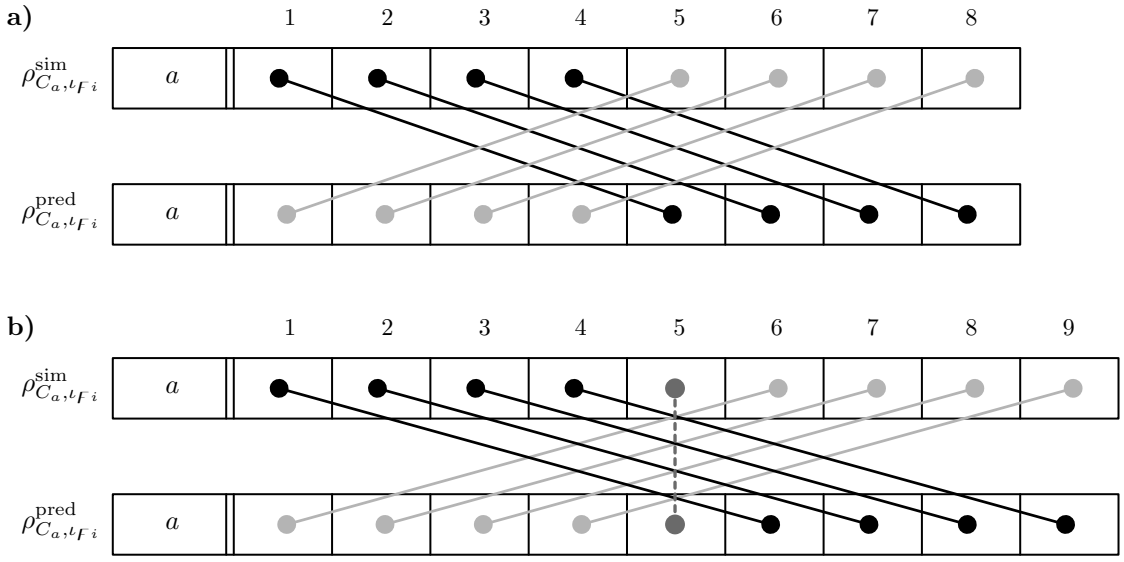


Figure 23: Determination of the maximum MRD mean ranking distance value for **a)** an even number of candidate co-schedules, and **b)** an odd number of candidate co-schedules. Identical co-schedules in both simulated and predicted rankings are indicated by connecting lines.

In equation 74, term $\left\lceil \frac{|\mathbf{C}_a^\psi|}{2} \right\rceil$ refers to the differences and $\left\lfloor \frac{|\mathbf{C}_a^\psi|}{2} \right\rfloor \cdot 2$ to the number of occurrences of these differences. For $\psi \in \boldsymbol{\psi} = \{2, 4, 8\}$, $MRD^{\max}_{\mathbf{C}_a^\psi}$ calculates to

- $\psi = 2 \Rightarrow |\mathbf{C}_a^\psi| = 9 \Rightarrow MRD^{\max}_{\mathbf{C}_a^\psi} = 4.44$,
- $\psi = 4 \Rightarrow |\mathbf{C}_a^\psi| = 84 \Rightarrow MRD^{\max}_{\mathbf{C}_a^\psi} = 42$,
- $\psi = 8 \Rightarrow |\mathbf{C}_a^\psi| = 36 \Rightarrow MRD^{\max}_{\mathbf{C}_a^\psi} = 18$.

Note that I verified $MRD^{\max}$ for $\psi = 2$ by simulation. For $\psi = 4$ and $\psi = 8$ however, simulation would be too time consuming due to the high number of permutations (36! and 84! respectively) that would have to be calculated and analyzed.
Given the maximum *MRD* value, I perform normalization on *MRD* according to

$$NMRD_{\mathbf{C}_a^\psi, \iota_{\digamma} i} = \frac{MRD_{\mathbf{C}_a^\psi}}{MRD^{\max}_{\mathbf{C}_a^\psi}} = \frac{\sum\limits_{C_a \in \mathbf{C}_a^\psi} \Delta\rho_{C_a, \iota_{\digamma} i}}{\left\lceil \frac{|\mathbf{C}_a^\psi|}{2} \right\rceil \cdot \left\lfloor \frac{|\mathbf{C}_a^\psi|}{2} \right\rfloor \cdot 2}. \tag{75}$$

Up to now, I only presented the way I calculate the normalized mean ranking difference for *one* application a and *one* interval $\iota_{\digamma} i$. To get an overall normalized mean ranking difference that can be applied to evaluate cache contention prediction techniques, I average ranking distances $\Delta\rho_{C_a, \iota_{\digamma} i}$ over all applications $a \in A$, over all possible candidate co-schedules $C_a \in \mathbf{C}_a^\psi = \mathfrak{C}(A \backslash \{a\}, \psi - 1)$ for each such a, and over all intervals $\iota_{\digamma} i \in \iota_{\digamma}$. I calculate $NMRD(\psi, \digamma)$ for a given number of parallel processor cores ψ and a given window size $\digamma$ according to

$$\begin{aligned} NMRD(\psi, \digamma) &= \frac{1}{|A|} \cdot \sum_{a \in A} \frac{1}{|\iota_{\digamma}|} \cdot \sum_{\iota_{\digamma} i \in \iota_{\digamma}} NMRD_{\mathbf{C}_a^\psi, \iota_{\digamma} i} \\ &= \frac{1}{|A|} \cdot \sum_{a \in A} \frac{1}{|\iota_{\digamma}|} \cdot \sum_{\iota_{\digamma} i \in \iota_{\digamma}} \frac{\sum\limits_{C_a \in \mathbf{C}_a^\psi} \Delta\rho_{C_a, \iota_{\digamma} i}}{\left\lceil \frac{|\mathbf{C}_a^\psi|}{2} \right\rceil \cdot \left\lfloor \frac{|\mathbf{C}_a^\psi|}{2} \right\rfloor \cdot 2}. \end{aligned} \tag{76}$$

Figure 24 shows the results of my evaluation calculating $NMRD(\psi, \digamma)$ for the number of parallel processor cores $\psi \in \boldsymbol{\psi} = \{2, 4, 8\}$ and interval sizes $\digamma \in \boldsymbol{\digamma} = \{2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}, 2^{25}, 2^{26}, 2^{27}, 2^{28}, 2^{29}\}$ instructions. Note that the results presented in figure 24 are

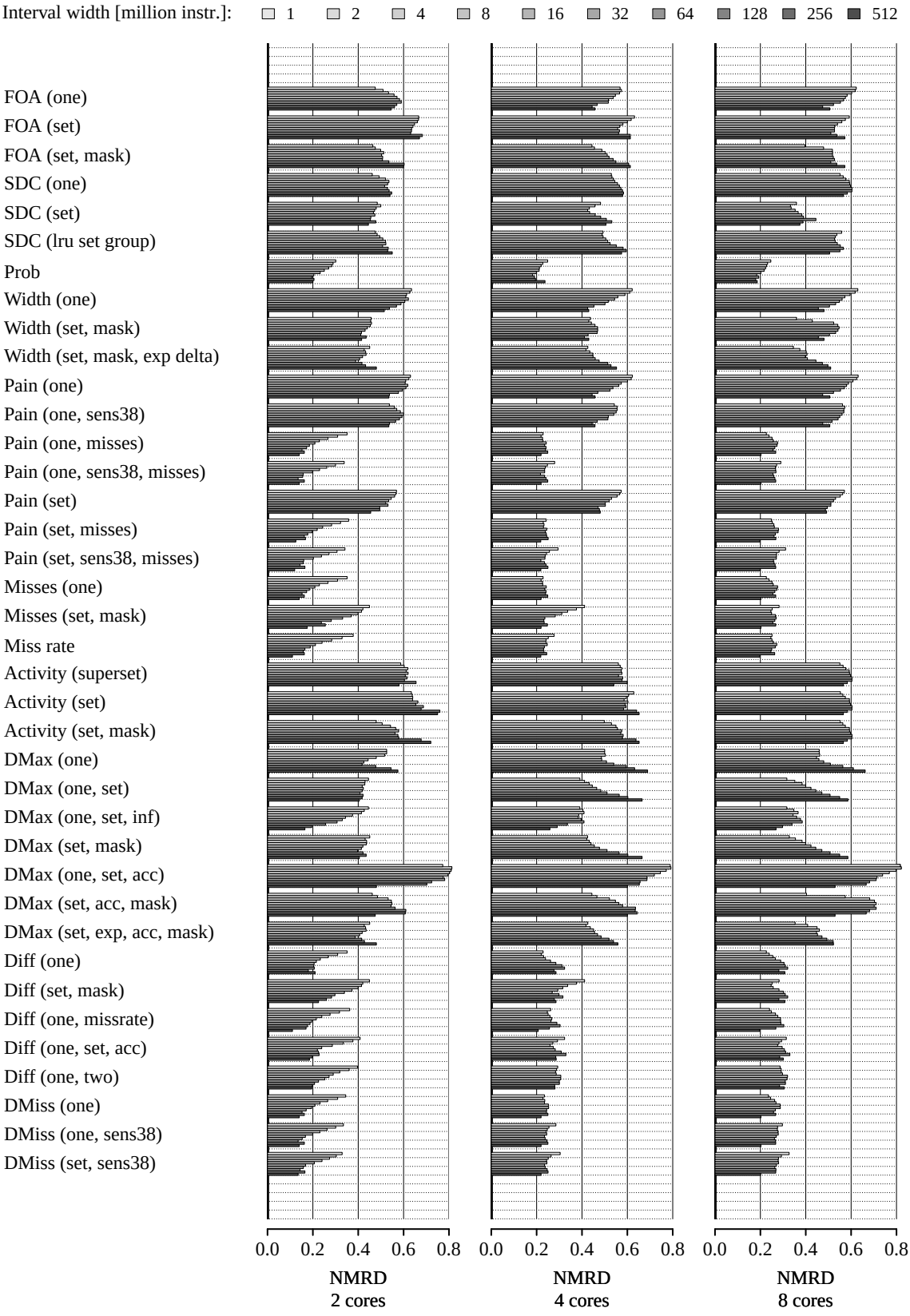


Figure 24: *NMRD* performance of cache contention prediction methods for $\psi \in \boldsymbol{\psi} = \{2, 4, 8\}$ processor cores and interval sizes $I \in \boldsymbol{I} = \{2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}, 2^{25}, 2^{26}, 2^{27}, 2^{28}, 2^{29}\}$ instructions, averaged over all intervals $\iota_{Ii} \in \iota_I$ and all applications $a \in A = \{$*astar*, *bzip2*, *gcc*, *gobmk*, *h264ref*, *hmmer*, *lbm*, *mcf*, *milc* and *povray*$\}$.

averaged over all $|A| \cdot \binom{|A|-1}{\psi-1} \cdot \frac{2^{29}}{F}$ combinations. For example, bars in figure 24 that depict NMRD for an interval width of $F = 2^{20}$ instructions and $\psi = 4$ processor cores are averaged over $10 \cdot \binom{9}{3} \cdot 512 = 430080$ values each. For an example of the *distribution* of NMRD values, see section 'Distributions' in the appendix. Note that a comprehensive collection of distributions is presented in [Zwick, 2010a]. In the following, I discuss figure 24.

Good scalability regarding the number of cores ψ

At first sight, the three columns in figure 24 that represent NMRD performance for $\psi =$ 2, 4 or 8 processor cores sharing the L2 cache look quite similar. Although there are differences, methods that perform 'good' (low NMRD values) when predicting contention for a given number of processor cores ψ do not suddenly perform 'bad' (high NMRD values) for different ψ and vice versa. For example, the *Activity vector* based methods as well as the *FOA* methods show poor prediction accuracy for all $\psi \in \boldsymbol{\psi} = \{2, 4, 8\}$; on the contrary, the *Prob* method, methods that rely on cache misses as predictor (e.g. *Misses*, *Pain (..., misses)*, *DMiss*), as well as methods that are based on the number of different keys that map to the same cache set (*Diff*) show good prediction accuracy for all $\psi \in \boldsymbol{\psi}$. This means that all methods generally show *good scalability* regarding the number of processor cores ψ: There is no method that is solely applicable for a high or a low number of processor cores.

Note that I chose the maximum number of processor cores $\psi = 8$ in order to match associativity α of the L2 data cache: In *per-cache-set* stack distance histograms $H^{sd,S}_{a,\iota_{F\,i},s}$, entries $1 \ldots \alpha$ often incorporate none or just a single entry being different from 0 (cf. figure 43 in the appendix). As a consequence, there is only little cache interference if $\psi = 2$. For $\psi = 8$, however, cache sets get filled up to a much larger extent and threads interfere much more than for $\psi = 2$. Therefore, values of $\psi \in \{2, 4, 8\}$ cover a broad range of cache interference and allow for a more thoroughly evaluation of cache contention prediction methods than it has been performed in the past. Such an evaluation seems to be important, as Chen and Aamodt reported that the *Prob* method "becomes inaccurate for systems with a large number of threads, particularly once the number of threads sharing a cache approaches or exceeds the associativity of the cache" [Chen and Aamodt, 2009], and I wanted to investigate this statement for other methods than the *Prob* method as well. According to my

results, however, neither the *Prob* method, nor any other method shows a severe performance degradation in case that the number of processor cores approaches associativity α. Note, however, that for $\psi = 2$ processor cores, as applied in [Chandra et al., 2005], my evaluation ranks accuracy of the *FOA (one)*, *SDC (one)* and *Prob* techniques just the way as it has been reported by Chandra et al.: The *Prob* technique is the most accurate one, followed by *SDC (one)* and *FOA (one)*.

Poor performance of access or hit based methods

As you can see from figure 24, methods that are primarily based on the number of cache accesses generally show poor prediction performance only (*FOA*, *SDC*, *Activity vector*, *Pain (one)*, *Pain (one, sens38)*, *Pain (set)* and *DMax (..., acc, ...)* methods). Note that expression 'primarily based on the number of cache accesses' also includes methods that are based on stand-alone cache hits or the distribution of entries in stack distance histograms, as $H^{sd}_{a,\iota_{F\,i}}(1) \approx \sum_{\delta=1}^{\alpha} H^{sd}_{a,\iota_{F\,i}}(\delta) \approx \sum_{\delta=1}^{\alpha+1} H^{sd}_{a,\iota_{F\,i}}(\delta)$ (cf. figures 41 and 42 in the appendix). Therefore, *nearly all state-of-the-art cache contention prediction methods are actually based on the number of memory accesses, which is about the same as* $H^{sd}_{a,\iota_{F\,i}}(1)$. For example, the FOA method calculates an effective associativity $\alpha'_{C_a,\iota_{F\,i}}$ by relating the number of accesses of application a to the number of accesses of applications $\{a\} \cup C_a$. The recently proposed *Pain (one)* method ([Zhuravlev et al., 2010], [Fedorova et al., 2010]), that introduces the concept of application intensity and application sensitivity, relates intensity to the number of cache accesses (cf. equation 29), and sensitivity to stack distance histogram entries, weighted by their stack distance (cf. equation 28). Applying the weighted number of stack distance entries as sensitivity measure actually makes sense, as only a memory reference that is a hit on stand-alone execution can turn into a contention miss. However, due to the domination of $H^{sd}_{a,\iota_{F\,i}}(1)$ (cf. figures 41 and 42 in the appendix) scaling $H^{sd}_{a,\iota_{F\,i}}(\delta)$ linearly by δ often does not have a big enough effect; therefore, sensitivity is still dominated by $H^{sd}_{a,\iota_{F\,i}}(1)$, which does not seem to be a favorable selection for reasons discussed in section 2.6, i.e. *Pain* variation 'one, misses'. If you compare the results of *Pain* variation 'one' to those of *Pain* variation 'one, misses' in figure 24, my considerations in section 2.6 seem to prove true: The number of stand-alone cache misses definitely is a much better intensity measure than the total number of memory references, and val-

ues obtained from a combination of stack distance entries $1 \dots \alpha$ or stack distance entries $1 \dots \alpha + 1$ are often too strongly dominated by $H^{sd}_{a,\iota_{F}i}(1)$.

Good performance of miss based and related methods

From figure 24 you can see that it is not only *Pain* variation 'one, misses' that shows good prediction performance, but all prediction methods that apply cache misses or a related measure as predictor in general: *Pain (..., misses)*, *Misses*, *Miss rate*, and *DMiss* all show comparable prediction performance — and it does not seem to be an easy task to significantly improve prediction performance of miss based methods.

Besides miss based methods, the *Diff* and the *Prob* methods show good prediction performance as well. Note that I actually introduced *Diff* variation 'one, two' as an improvement to *Diff* variation 'one' in order to 'exclude memory references that are referenced only once and therefore do contribute to cold misses only'. However, variation 'one, two' turns out *not* to be an improvement to variation 'one'; it has rather a degradational effect. Thinking it over, however, the simulation results indeed make sense: If you adopt the sensitivity/intensity model applied in the *Pain* method, cold misses do not contribute to the *sensitivity* part, but they *do* contribute to the *intensity* part. The intensity part, however, seems to be the significant part, as *Pain (one, misses)* and *Misses (one)* show exactly the same shape for all window sizes F and all number of cores $\psi \in \{2, 4, 8\}$. The high domination of the intensity part further points out when comparing the results of *Pain (one, misses)* to those of *Pain (one, sens38, misses)*: Both variations show almost the same performance, although the sensitivity applied in both variations varies by multiple orders of magnitude. Contrary, the difference in performance between *Pain (one)* and *Pain (one, sens38)* is much larger, in particular for small window sizes F and a high number of processor cores.

Regarding miss based and related methods, you can observe that there is a significant performance degradation if the number of cores and the interval width limits the number of memory references included in the prediction, as it is the case for $\psi = 2$ and low values of F. Regarding $\psi = 2$ only, a reason for this observation might be that predictors are generally calculated on a per-interval basis, i.e. they incorporate address information from a single interval ι_{Fi} only, while the actual memory access time determined by MCCC-

Sim incorporates effects from all previously executed intervals; and the longer the interval width, the more memory references occur, which renders the error small. Arguing this way, however, the same effect would have to occur for $\psi = 4$ and $\psi = 8$, which is not the case. A more reasonable explanation for this behavior might be a method's *sensitivity* to cache contention: Generally, a small interval width and a low number of processor cores will introduce less cache contention than a large interval width and a high number of processor cores. This statement seems to be obvious, as a larger interval width increases the probability that co-scheduled applications provide memory access patterns that displace LRU stack entries. Similarly, a higher number of processor cores ψ introduces more co-scheduled applications that compete for a cache of constant size. If there is only a small amount of cache contention introduced, a prediction method has to be very sensitive to differentiate between cases where cache contention actually turns into additional cache misses, and where not. As figure 24 suggests, most methods perform better if contention is more obvious, and degrade in performance if cache contention is low.

But the main reason might be as follows: A stand-alone cache miss implies that an address key is pushed onto the LRU stack. To predict cache contention exploiting this observation, a key that is pushed onto the LRU stack actually has to replace an LRU stack entry that has to be re-fetched later on. If ψ and F are small, it is less likely that pushing a key onto the LRU stack actually replaces an entry that has to be re-fetched. For higher values of F, this probability increases, but not linearly: If you assume that an application a will reference $\varkappa_{a,\iota_{F\,i}}$ different cache lines in an interval of size F, then a will, on average, reference much less than $n \cdot \varkappa_{a,\iota_{F'i}}$ different cache lines in intervals of width $F' = n \cdot F$. However, the number of different cache lines per set that are accessed increases linearly with the number of applications ψ that concurrently access the cache (assuming low spatial locality and comparable applications), if the applications do not share any data, what is assumed. If the number of applications ψ results in enough references to different cache lines to fill up the LRU stack, then every stand-alone cache miss will displace an already fetched entry from the LRU stack and a contention miss occurs if the displaced key is re-fetched.

Per-cache-set calculation not beneficial

Analyzing figure 24 reveals that predicting cache contention on a per-cache-set-basis does not have the big positive effect one might expect: While some methods benefit from per-cache-set based prediction, like the SDC method for example, others do not show any significant performance change or even show performance degradation, as it is the case for the FOA method. Figure 25 qualitatively summarizes figure 24 regarding performance gain achieved by per-cache-set calculation. I apply '+' to indicate a positive and '−' to indicate a negative effect on NMRD performance; '=' means that there is no significant effect.

Compared Methods/Variations	Access or Miss based	Gain
FOA (one) ↔ FOA (set)	Access	−
SDC (one) ↔ SDC (set)	Access	+
Pain (one) ↔ Pain (set)	Access	+
Pain (one, misses) ↔ Pain (set, misses)	Miss	=
Pain (one, sens38, misses) ↔ Pain (set, sens38, misses)	Miss	=
Activity (superset) ↔ Activity (set)	Access	−
DMax (one) ↔ DMax (one, set)	n/a	+
DMiss (one, sens38) ↔ DMiss (set, sens38)	Miss	=

Figure 25: Qualitative evaluation of prediction on per-cache-set basis.

Applying per-cache-set prediction, methods that predict cache contention primarily by the number of cache accesses or by the number of cache hits respectively, achieve either a performance gain, or they suffer a degradation. But even those access/hit based methods that benefit from per-cache-set calculation do not even approach the performance of miss based methods. Methods that rely on the number of cache misses to predict cache contention, however, do not show any significant change in performance when calculated per cache set; the only significant effect is an increased execution time (cf. section 3.4). Therefore, per-cache-set prediction does not seem to be a reasonable approach, which might be

caused by a limited amount of spatial locality available in applications A.

Note that I classify Pain variation 'one, misses' to rely rather on cache misses than on cache accesses, although the number of cache accesses in $\chi^{sens}_{a,\iota_F i}$ exceeds the number of cache misses $H^{sd}_{a,\iota_F i}(\alpha+1)$ by several orders of magnitude. However, if you compare the results of Pain (one, misses) to the results of Pain (one) and Misses (one) (cf. figure 24), you can see that it is the *misses* and not the number of accesses that causes prediction results.

Similarly, I classify the Activity vector method to be access based rather than miss based. Although the Activity vector method relies on both the number of accesses and the number of misses, its performance suggests that the number of accesses is the dominating factor: The Activity vector method rather performs like the FOA method than like the Misses method, which might be caused by an improper selection of Ω^{miss}_F.

Limited gain of masking

In figure 26, I qualitatively summarize the results presented in figure 24 regarding masking performance, i.e. gain in NMRD performance when selecting only those cache sets $s \in S$ with $\sum_{a' \in \{a\} \cup C_a} \delta^{max,S}_{a',\iota_F i,s} > \alpha$.

Figure 26 shows that masking has quite different effects on the various methods/variations: While four methods benefit from masking, one method does not show any significant performance change, while two other methods even show a performance degradation. Interestingly, there seems to be a correlation between masking performance and the exploited resource the corresponding prediction technique is primarily based on: Methods that *benefit* from masking primarily rely on the number of memory *accesses* as predictor, while methods that degrade in performance when applying masking are based on the number of cache *misses* or a related measure. Note that I classify the Activity vector method to be an access based method, as I already did in the previous section.

Compared Methods/Variations	Access or Miss based	Gain
FOA (set) ↔ FOA (set, mask)	Access	+
Width (one) ↔ Width (set, mask)	Access	+
Misses (one) ↔ Misses (set, mask)	Miss	−
Activity (set) ↔ Activity (set, mask)	Access	+
DMax (one, set) ↔ DMax (set, mask)	n/a	=
DMax (one, set, acc) ↔ DMax (set, acc, mask)	rather Access	+
Diff (one) ↔ Diff (set, mask)	rather Miss	−

Figure 26: Qualitative evaluation of masking performance shown in figure 24.

If a method primarily relies on the number of misses, its predictor is related to $H^{sd}(\alpha+1)$. But masking does not consider $H^{sd}(\alpha+1)$, but $H^{sd}(1\ldots\alpha)$. If a method relies on the number of accesses, it applies the whole stack distance histogram $H^{sd}(1\ldots\alpha+1)$ for prediction. However, as $\sum_{\delta=1}^{\alpha} H^{sd}(\delta) \gg H^{sd}(\alpha+1)$, such methods are actually based on the number of *hits*. If a method predicts cache contention primarily by the number of hits, masking will generally have a positive influence on the prediction, as it masks out those cases in which stand-alone cache hits are not able to displace a stand-alone cache hit from the LRU stack.

In a nutshell: Masking has high potential to improve NMRD performance of cache contention prediction methods that primarily apply the number of cache hits/accesses as predictor. However, such prediction methods generally show poor prediction performance and the effect of masking is not big enough to make those methods a serious competitor to miss based methods. Applied on miss based methods, however, masking has a rather degrading effect and cannot be recommended.

Further note that a higher number of processor cores reduces the effect of masking (cf. figure 24), as a high number of applications that are executed in parallel introduce additional references to each cache set, which reduces the number of cache sets that get masked.

Weighting stack distance entries

Giving more weight to stack distance entries with higher values of δ than to those with lower values of δ achieves varying results. There is a definite performance gain in some cases: Pain (one) $\leftrightarrow$ Pain (one, sens38); DMax (set, acc, mask) $\leftrightarrow$ DMax (set, exp, acc, mask). But there are also many cases that show both a performance gain and a performance degradation, depending on the interval width $\mathcal{F}$ and the number of processor cores ψ: Width (set, mask) $\leftrightarrow$ Width (set, mask, exp delta); Pain (one, misses) $\leftrightarrow$ Pain (one, sens38, misses); Pain (set, misses) $\leftrightarrow$ Pain (set, sens38, misses); DMiss (one) $\leftrightarrow$ DMiss (one, sens38). For small interval sizes and a small number of processor cores, giving more weight to stack distance entries with higher δ and less weight to those with lower δ seems to be a good choice.

Wider stack distance histograms

Comparing SDC (one) to SDC (lru set group) suggests that performance of the SDC method can slightly be improved by increasing the number of stack distance histogram entries from α (+1) to a larger value $\alpha \cdot |G|$ (+1), partitioning the cache set into $|G|$ groups and tracking LRU information of group accesses. However, the obtained gain is focused on cases with a higher number of cores (e.g. $\psi \in \{4, 8\}$) only and is rather small. In some cases of ψ and $\mathcal{F}$, however, performance slightly degrades.

Infinite LRU stack

Comparing DMax (one, set) to DMax (one, set, inf), performance degradation of DMax (one, set) for higher values of ψ and higher values of $\mathcal{F}$ obviously originates from the limited capacity of the LRU stack: For high ψ and especially for high $\mathcal{F}$, many elements that have already been fetched to LRU stacks get displaced; as $\delta^{max,S}_{a,\iota_{\mathcal{F}}i,s}$ is limited to α for DMax variation 'one, set', values of $\delta^{max,S}_{a,\iota_{\mathcal{F}}i,s} > \alpha$ cannot be distinguished and prediction performance degrades. If $\delta^{max,S}_{a,\iota_{\mathcal{F}}i,s}$, however, is calculated applying an LRU stack of infinite size, then $\delta^{max,S}_{a,\iota_{\mathcal{F}}i,s}$ is not limited to α and values of $\delta^{max,S}_{a,\iota_{\mathcal{F}}i,s}$ beyond α can be distinguished, achieving acceptable prediction performance.

TLB effects

With figures 27 and 28 on the next doublepage, I show that effects from memory address translation, such as TLB misses, hardly have any impact on NMRD performance. Figure 27 is identical to figure 24 and represents NMRD performance in case that algorithm 13 is executed with $tlb =$ **true**, i.e. ground truth reference includes TLB backed memory address translation, as it is common in processor systems. Figure 28 represents NMRD performance in case that algorithm 13 is executed with $tlb =$ **false**, which corresponds to a processor architecture that does *not* incorporate address translation and therefore cannot suffer from additional effects introduced from TLB misses. As both figures are almost identical, additional TLB misses do not seem to have any significant effect on prediction performance, not even if $\psi = 8$, when TLBs are shared amongst many applications.

Comparing results to others

Given $\psi = 2$, as it is the case in [Chandra et al., 2005], figure 24 shows that the Prob method performs much better than the SDC (one) method, and SDC (one) outperforms FOA (one). These results exactly match Chandra et al.'s observations.

Chen and Aamondt's observation, however, that the Prob method becomes inaccurate if the number of threads approaches associativity α cannot be affirmed. According to figure 24, NMRD performance of the Prob method is nearly independet of ψ and the minor differences that can be observed indicate a rather *better* performance for higher ψ than for lower ψ. For $\psi \in \{4, 8\}$, the Prob method turns out to be the best cache contention prediction technique regarding NMRD accuracy.

In [Fedorova et al., 2010] however, the authors demonstrate that Pain (one) achieves better performance than the Miss rate method, and that the Miss rate method outperforms the SDC method. The authors perform their evaluation applying a *real hardware* system and explain poor SDC performance by the assumption that "contention for the shared cache [...] is not the main cause of performance degradation experienced by competing applications on multicore systems. Contention for other shared resources, such as the front-side bus, prefetching resources and the memory controller are the dominant causes for performance degradation" [Fedorova et al., 2010].

In my evaluation, however, the SDC method shows same poor performance as presented in [Fedorova et al., 2010]. Note, however, that the MCCCSim simulator I apply as ground truth reference does *not* model contention effects on busses, prefetching resources and memory controller. As a consequence, poor performance of the SDC method cannot be explained by contention of other resources than caches. Anyway, contention regarding the front-side bus and the memory controller should be highly correlated to last level cache contention, as only those references can contend for the bus that turn out to be contention misses at all.

According to my evaluation and the discussion regarding hit/access based vs. miss based methods in previous sections, the SDC method seems to achieve poor performance due to its addiction to cache *hits/accesses* as predictor. And I already demonstrated that this is an inappropriate way to predict cache contention due to high temporal program locality that can be observed from the high concentration of stack distance histograms.

So, the poor performance of the SDC method can be explained, just as the excellent applicability of stand-alone cache misses or cache miss rates to predict cache contention. Given the MCCCSim simulator as ground truth measure, however, I cannot reproduce the good prediction results of Pain (one) that have been presented in [Fedorova et al., 2010]. And I cannot explain why Pain (one) should achieve good prediction results at all, as it primarily relies on the number of cache hits/accesses as predictor. Applying cache misses as measure of intensity, however, dramatically improves performance of the Pain method — and it seems to be obvious that the number of cache accesses is an inappropriate intensity measure. Possibly, the apparent performance difference between Pain (one) in this thesis and the Pain method presented in [Fedorova et al., 2010] are caused by different distributions of memory references of the applied applications.

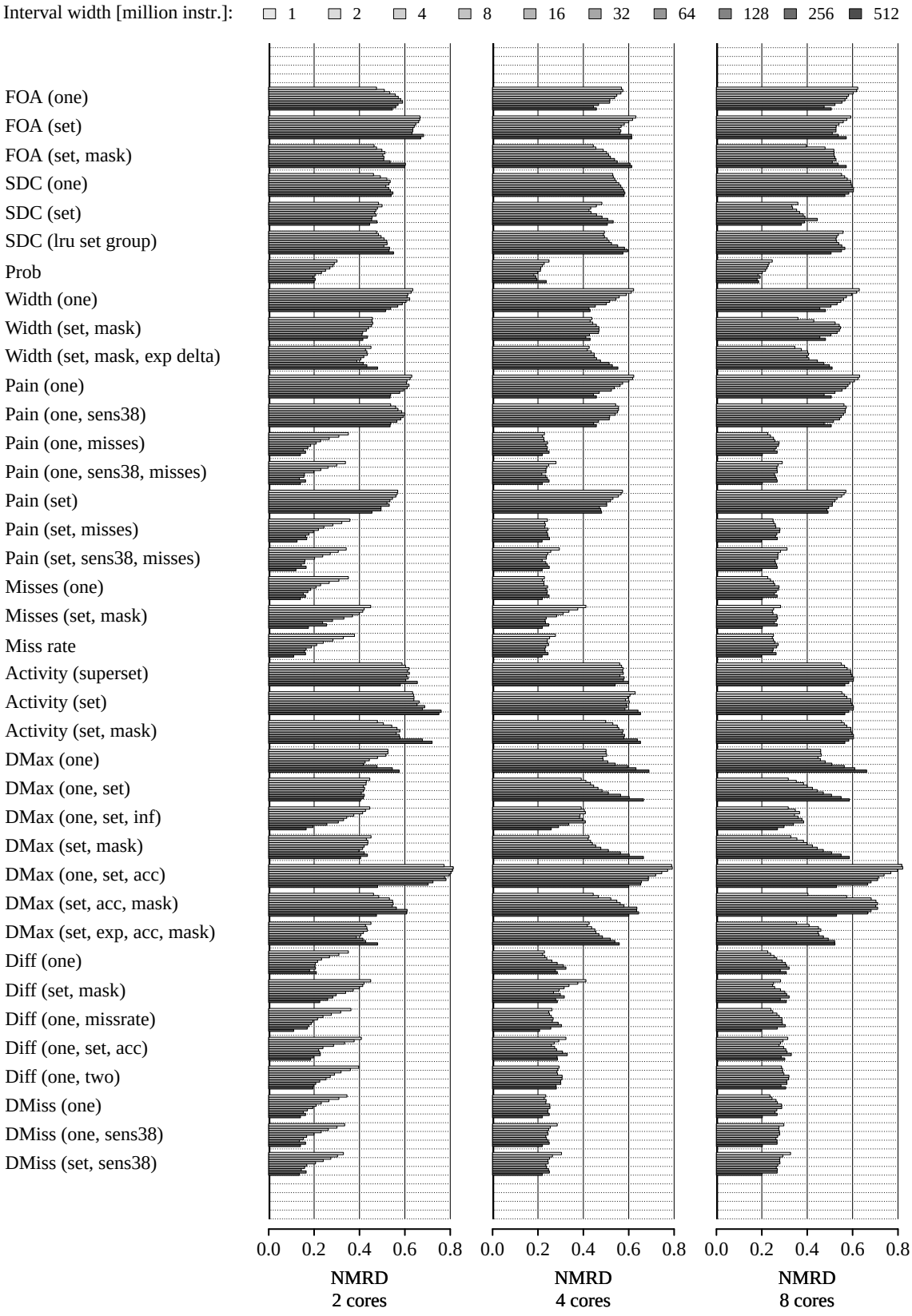


Figure 27: NMRD performance as presented in figure 24, i.e. with *address translation enabled* in the MCCCSim simulator, considering additional penalties from TLB misses.

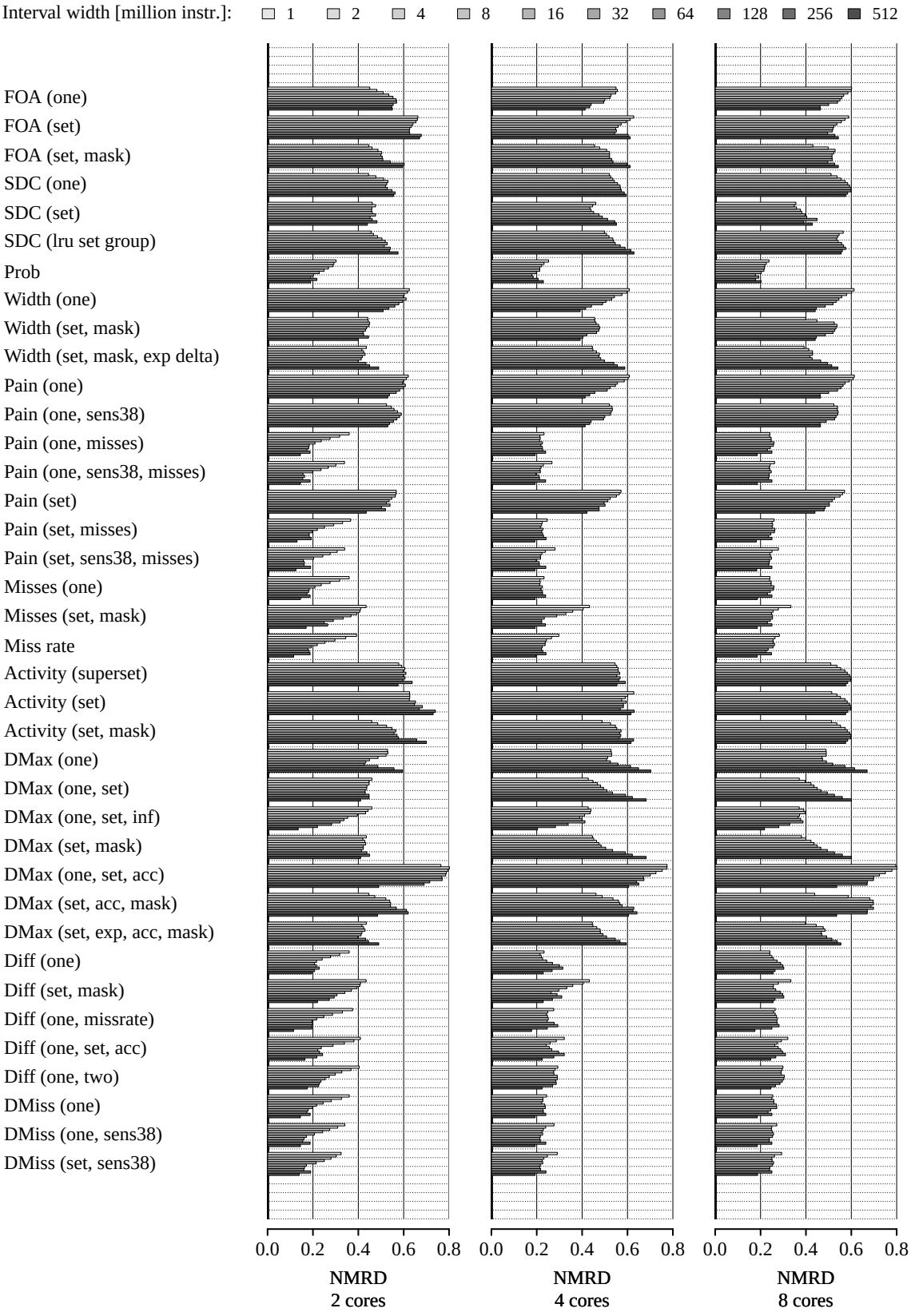


Figure 28: NMRD performance if effects from address translation, such as TLB misses, are not considered.

MP - Mean Penalty

While NMRD evaluates cache contention prediction techniques regarding the difference between predicted and actual ranking positions of a set of candidate co-schedules, the MP (mean penalty) measure performs its evaluation regarding the *time* predicted rankings are off actual rankings. Figure 29 presents the key principle of the *MP* evaluation function for $a =$ astar and co-schedules $\mathbf{C}_a = \{\{\text{bzip}\}, \{\text{gcc}\}, \{\text{gobmk}\}, \{\text{h264ref}\}, \{\text{hmmer}\}, \{\text{lbm}\}, \{\text{mcf}\}, \{\text{milc}\}, \{\text{povray}\}\}$.

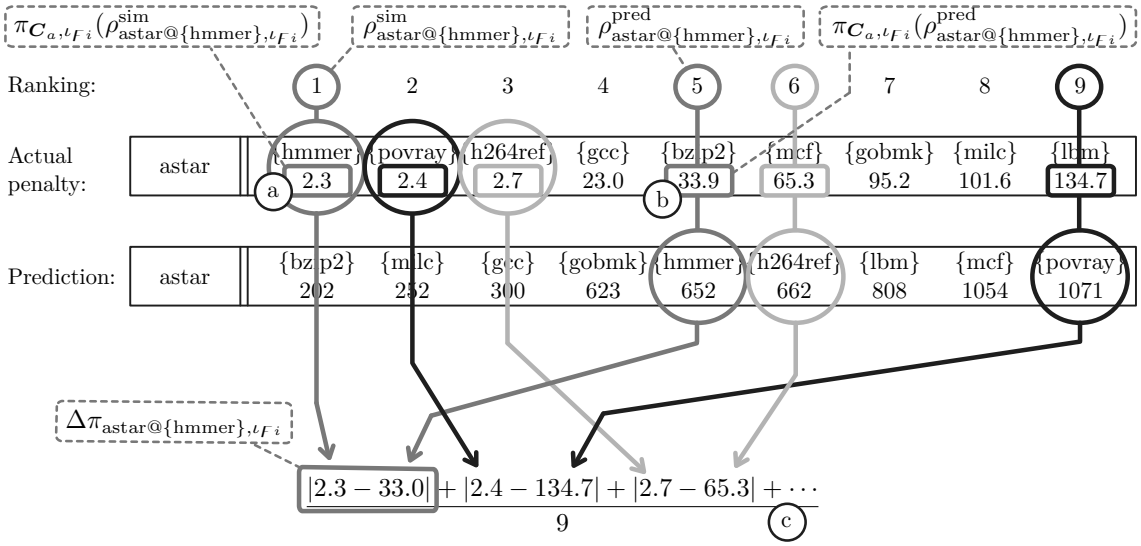


Figure 29: Principal idea of the MP (Mean Penalty) method applied to evaluate cache contention prediction techniques.

As you can see from figure 29, I calculate, for each candidate co-schedule $C_a \in \mathbf{C}_a$, the difference between *actual* penalty $\pi_{C_a,\iota_{\boldsymbol{F}}i}$, e.g. (a), and the penalty at predicted ranking position $\rho^{\text{pred}}_{C_a,\iota_{\boldsymbol{F}}i}$, e.g. (b). Then, differences are averaged (c).

Let $\boldsymbol{\pi}_{\mathbf{C}_{a,\iota_{\boldsymbol{F}}i}} = (\pi_{C_{a,1},\iota_{\boldsymbol{F}}i}, \pi_{C_{a,2},\iota_{\boldsymbol{F}}i}, \dots)$ be the tuple of sorted penalties as presented in figure 17, part (D) and part '(D) Generation of Ground Truth Reference' of section 3.1; let $\boldsymbol{\pi}_{\mathbf{C}_{a,\iota_{\boldsymbol{F}}i}}(\rho)$ be the operation that returns penalty $\pi_{C_{a,j},\iota_{\boldsymbol{F}}i}$ stored at ranking position ρ. Given $\boldsymbol{\pi}_{\mathbf{C}_{a,\iota_{\boldsymbol{F}}i}}$, $\rho^{\text{pred}}_{C_a,\iota_{\boldsymbol{F}}i}$, and $\rho^{\text{sim}}_{C_a,\iota_{\boldsymbol{F}}i}$, I calculate penalty differences

$$\Delta\pi_{C_a,\iota_{\boldsymbol{F}}i} = \left|\boldsymbol{\pi}_{\mathbf{C}_{a,\iota_{\boldsymbol{F}}i}}(\rho^{\text{sim}}_{C_a,\iota_{\boldsymbol{F}}i}) - \boldsymbol{\pi}_{\mathbf{C}_{a,\iota_{\boldsymbol{F}}i}}(\rho^{\text{pred}}_{C_a,\iota_{\boldsymbol{F}}i})\right| \tag{77}$$

for all $a \in A$, all $\psi \in \boldsymbol{\psi}$, all $C_a \in \mathbf{C}^{\psi}_a = \mathfrak{C}(A\backslash\{a\}, \psi - 1)$, all $\boldsymbol{F} \in \boldsymbol{F}$, and all $\iota_{\boldsymbol{F}i} \in \iota_{\boldsymbol{F}}$.

Then, I calculate mean penalty $MP(\psi, \digamma)$ by

$$MP(\psi, \digamma) = \frac{1}{\digamma} \frac{1}{|\iota_{\digamma}|} \sum_{\iota_{\digamma} i \in \iota_{\digamma}} \frac{1}{|A|} \cdot \sum_{a \in A} \frac{1}{|\mathbf{C}_a^{\psi}|} \sum_{C_a \in \mathbf{C}_a^{\psi}} \Delta\pi_{C_a, \iota_{\digamma} i}. \tag{78}$$

Note that I apply $\frac{1}{\digamma}$ in order to normalize MP to 'penalty per instruction'. This way, values of $MP(\psi, \digamma)$ can be plotted side by side in the same diagram for all $\digamma \in \boldsymbol{\digamma}$ without suffering a reduced resolution.

Besides the idea that MP represents time and NMRD represents ranking positions, there is a further aspect to evaluate prediction performance by mean penalty MP: In the NMRD method, ranking positions are evaluated as if they were equipollent, since the NMRD method only evaluates differences in ranking positions, what actually is a good measure to get an idea of the average performance of a prediction method. However, as you can see from figure 29, ranking positions relate to penalties that heavily differ from each other: If a prediction method, for example, gets all but the *first* two ranking positions in figure 29 right, the overall penalty would be $|2.3 - 2.4| + |2.4 - 2.3| = 0.2\ \mu s$. If, however, all but the *last* two ranking positions were predicted correctly, overall penalty would be $|101.6 - 134.7| + |134.7 - 101.6| = 66.2\ \mu s$.

Figure 31 presents my evaluation results regarding *MP* performance for the methods introduced in chapter 2, measured in picoseconds per instruction (ps/instr.). Comparing MP performance to NMRD performance (figure 30), you will realize that evaluation results regarding MP performance are very similar to the already discussed NMRD performance results; therefore, the same obervations and findings that have been discussed in the previous section apply for this section as well. Relative performance *difference* between well and poor performing methods, however, seems to be a bit larger for MP based evaluation than for NMRD based evaluation.

Also when applying the time based MP evaluation function, miss based cache contention prediction methods perform superior compared to methods that rely on stand-alone cache hits or the total number of memory references as predictor. The Prob method turns out to be the most accurate prediction method for larger values of ψ.

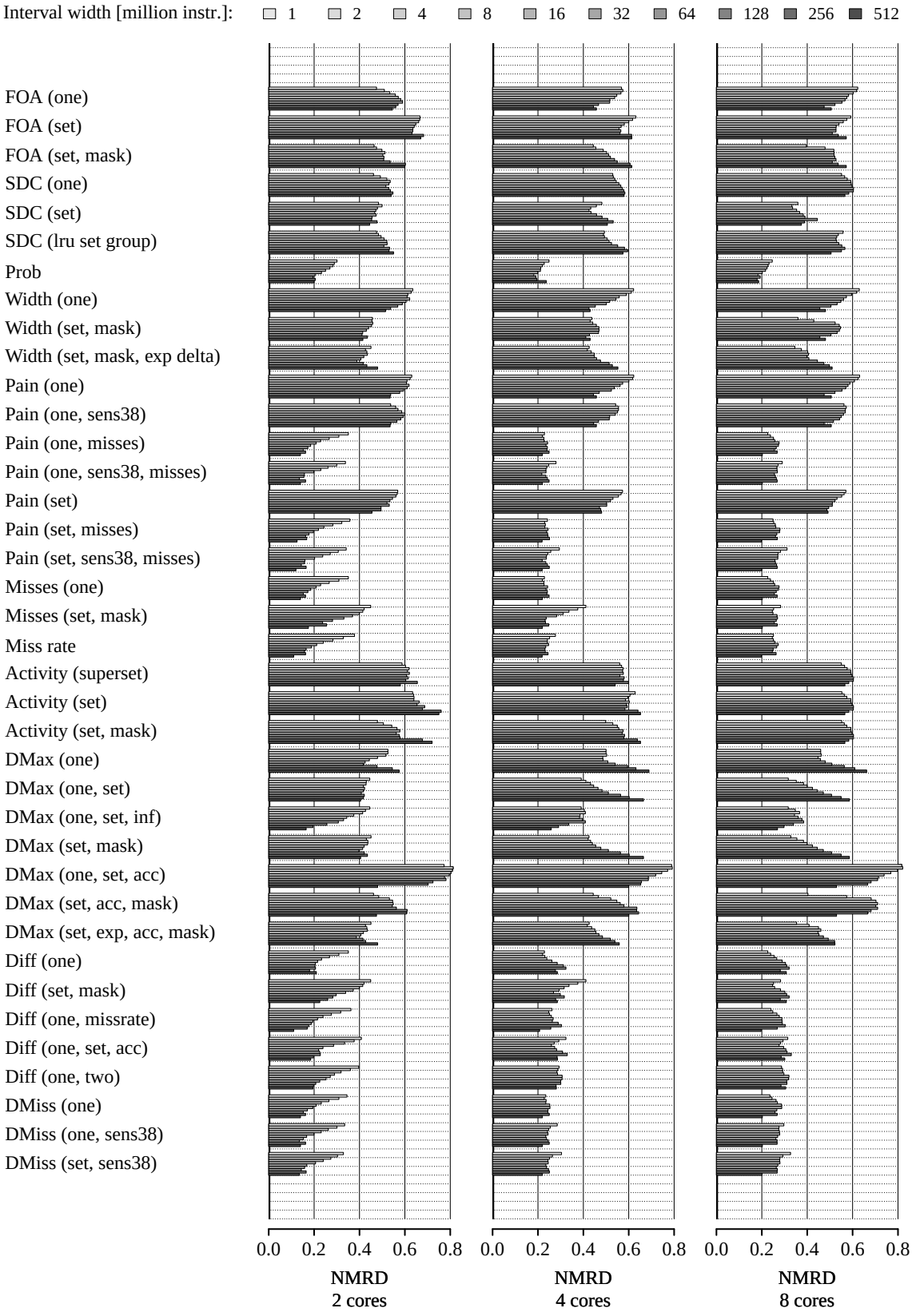


Figure 30: *NMRD* performance of cache contention prediction methods; figure is identical to figure 24.

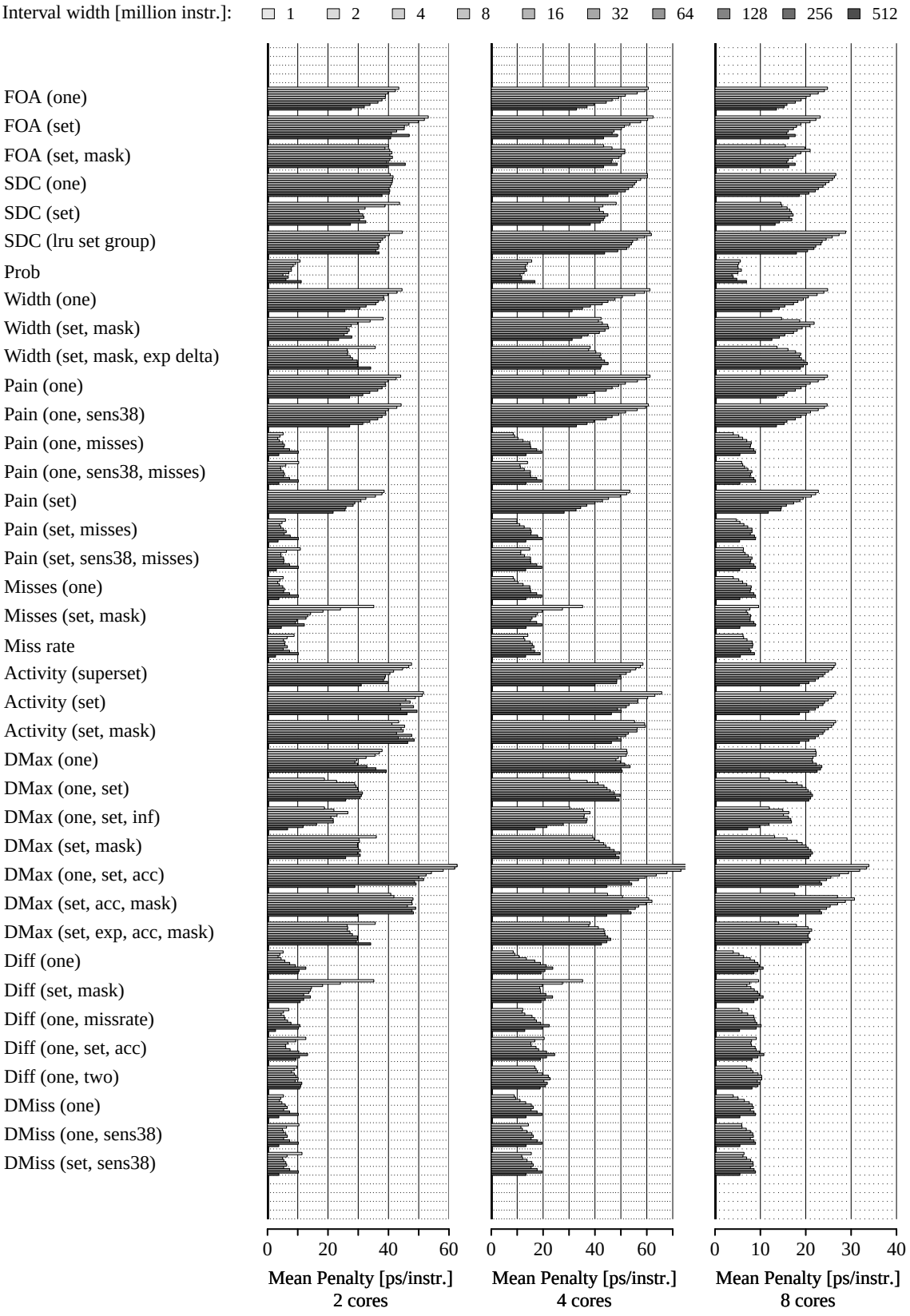


Figure 31: *MP* performance of cache contention prediction methods for $\psi \in \boldsymbol{\psi} = \{2, 4, 8\}$ processor cores and interval sizes $\digamma \in \boldsymbol{\digamma} = \{2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}, 2^{25}, 2^{26}, 2^{27}, 2^{28}, 2^{29}\}$ instructions, averaged over all intervals $\iota_{\digamma i} \in \iota_{\digamma}$ and all applications $a \in A = \{$*astar*, *bzip2*, *gcc*, *gobmk*, *h264ref*, *hmmer*, *lbm*, *mcf*, *milc* and *povray*$\}$.

Big Picture

Most state-of-the-art cache contention prediction methods rely on memory reuse patterns, i.e. the distribution of elements in a stack distance histogram, to perform cache contention prediction. In most cases, state-of-the-art prediction method apply (the distribution of) elements $1 \ldots$ associativity α, i.e. the stand-alone cache *hits*, or the sum of all elements $1 \ldots \alpha + 1$, i.e. the *total number of cache accesses*, to perform a prediction. Single element $\alpha + 1$ of stack distance histograms, i.e. the number of cache *misses*, is rarely used. The motivation is that "if a thread hardly ever reuses its cached data – as would be the case with a video-streaming application that touches that data only once – it will not suffer from contention even if it brings lots of data into the cache" [Fedorova et al., 2010]. This means that the number of cache misses has not been regarded as a measure for cache contention, as a stand-alone cache miss cannot turn into a contention miss: It already is a miss. It has been argued that only stand-alone cache *hits* can turn into contention misses; therefore, it is primarily stand-alone cache hits that have been applied as a measure of cache contention in the past. [Chandra et al., 2005, Fedorova et al., 2010]

But there is one point that has generally been ignored: Stack distance histograms are primarily dominated by entries $H^{sd}_{a,\iota_{F\,i}}(1)$, and $H^{sd}_{a,\iota_{F\,i}}(1) \approx \sum_{\delta=1}^{\alpha} H^{sd}_{a,\iota_{F\,i}}(\delta) \approx \sum_{\delta=1}^{\alpha+1} H^{sd}_{a,\iota_{F\,i}}(\delta)$ for most applications a (cf. section 'Stack Distance Histograms' in the appendix). Therefore, state-of-the-art methods primarily rely on $H^{sd}_{a,\iota_{F\,i}}(1)$ to perform their prediction. But this is the worst choice they actually can make: Given an application a and a set of co-scheduled applications C_a, then those methods primarily base on estimating the probability to make entries $H^{sd}_{a,\iota_{F\,i}}(1)$ become misses; and they estimate it by entries $H^{sd}_{a',\iota_{F\,i}}(1)$ for all $a' \in C_a$. In order to turn an entry $H^{sd}_{a,\iota_{F\,i}}(1)$ into a miss, however, the applications in C_a have to perform many accesses *to different cache lines* of the same cache set; this is unlikely, however, as applications primarily access cache lines with $\delta = 1$, i.e. they nearly always reference the cache line that has just been referenced before; and references to such lines are generally hits, and are therefore not suddenly pushed onto the LRU stack to displace an already fetched entry. But it even gets worse: In order to replace an LRU stack entry of a, the accesses of the applications in C_a also have to be performed *between two references of a to the same cache line*. This means that the probability to replace a

cache line of a from the LRU stack is high, if there is much time between two references of a to the same cache line, and low, if a references the same cache line with every memory access. However, $H^{sd}_{a,\iota_{F}i}(1) \approx \sum_{\delta=1}^{\alpha} H^{sd}_{a,\iota_{F}i}(\delta) \approx \sum_{\delta=1}^{\alpha+1} H^{sd}_{a,\iota_{F}i}(\delta)$ directly implies the last point: If almost all accesses of a are performed with distance $\delta = 1$, then there is actually no time for other applications to replace those entries.

Further, $H^{sd}_{a',\iota_{F}i}(1)$, $\sum_{\delta=1}^{\alpha} H^{sd}_{a',\iota_{F}i}(\delta)$ and $\sum_{\delta=1}^{\alpha+1} H^{sd}_{a',\iota_{F}i}(\delta)$ all have similar values for all $a' \in A$: They differ by a maximum factor of 2. Compared to a factor of up to about 1300 that can be observed regarding the number of stand-alone cache misses, i.e. $H^{sd}_{a',\iota_{F}i}(\alpha+1)$, $a' \in A$, there seems to be no significant variance in the number of stand-alone cache hits or cache accesses.

A much better predictor for cache contention is the number of *cache misses*, i.e. stack distance entries with $\delta = \alpha + 1$. First, there is much variance in these data, regarding the applications $a' \in A$. Secondly, each reference to an entry in $H^{sd}_{a,\iota_{F}i}$ definitely pushes a new address key onto the LRU stack. This means that there will definitely be a displacement, if the LRU stack is already full (very likely for high values of F and ψ).

Although stand-alone cache misses cannot be turned into contention misses, they cause stand-alone hits of other applications to turn into contention misses.

This makes the Pain method also a good choice, if you apply misses rather than cache accesses as intensity measure; further, applying stack distance histogram entries $3 \ldots \alpha$ rather than $1 \ldots \alpha$ as sensitivity measure can slightly increase performance.

The Prob method shows best NMRD performance in many cases. But it is quite complex, as you can see from the timing and the cost-gain analysis performed in the following sections.

In the past, stand-alone cache misses have gained only limited attention predicting cache contention. In [Knauerhase et al., 2008], the autors focused on improving operating system scheduling decisions in order to reduce interference of processes that share a last level cache. They applied the number of last level cache misses per cycle as a predictor for cache contention. Note that the applied misses per cycle were not stand-alone misses, but misses measured with performance counters when already *sharing* the last level cache. To predict cache contention for an interval ι_{Fi}, Knauerhase et al. applied cache misses that were measured under *cache sharing* in interval ι_{Fi-1}.

Fedorova et al. refer to Knauerhase et al.'s work and investigate cache contention prediction applying cache miss rates as predictor; they discover that "cache-miss rate turned out to be an excellent predictor for contention for the memory controller, prefetching hardware and the front-side bus" [Fedorova et al., 2010].
Fedorova et al. find the good performance of the misses method very surprising: "Who could have imagined that the best way to approximate the *Pain* metric would be to use the LLC miss rate?" [Fedorova et al., 2010]
The high concentration of stack distance histograms, however, i.e. the dominating effect of $H^{sd}_{a,\iota_{\mathcal{F}}i}(1)$ can explain this observation.

3.3 Best-Selection Performance

In this section, I present two evaluation functions (cf. (13) in figure 17) to evaluate prediction methods regarding their ability to select the co-schedule C_a from a given set of candidate co-schedules $\mathbf{C}_a = \{C_{a,1}, C_{a,2}, \dots\}$ that minimizes cache contention for a given application a.

PPBAB - Penalty Predicted Best vs. Actual Best

The *PPBAB* (Penalty Predicted Best vs. Actual Best) evaluation function determines the amount of time cache contention penalty of the *actual* best co-schedule differs from the penalty of a co-schedule that is *predicted* to achieve the least penalty, as it is exemplarily shown in figure 32.
Given an application a, the set of all candidate co-schedules of a for a given number of processor cores ψ, i.e. $\mathbf{C}^{\psi}_a = \mathfrak{C}(A\backslash\{a\}, \psi - 1)$, and an interval size $\mathcal{F}$, I calculate penalty difference $\Delta\pi^{\text{PPBAB}}_{\mathbf{C}_a,\iota_{\mathcal{F}}i}$ for interval $\iota_{\mathcal{F}i}$ according to

$$\Delta\pi^{\text{PPBAB}}_{\mathbf{C}^{\psi}_a,\iota_{\mathcal{F}}i} = \Big|\boldsymbol{\pi}_{\mathbf{C}^{\psi}_a,\iota_{\mathcal{F}}i}(1) - \sum_{C_a\in\mathbf{C}^{\psi}_a \,\big|\, \rho^{\text{pred}}_{C_a,\iota_{\mathcal{F}}i}=1} \boldsymbol{\pi}_{\mathbf{C}^{\psi}_a,\iota_{\mathcal{F}}i}(\rho^{\text{sim}}_{C_a,\iota_{\mathcal{F}}i})\Big|. \tag{79}$$

Note that I apply the sum in equation 79 just to select the candidate co-schedule that is predicted to minimize cache contention; there is no addition performed.
Given $\Delta\pi^{\text{PPBAB}}_{\mathbf{C}_a,\iota_{\mathcal{F}}i}$, a number of processor cores ψ, and an interval size $\mathcal{F}$, I calculate

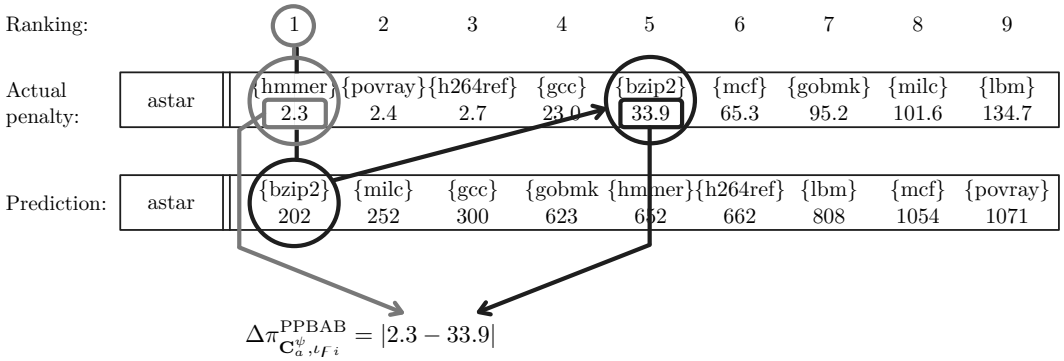


Figure 32: Calculating penalty difference $\Delta\pi_{\mathbf{C}^{\psi}_{a},\iota_{F}i}$ for the actual and the predicted best co-schedule.

$PPBAB(\psi, F)$ by

$$PPBAB(\psi, F) = \frac{1}{F} \cdot \frac{1}{|\iota_F|} \sum_{\iota_F i \in \iota_F} \frac{1}{|A|} \cdot \sum_{a \in A} \Delta\pi^{\text{PPBAB}}_{\mathbf{C}^{\psi}_{a},\iota_F i}. \tag{80}$$

As I did in the last section, I normalize PPBAB values to 'penalty per instruction'.

In figure 34, I present my evaluation regarding the PPBAB evaluation function. If you compare PPBAB results in figure 34 (logarithmic scale) to the MP results presented in figure 33 (linear scale), you will observe that an evaluation of the methods *in relation to one another* yields about the same results for the PPBAB evaluation function as it is the case for the MP evaluation function. There are, however, some minor differences, such as FOA (set) performs slightly better than FOA (one) for some F when $\psi = 2$. But such minor results are not discussed any further.

Note that the *range* of results for PPBAB is much larger than for MP, as MP results are averaged over all candidate co-schedules, while PPBAB results are not.

As it has been the case with the MP evaluation function, miss based methods perform far better than hit or access based prediction methods due to the high concentration of stack distance histograms (cf. section 'Stack Distance Histograms' in the appendix).

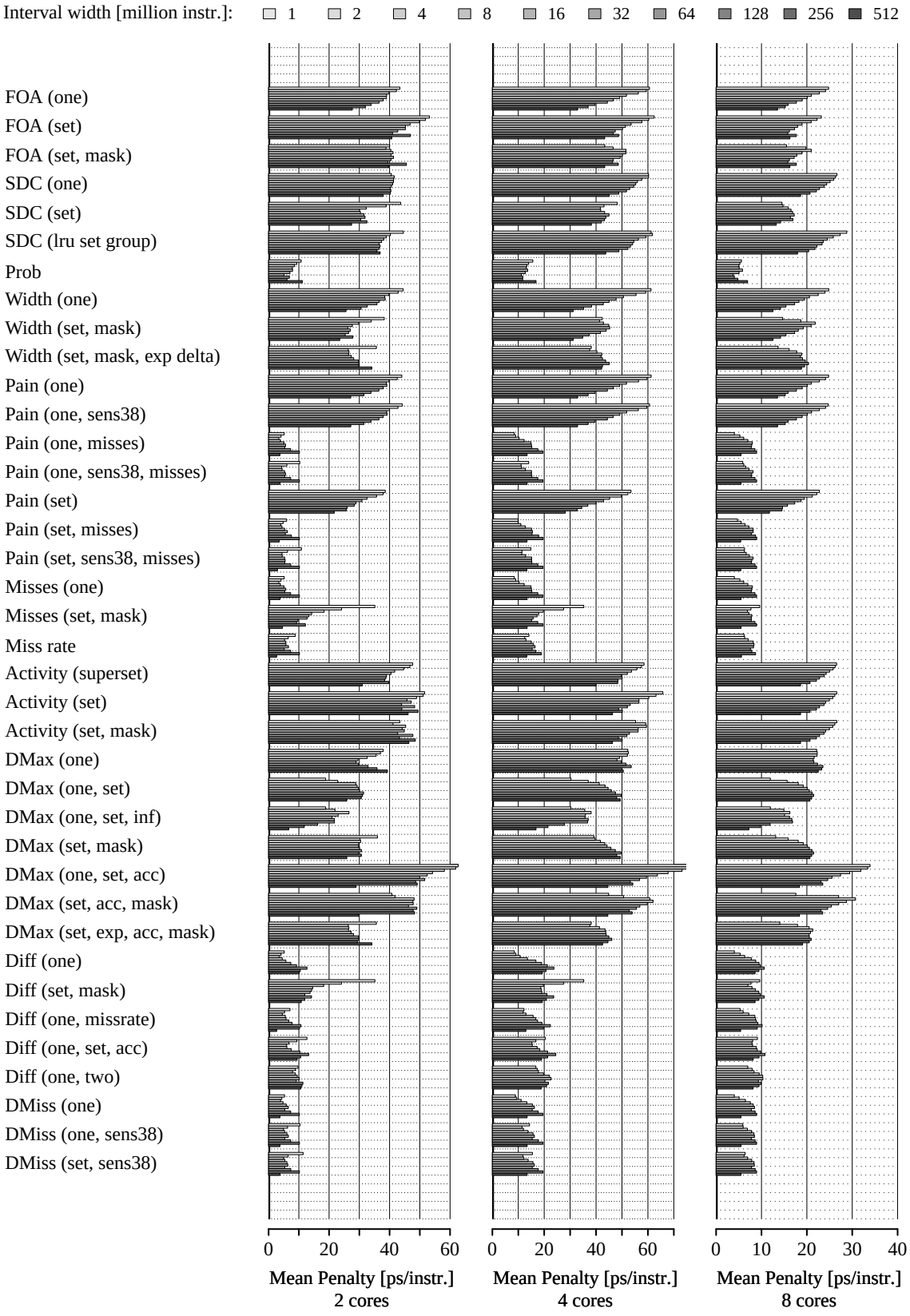


Figure 33: *MP* performance of cache contention prediction methods; figure is identical to figure 31.

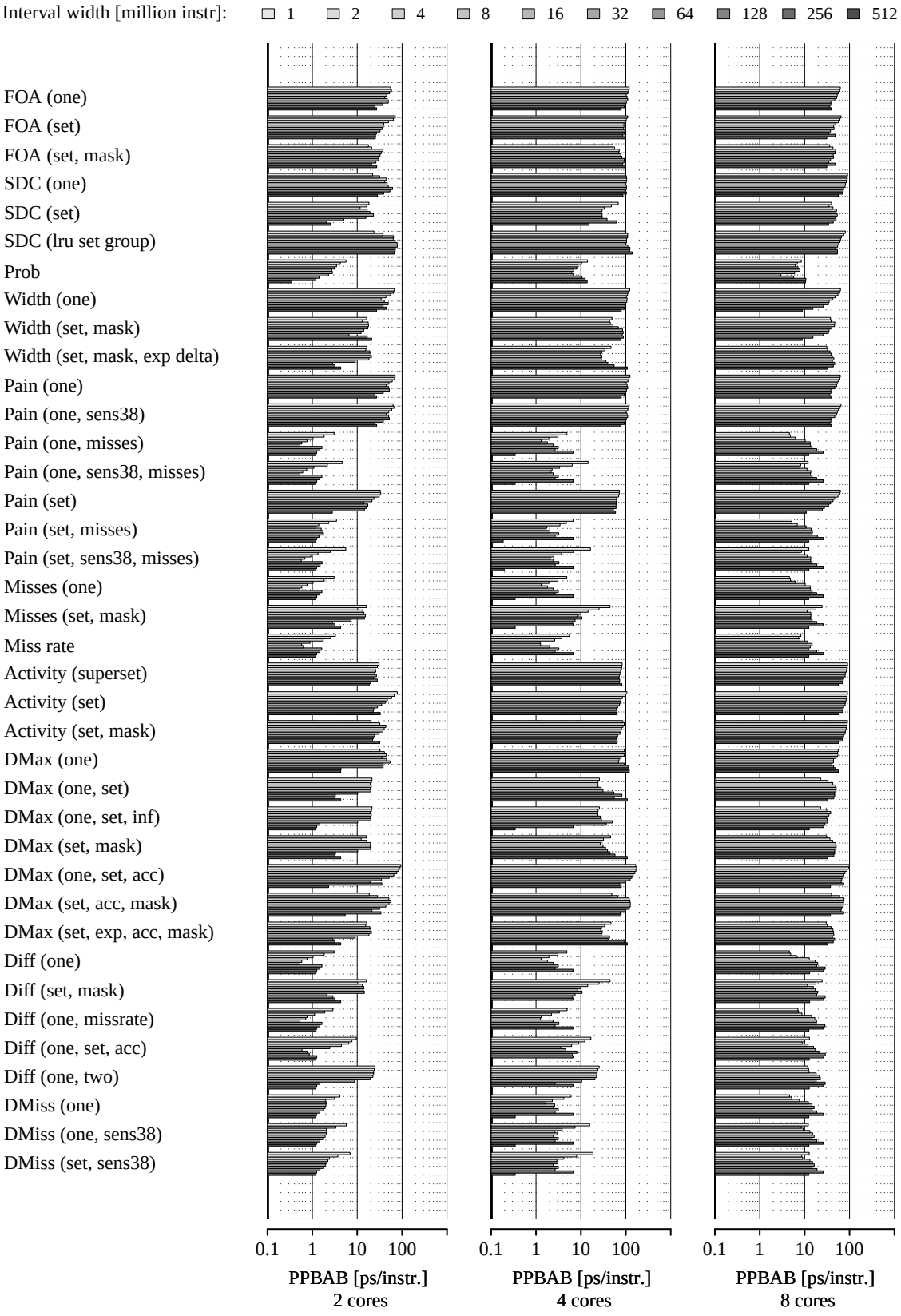


Figure 34: Evaluation of cache contention prediction methods applying the $PPBAB(\psi, F)$ evaluation function for $\psi \in \boldsymbol{\psi}$ processor cores and interval sizes $F \in \boldsymbol{F}$; results are averaged over all intervals $\iota_{Fi} \in \iota_F$ and all applications $a \in A$. Note the logarithmic scale.

PPBRS - Penalty Predicted Best vs. Random Selection (Gain)

With the PPBRS (Penalty Predicted Best vs. Random Selection) evaluation function, I determine the performance *gain* (units of time) a cache contention prediction method can achieve: I compare the penalty of the co-schedule that a prediction technique selects as best co-schedule to the penalty achieved on average when chosing candidate co-schedules randomly. Although state-of-the-art schedulers operate according to specific rules and characteristics, I assume them to co-schedule applications randomly *with respect to cache contention.*

Figure 35 exemplarily shows the way the PPBRS method determines gain $\Delta\pi^{\text{PPBRS}}_{\mathbf{C}^{\psi}_{a,\iota_{F}i}}$:

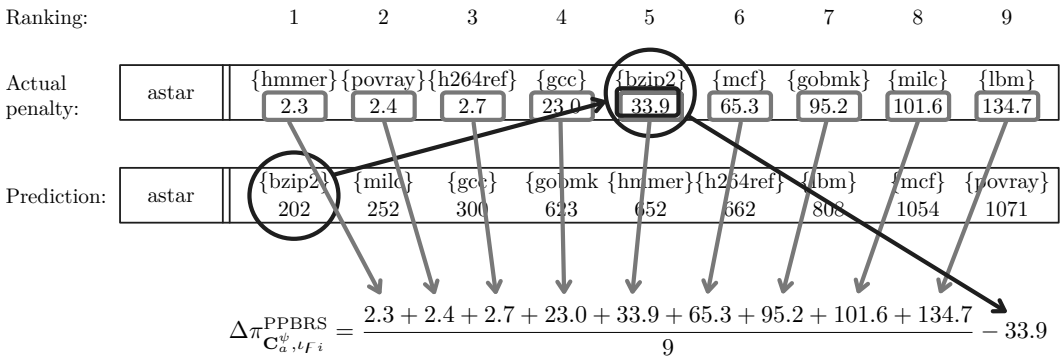


Figure 35: Calculation of gain, i.e. the difference between expected penalty and the penalty of the candidate co-schedule that is predicted to be the best co-schedule.

Given an application a, the set of all candidate co-schedules of a for a given number of processor cores ψ, i.e. $\mathbf{C}^{\psi}_{a} = \mathfrak{C}(A\backslash\{a\}, \psi - 1)$, and an interval size F, I calculate penalty difference $\Delta\pi^{\text{PPBRS}}_{\mathbf{C}^{\psi}_{a},\iota_{F}i}$ for interval $\iota_{F}i$ by

$$\Delta\pi^{\text{PPBRS}}_{\mathbf{C}^{\psi}_{a},\iota_{F}i} = \frac{1}{|\mathbf{C}^{\psi}_{a}|}\sum_{C_a\in\mathbf{C}^{\psi}_{a}} \boldsymbol{\pi}_{\mathbf{C}^{\psi}_{a},\iota_{F}i}(\rho^{\text{sim}}_{C_a,\iota_{F}i}) \quad - \sum_{C_a\in\mathbf{C}^{\psi}_{a}\left|\rho^{\text{pred}}_{C_a,\iota_{F}i}=1\right.} \boldsymbol{\pi}_{\mathbf{C}^{\psi}_{a},\iota_{F}i}(\rho^{\text{sim}}_{C_a,\iota_{F}i}). \tag{81}$$

Given $\Delta\pi^{\text{PPBRS}}_{\mathbf{C}_{a,\iota_{F}i}}$, I calculate $PPBRS(\psi, F)$ by

$$PPBRS(\psi, F) = \frac{1}{F}\cdot\frac{1}{|\iota_{F}|}\sum_{\iota_{F}i\in\iota_{F}}\frac{1}{|A|}\sum_{a\in A}\Delta\pi^{\text{PPBRS}}_{\mathbf{C}^{\psi}_{a},\iota_{F}i}. \tag{82}$$

Figure 36 shows evaluation results for the PPBRS evaluation function. A positive value indicates that the corresponding prediction method performs better than selecting a co-schedule randomly; a negative value indicates that selecting co-schedules randomly achieves better results than applying the prediction method.

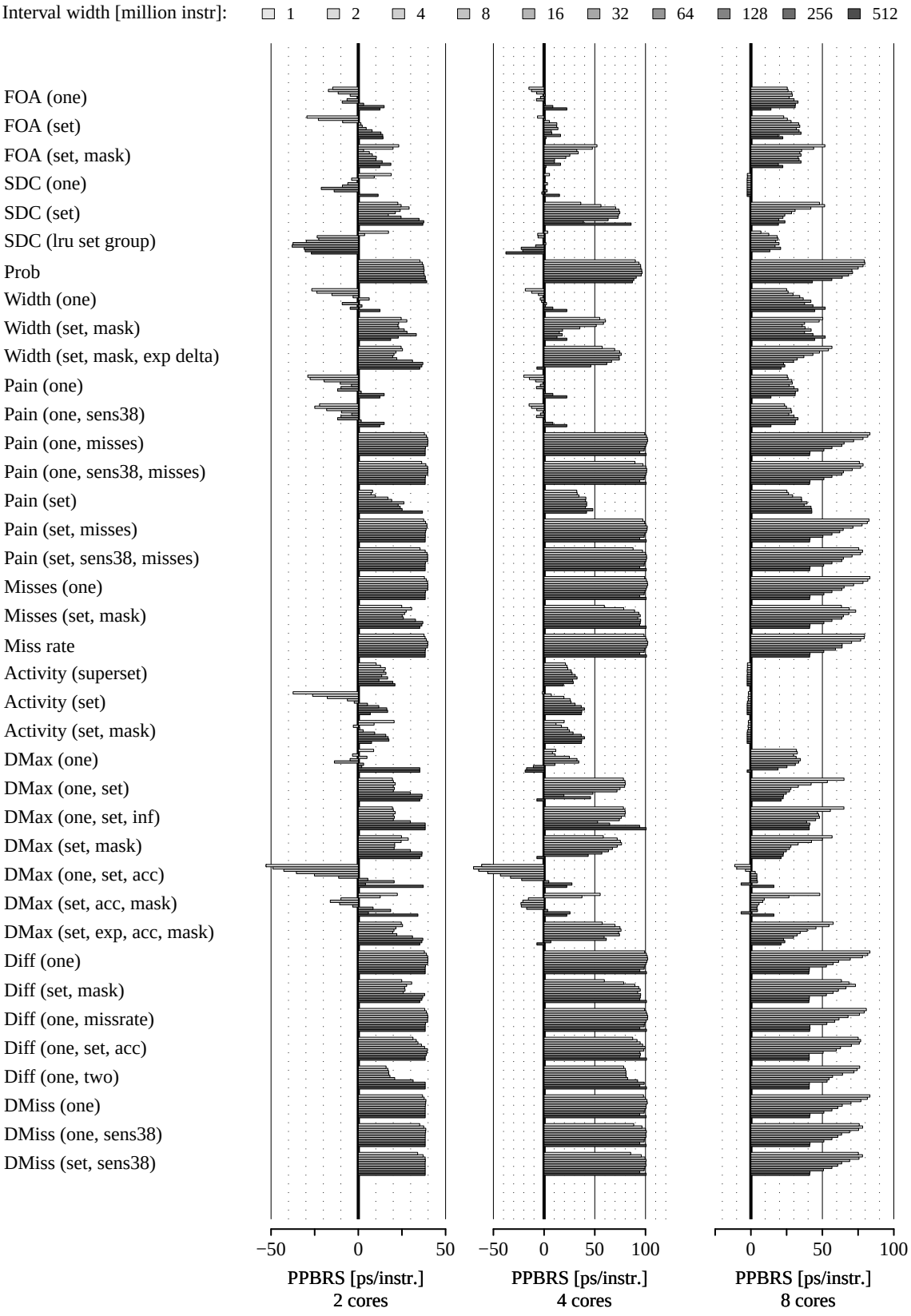


Figure 36: Evaluation of cache contention prediction techniques applying the $PPBRS(\psi, F)$ evaluation function for $\psi \in \boldsymbol{\psi}$ processor cores and interval sizes $F \in \boldsymbol{F}$; results are averaged over all intervals $\iota_{Fi} \in \iota_F$ and all applications $a \in A$.

3.4 Timing Performance (Cost)

Besides accuracy, *prediction time* is a fundamental quality measure for prediction methods and has to be considered in the evaluation: Applying cache contention prediction techniques in order to benefit from better co-schedules, even the best accuracy will be worthless if the time to perform a prediction (i.e. cost) is not acceptable.

In section 3.1, part 'Ⓒ Calculation of Prediction Rankings', I already presented the way I determine $\tau_{C_a}^{\text{user}}$, $\tau_{C_a}^{\text{syst}}$ and $\tau_{C_a}^{\text{elap}}$ for all intervals $\iota_{Fi} \in \iota_F$ (cf. algorithm 12). Note that each τ_{C_a} represents timing information for all $|\iota_F|$ intervals, i.e. 2^{29} instructions.

I determine execution times for each $F \in \boldsymbol{F}$ and each $\psi \in \boldsymbol{\psi}$, as it is exemplarily shown in figure 37 for *FOA (one)* and the *Prob* method. See [Zwick, 2011] for the execution times of all methods presented in this thesis. In figure 37, 'U' depicts user time $\tau^{\text{user}}(\psi, F)$, 'S' depicts system time $\tau^{\text{syst}}(\psi, F)$, and 'E' represents elapsed time $\tau^{\text{elap}}(\psi, F)$. Note that the presented values are averaged over all applications $a \in A$, $C_a \in \mathbf{C}_a^{\psi} = \mathfrak{C}(A \backslash \{a\}, \psi - 1)$ and all intervals $\iota_{Fi} \in \iota_F$. Hereby, τ is calculated from τ_{C_a} as it is exemplarily shown for τ^{elap}:

$$\tau^{\text{elap}}(\psi, F) = \frac{1}{F} \cdot \frac{1}{|A|} \cdot \sum_{a \in A} \frac{1}{|\mathbf{C}_a^{\psi}|} \sum_{C_a \in \mathbf{C}_a^{\psi}} \tau_{C_a}^{\text{elap}} \tag{83}$$

User time $\tau^{\text{user}}(\psi, F)$ and system time $\tau^{\text{syst}}(\psi, F)$ calculate accordingly.

Figure 37 shows that, besides some measuring inaccuracies, higher values of window size F imply a higher execution time; for lower values of F, execution time is nearly constant. The reason for this behavior is as follows:

Predictions are performed *per interval* ι_{Fi} (see algorithm 12). For $F = 2^{29}$ (column '512 million instructions' in figure 37), there is only 1 set of predictors that has to be fetched from disk. Although most predictors are only some byte in size, the operating system has to fetch a full 4 k memory page from disk. The page then resides in memory and can be accessed much faster for subsequent references; however, in case $F = 2^{29}$, it is only accessed once, as only $|\iota_F| = 1$ prediction is performed.

For $F = 2^{20}$ (column '1 million instructions' in figure 37), however, a single page fetch reads many predictors at the same time and makes them reside in main memory or even in the processor cache. As a consequence, predictors of subsequent intervals ι_{Fi}, $i > 1$, can

be accessed much faster; note that in case $F = 2^{20}$, $|\iota_F| = 512$ predictions are performed. If predictor size is small enough to make all $|\iota_F|$ predictors fit a single page, then

- $F = 1 \Rightarrow$ average predictor access time is composed of $\frac{1}{512}$ parts disk access time and $\frac{511}{512}$ parts memory/cache access time, while
- $F = 512 \Rightarrow$ average predictor access time is composed of 1 disk access time only. □

	Type	Window size [million instructions]									
	[unit]	1	2	4	8	16	32	64	128	256	512
FOA (one, 2 cores)	U [μs]	5.95	5.96	6.02	6.01	6.23	6.47	7.09	8.31	10.6	14.3
	S [μs]	10.4	10.3	10.5	10.1	10.7	11.6	13.8	18.0	26.1	43.1
	E [μs]	16.9	16.9	17.3	17.0	19.4	23.7	30.8	47.3	77.7	125
FOA (one, 4 cores)	U [μs]	11.8	11.8	11.9	11.8	11.9	12.1	12.7	13.5	15.2	18.2
	S [μs]	21.0	20.7	20.7	19.8	20.0	20.9	22.7	26.0	32.9	45.4
	E [μs]	33.1	32.8	32.9	31.8	32.1	33.6	36.9	40.1	49.1	65.2
FOA (one, 8 cores)	U [μs]	23.5	23.5	23.5	23.4	23.5	23.8	24.3	25.4	27.3	30.7
	S [μs]	41.8	41.5	41.3	39.5	39.7	40.9	43.5	48.5	58.0	77.4
	E [μs]	65.5	65.6	65.6	63.2	63.4	65.2	68.8	74.7	86.8	110
Prob (one, 2 cores)	U [μs]	30.8	30.8	30.8	30.7	30.8	31.0	31.5	32.7	34.8	38.5
	S [μs]	16.2	16.3	16.4	15.4	15.7	16.5	18.3	22.8	30.5	46.7
	E [μs]	47.4	47.6	47.7	46.4	46.7	47.8	50.1	56.5	66.1	86.5
Prob (one, 4 cores)	U [μs]	82.7	82.7	82.6	82.5	82.5	82.9	83.3	84.3	86.5	90.2
	S [μs]	32.9	32.7	32.5	30.9	31.1	32.2	34.9	40.1	50.5	70.5
	E [μs]	119	118	117	116	115	116	120	126	140	162
Prob (one, 8 cores)	U [μs]	187	187	187	186	186	187	187	189	192	196
	S [μs]	65.1	64.9	64.4	61.3	62.0	62.8	66.5	73.0	87.9	115
	E [μs]	255	254	253	252	253	252	259	264	284	318

Figure 37: Execution times (excerpt).

In order to

- make execution time reflect predictor access time linearly regarding its size (and not quantized to 4 k, i.e. page size, steps),
- avoid the time necessary for calculations to be masked out by the time waiting for disk fetches, but still
- incorporate both *predictor access time* and the *time to calculate predictions from predictors* in the evaluation,

I choose $\tau^{\text{elap}}(\psi, F)$ with $F = 1$ to represent the time a specific prediction methods applies to perform *one* prediction. Figure 38 summarizes the results. Note that the *time to create predictors* is *not included*, as it can be calculated *offline* in advance of the prediction. Figure 38 shows that the fastest methods take about 8 μs per prediction in case $\psi = 2$. Note that predictions that are performed per-cache-set (variations 'set') do not take $|S|$ times as long as variations that summarize all cache sets in a single predictor (variations 'one'); this is caused by paging and caching effects (see above). Therefore, computationally more intensive methods have a less favorable timing ratio of variations 'set' to 'one', e.g.

- $\tau^{\text{elap}}_{\text{SDC (set)}}(\psi = 2, F = 1) \ : \ \tau^{\text{elap}}_{\text{SDC (one)}}(\psi = 2, F = 1) \ \approx 121$
- $\tau^{\text{elap}}_{\text{DMiss (one, sens38)}}(\psi = 2, F = 1) \ : \ \tau^{\text{elap}}_{\text{DMiss (set, sens38)}}(\psi = 2, F = 1) \ \approx 8.5.$

Note that miss based methods such as *Misses (one)* do not only provide accurate prediction results, but also perform their predictions in a minimum amount of time. Contrarily, the *Prob* method, which showed, for many cases of ψ and F, even better prediction accuracy than miss based methods, takes much more time to perform a prediction.

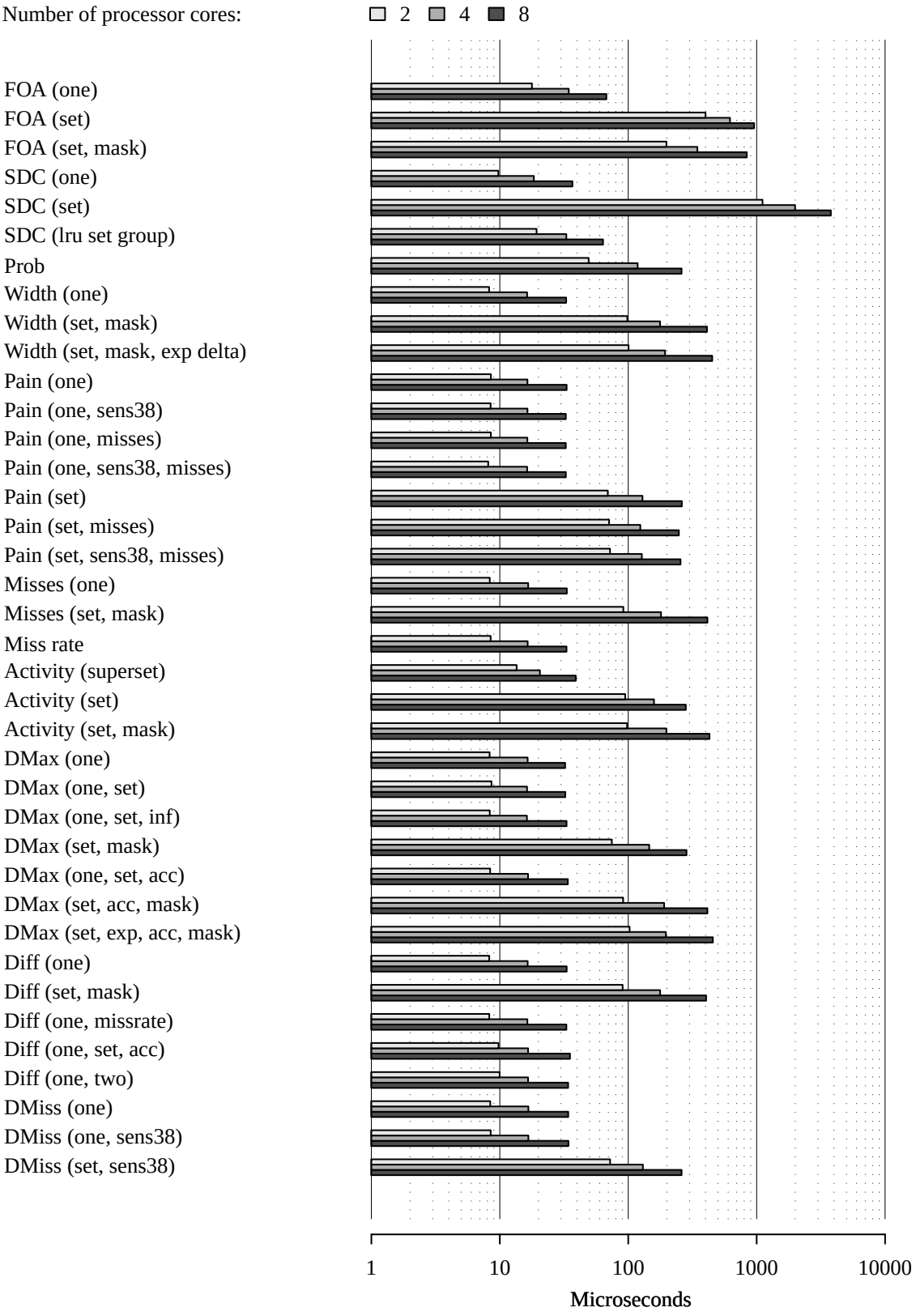


Figure 38: Elapsed time to calculate predictions.

3.5 Gain vs. Cost Analysis

There might be situations where gain must outweigh cost in order to make the application of cache contention prediction techniques reasonable. In other situations, there might be less rigid requirements, for example, if there is a lot of computational performance available, but the memory system is heavily overloaded.

In this section, I perform a gain vs. cost analysis by calculating the ratio of PPBRS gain to cost $\tau^{\text{elap}}(\psi, F = 1)$ according to

$$\mathcal{G}(\psi, F) = \frac{PPBRS(\psi, F) \cdot F}{\tau^{\text{elap}}(\psi, 1)}. \tag{84}$$

Note that multiplicative term F is introduced to compensate for $\frac{1}{F}$ in equation 82. Figure 39 presents the gain vs. cost analysis. Values < 0.1 are cut off. Note that the figure lets you estimate how many predictions can be performed until $\mathcal{G}(\psi, F)$ gets ≤ 1. As an example, for $\psi = 2$, *Misses* (one) allows for 2350 predictions if $F = 2^{29}$ (512 million instructions), but only for 4 predictions in case $F = 2^{20}$ (1 million instructions).

As you can observe from figure 39, miss based methods such as *Misses (one)*, *Miss rate*, *Pain (.., misses)* as well as many variations of the *Diff* and *DMiss* methods show best gain vs. cost ratio. The *Prob* method that achieved best prediction accuracy for many ψ and F suffers from its complexity.

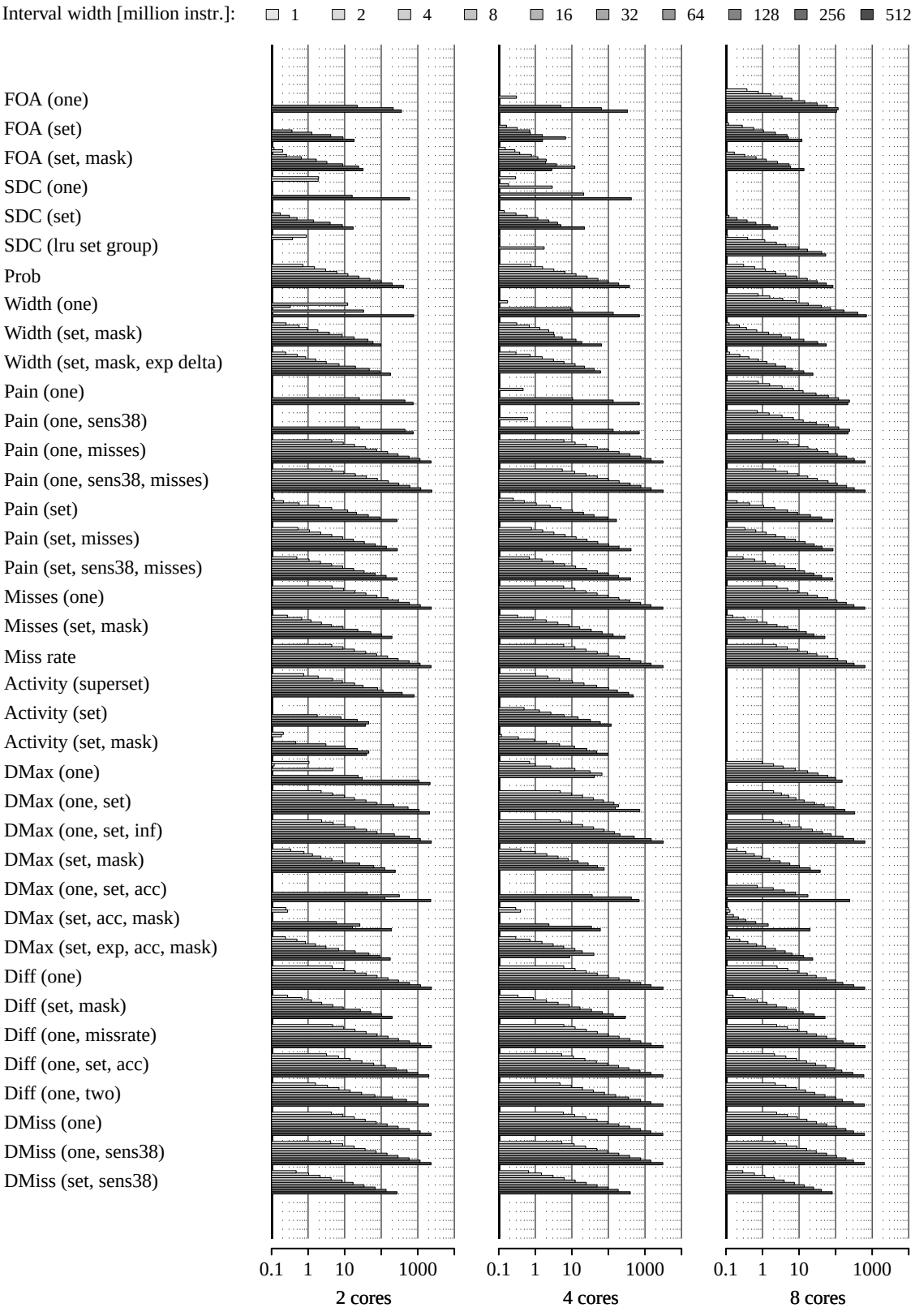


Figure 39: Gain vs. cost analysis.

4 Conclusion

In this thesis, I evaluated cache contention prediction methods according to prediction *accuracy*, the *time* necessary to perform a prediction, and a *gain vs. cost* analysis. Applying a stack distance based notation on cache contention prediction techniques, I showed that

- most state-of-the-art methods primarily rely on the number of cache *accesses*, cache *hits*, or on the *distribution* of references to cache LRU stack positions as predictor. This thesis revealed, however, that
- cache accesses, stand-alone cache hits and even distributions of cache LRU stack reference distances are often inappropriate measures to predict cache contention, as they primarily represent those memory references that are most recently used and are therefore poor candidates to either be displaced from a cache or displace any other cached data (cf. section 3.2, 'Big Picture'). I discovered that
- a much better predictor for cache contention is the amount of stand-alone cache misses. This applies to both the prediction of a candidate co-schedule ranking with respect to the amount of cache contention candidate co-schedules introduces to other applications, as well as to the selection of the co-schedule from a given set of candidate co-schedules that minimizes cache contention. The high prediction accuracy results from the high probability that stand-alone cache misses displace cache LRU stack entries of other applications (cf. section 3.2, 'Big Picture'). Additionally, my evaluation pointed out that
- most miss based methods do not only show superior prediction accuracy, but are also very fast in execution and therefore achieve a good cost vs. gain ratio.

Based on these observations, I further showed that

- prediction on a per-cache-set basis does not achieve significantly better prediction results than a prediction that uses a single predictor for *all* cache sets. Additionally, I showed that
- an enhanced weighting of stack distance histogram entries does not achieve any significant performance improvements.

Bibliography

[Baer and Wang, 1988] Baer, J.-L. and Wang, W.-H. (1988). On the inclusion properties for multi-level cache hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture, 1988*, pages 73–80.

[Chandra et al., 2005] Chandra, D., Guo, F., Kim, S., and Solihin, Y. (2005). Predicting inter-thread cache contention on a chip multi-processor architecture. *Proceedings of the 11th International Symposium on High-Performance Computer Architecture, 2005*, pages 340–351.

[Chen and Aamodt, 2009] Chen, X. E. and Aamodt, T. M. (2009). A first-order fine-grained multithreaded throughput model. In *Proceedings of the 15th Annual IEEE International Symposium on High Performance Computing and Applications, 2009*, pages 329–340.

[El-Moursy et al., 2006] El-Moursy, A., Garg, R., Albonesi, D., and Dwarkadas, S. (2006). Compatible phase co-scheduling on a CMP on multi-threaded processors. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium, 2006*.

[Fedorova et al., 2010] Fedorova, A., Blagodurov, S., and Zhuravlev, S. (2010). Managing contention for shared resources on multicore processors. *Communications of the ACM*, 53(2):49–57.

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

[Hill and Smith, 1989] Hill, M. D. and Smith, A. J. (1989). Evaluating associativity in CPU caches. In *IEEE Transactions on Computers*, volume 38, pages 1612–1630.

[Huffmire and Sherwood, 2006] Huffmire, T. and Sherwood, T. (2006). Wavelet-based phase classification. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques, 2006*, pages 95–104.

[Kihm and Connors, 2004] Kihm, J. L. and Connors, D. A. (2004). Implementation of fine-grained cache monitoring for improved SMT scheduling. In *Proceedings of the IEEE*

International Conference on Computer Design: VLSI in Computers & Processors, 2004, pages 326–331.

[Kihm et al., 2005] Kihm, J. L., Settle, A., Janiszewski, A., and Connors, D. A. (2005). Understanding the impact of inter-thread cache interference on ILP in modern SMT processors. *The Journal of Instruction-Level Parallelism*, 7:1–28.

[Knauerhase et al., 2008] Knauerhase, R., Brett, P., Hohlt, B., Li, T., and Hahn, S. (2008). Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66.

[Luk et al., 2005] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klausner, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005*, pages 190–200.

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

[Olukotun et al., 1996] Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K., and Chang, K. (1996). The case for a single-chip multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 30, pages 2–11.

[Otellini, 2006] Otellini, P. (2006). Keynote at Intel Developer Forum, Sept. 26, 2006.

[Settle et al., 2004] Settle, A., Kihm, J. L., Janiszewski, A., and Connors, D. A. (2004). Architectural support for enhanced SMT job scheduling. In *Proceedings of the 13th International Conference of Parallel Architectures and Compilation Techniques, 2004*, pages 63 – 73.

[Sherwood et al., 2003] Sherwood, T., Perelman, E., Hamerly, G., Sair, S., and Calder, B. (2003). Discovering and exploiting program phases. *IEEE Micro: Micro's Top Picks from Microarchitecture Conferences*, 23(6):84–93.

[Snavely and Tullsen, 2000] Snavely, A. and Tullsen, D. (2000). Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, 2000*, pages 234–244.

[Song et al., 2007] Song, F., Moore, S., and Dongarra, J. (2007). L2 cache modeling for scientific applications on chip multi-processors. In *Proceedings of the 2007 International Conference on Parallel Processing*, pages 51–58.

[Suh et al., 2002] Suh, G., Devadas, S., and Rudolph, L. (2002). A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture, 2002*, pages 117–128.

[Tang et al., 2005] Tang, Y., Deng, K., and Zhou, X. (2005). The design space of CMP vs. SMT for high performance embedded processor. In Yang, L., editor, *Embedded Software and Systems*, volume 3820/2005 of *Lecture Notes in Computer Science*, pages 30–38. Springer-Verlag, Berlin.

[Tullsen et al., 1995] Tullsen, D., Eggers, S., and Levy, H. (1995). Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403.

[Wulf and McKee, 1995] Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24.

[Zhuravlev et al., 2010] Zhuravlev, S., Blagodurov, S., and Fedorova, A. (2010). Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th Edition on Architectural Support for Programming Languages and Operating Systems 2010*, pages 129–141.

[Zwick, 2010a] Zwick, M. (2010a). Evaluation of cache contention prediction techniques: Further results and plots. Technical report, Technische Universität München.

[Zwick, 2010b] Zwick, M. (2010b). Predicting memory phases. In Ao, S.-I., Rieger, B., and Amouzegar, M. A., editors, *Machine Learning and Systems Engineering*, volume 68 of *Lecture Notes in Electrical Engineering*, pages 411–421. Springer-Verlag, Berlin.

[Zwick, 2011] Zwick, M. (2011). Setvectors - an efficient method to predict cache contention. In Ao, S.-I., Castillo, O., and Huang, X., editors, *Intelligent Control and Computer Engineering*, volume 70 of *Lecture Notes in Electrical Engineering*. Springer-Verlag, Berlin.

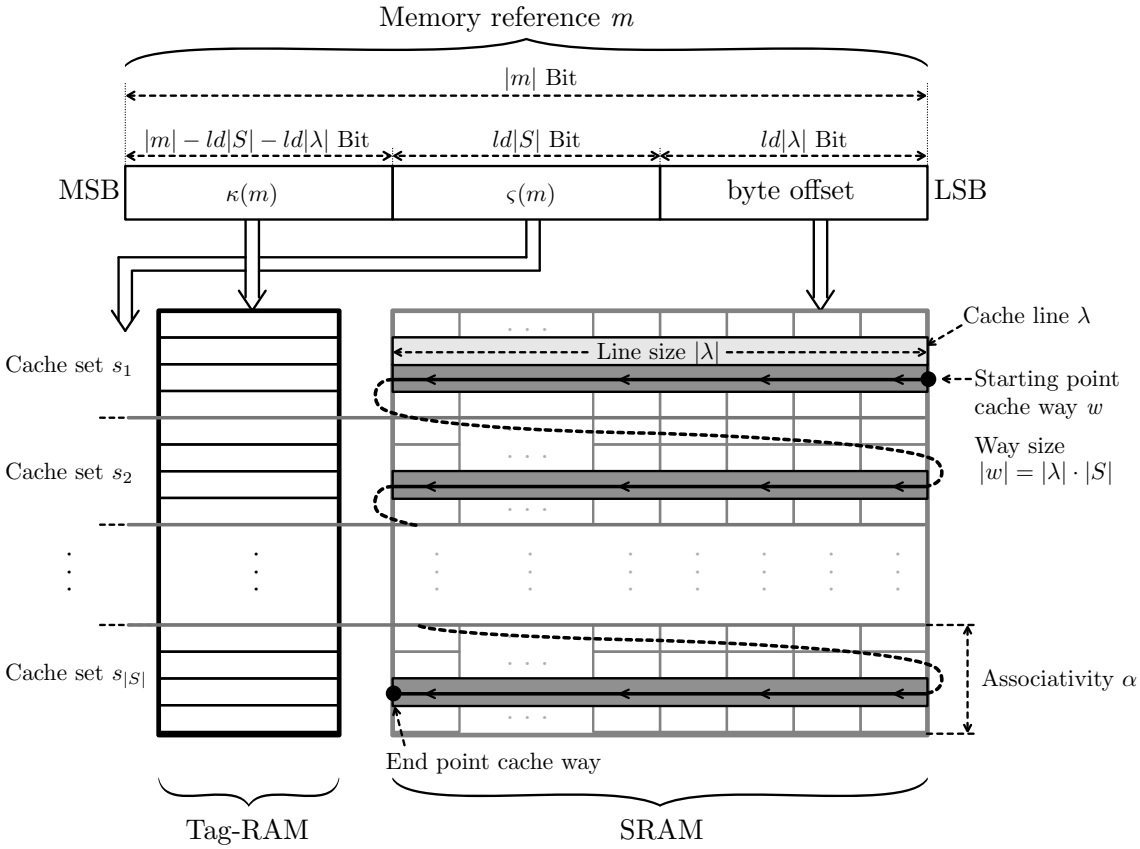
[Zwick et al., 2009a] Zwick, M., Durkovic, M., Obermeier, F., Bamberger, W., and Diepold, K. (2009a). MCCCSim - A highly configurable multi core cache contention simulator. Technical report, Technische Universität München.

[Zwick et al., 2009b] Zwick, M., Durkovic, M., Obermeier, F., and Diepold, K. (2009b). Setvectors for memory phase classification. In Ao, S. I., Douglas, C., Grundfest, W. S., and Burgstone, J., editors, *Proceedings of the World Congress on Engineering and Computer Science 2009*, volume I, pages 322–327. Newswood Limited, Hong Kong.

[Zwick et al., 2010] Zwick, M., Obermeier, F., and Diepold, K. (2010). Predicting cache contention with Setvectors. In Ao, S. I., Castillo, O., Douglas, C., Dagan Feng, D., and Lee, J.-A., editors, *Proceedings of the International MultiConference of Engineers and Computer Scientists 2010*, volume I, pages 244–251. Newswood Limited, Hong Kong.

Appendix

Cache Glossary



Example LRU Stack $\zeta^S_{s_j}$ for cache set s_j if $\alpha = 4$:

$\zeta^S_{s_j}(4)$	D
$\zeta^S_{s_j}(3)$	C
$\zeta^S_{s_j}(2)$	B
$\zeta^S_{s_j}(1)$	A

Operations:

$\zeta^S_{s_j}(3) = C$

$\zeta^S_{s_j}\{C\} = 3$

Figure 40: Cache glossary.

Stack Distance Histograms

Figure 41 shows stack distance histograms $H^{sd}_{a,\iota_{\digamma}\,i}$ for all $a \in A$; $\digamma = 2^{29}$, $i = 1$, $\alpha = 8$. Note that these histograms represent *all* memory references I apply in my evaluation.

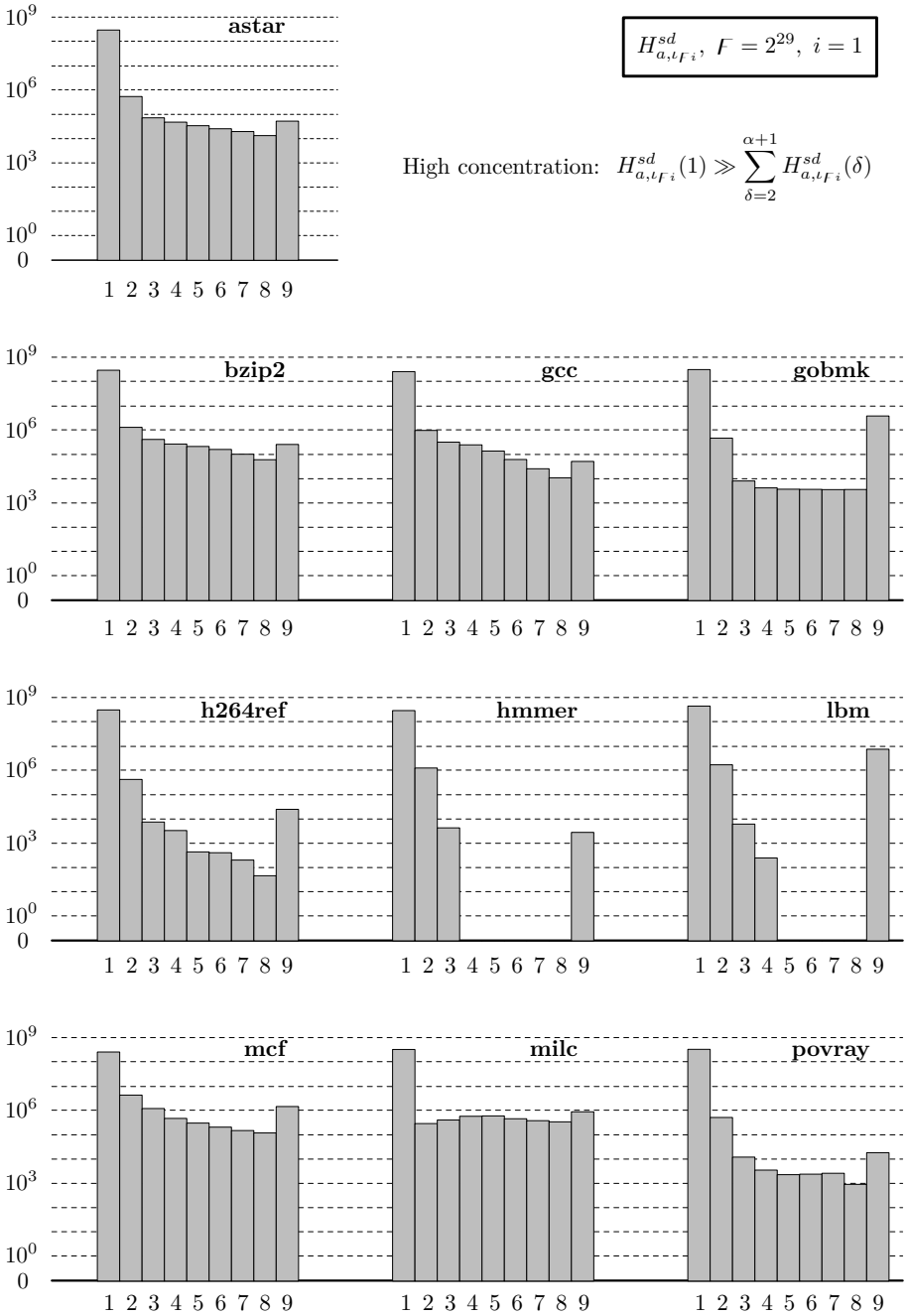


Figure 41: Stack distance histograms $H^{sd}_{a,\iota_{\digamma}\,i}$ for all $a \in A$; $\digamma = 2^{29}$, $i = 1$.

For smaller interval sizes Γ, stack distance histogram entries get thinned out and many entries become 0. Note the high concentration of the histograms that makes *hit* and *access* based methods perform poorly, while *miss* based methods generally show good prediction performance.

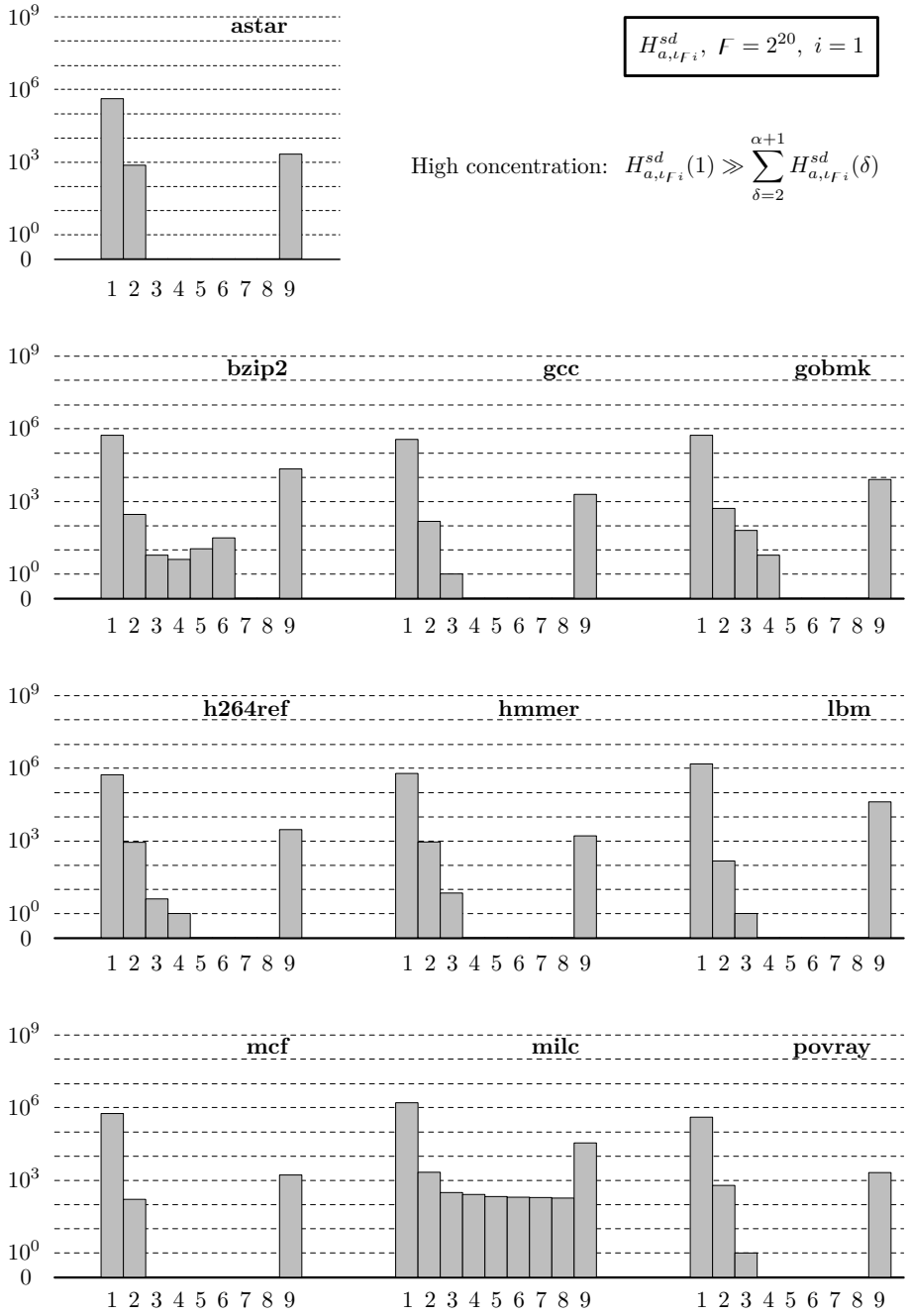


Figure 42: Stack distance histograms $H^{sd}_{a,\iota_{\Gamma i}}$ for all $a \in A$; $\Gamma = 2^{20}$, $i = 1$.

Note that on per-cache-set stack distance histograms $H^{sd,S}_{a,\iota_{F\,i},s}$, entries $2 \ldots \alpha$ are 0 for most histograms. Due to spatial locality, there are even histograms that do not have a single entry different from 0. Figure 43 shows per-cache-set stack distance histograms $H^{sd,S}_{a,\iota_{F\,i},s}$ for all $a \in A$ and $F = 2^{20}$, $i = 1$, $s = 1$.

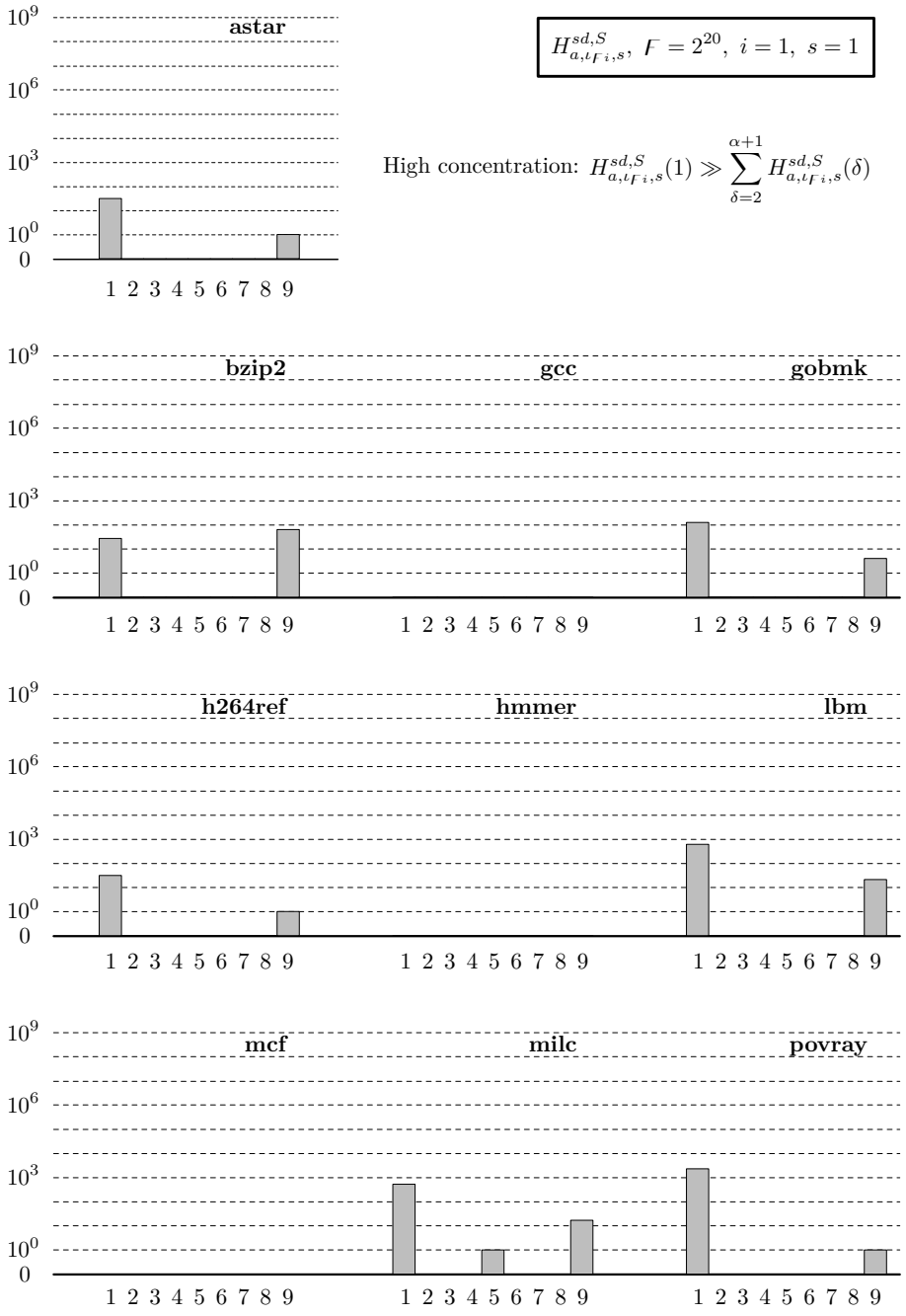


Figure 43: Per-cache-set stack distance histograms $H^{sd,S}_{a,\iota_{F\,i},s}$.

Distributions

NMRD - Normalized Mean Ranking Difference

Figure 44 exemplarily shows NMRD distribution for *FOA (one)*, a cache contention prediction technique of rather poor accuracy, and *Prob*, a prediction technique of high accuracy. You can see that *Prob* drops off much faster than *FOA (one)*, i.e. there are much more high (=poor) NMRD values for *FOA (one)* than for *Prob*.

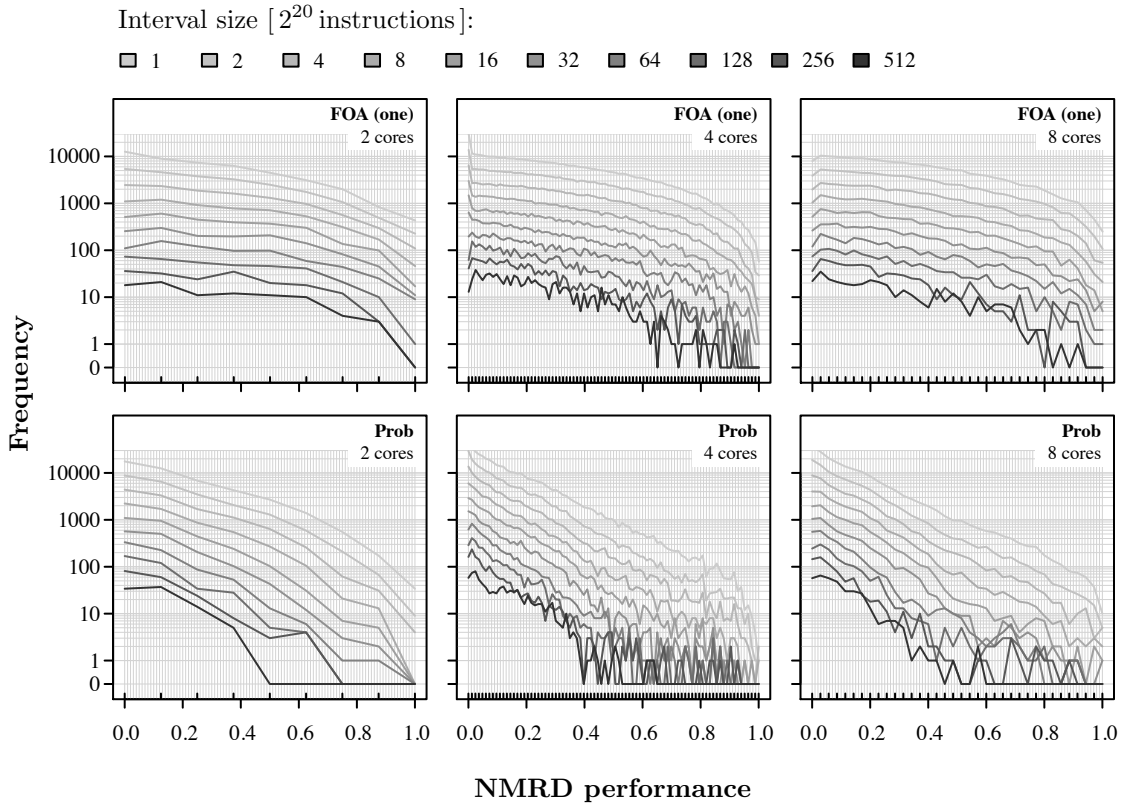


Figure 44: NMRD distribution for the *FOA (one)* and the *Prob* techniques.

Note that I calculate NMRD values for NMRD distribution *per co-schedule*, i.e. per C_a, and not per set of co-schedules, i.e. $\mathbf{C}_a$, as it is performed in equation 75. Therefore, I apply $MRD^{max} = |\mathbf{C}_a^{\psi}| - 1$. See [Zwick, 2010a] for NMRD distributions of all other prediction methods.

MP - Mean Penalty

Figure 45 exemplarily shows MP distribution for the low performing *FOA (one)* technique and the high performing *Prob* technique. As it has been the case in NMRD distribution, *Prob* drops off much faster than *FOA (one)*.

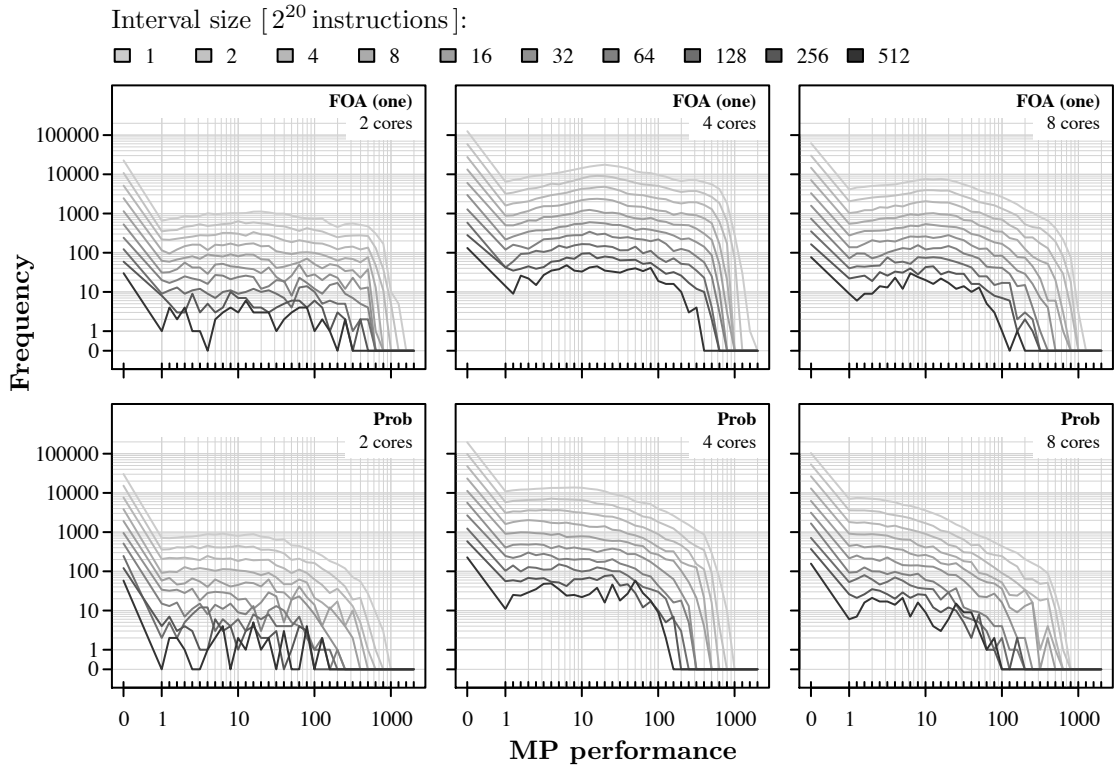


Figure 45: MP distribution for the *FOA (one)* and the *Prob* techniques.

Figure 46 shows how values in figure 45 have to be interpreted. Beginning from 1, both axis are of logarithmic scale. See [Zwick, 2010a] for further information and distributions.

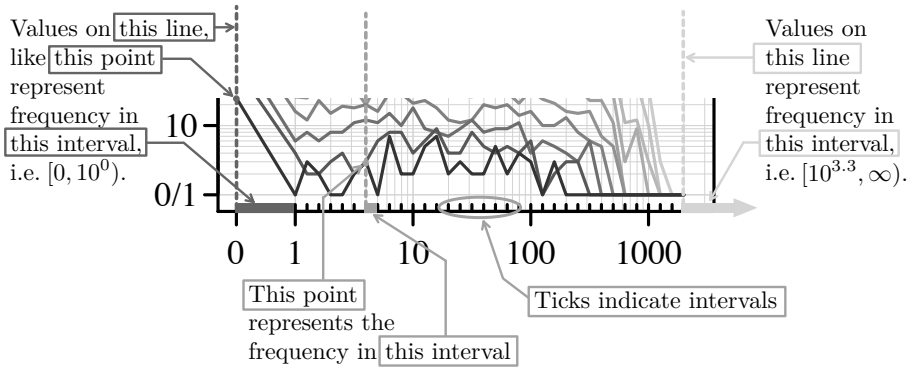


Figure 46: MP distribution glossary.

List of Symbols and Abbrevations

Symbol	Meaning
$\leftarrow$	Assignment operator used in algorithms; $a \leftarrow b$ means that the value of b is assigned to a
$\mapsto$	Maps to; $a \mapsto b$ means that a is mapped to b; in the context of algorithm description, $a \mapsto b$ means that variable a in the algorithm has to be replaced by variable b
$=$	Compare and assignment operator; in equations, I apply '=' both as assignment and compare operator, depending on the context; in algorithms, I apply '=' to compare two values; then, '=' returns either **true** or **false**
b **op** c ? d : e	Conditional value; **op** is any operation that returns a boolean value, for example $<, \geq, \ldots$; if b **op** c is **true**, then the whole expression is replaced by the value of d; otherwise, it is replaced by the value of e
α	Cache associativity, i.e. the number of cache lines per cache set
$\alpha'_{C_a,\iota_F i}$	Effective cache associativity introduced to application a when sharing the cache with applications C_a in interval $\iota_F i$
$\aleph^G_{a,\iota_F i}$	Activity vector of size $2 \cdot \lvert G\rvert = 64$ bit; $\aleph^G_{a,\iota_F i} = \left[\aleph^{acc,G}_{a,\iota_F i} \;\; \aleph^{miss,G}_{a,\iota_F i}\right]$
$\aleph^{acc,G}_{a,\iota_F i}$	Activity vector for cache accesses of size $\lvert G\rvert = 32$ bit
$\aleph^{miss,G}_{a,\iota_F i}$	Activity vector for cache misses of size $\lvert G\rvert = 32$ bit
$\aleph^S_{a,\iota_F i}$	Activity vector of size $2 \cdot \lvert S\rvert = 4096$ bit; $\aleph^S_{a,\iota_F i} = \left[\aleph^{acc,S}_{a,\iota_F i} \;\; \aleph^{miss,S}_{a,\iota_F i}\right]$
$\aleph^{acc,S}_{a,\iota_F i}$	Activity vector for cache accesses of size $\lvert S\rvert = 2048$ bit
$\aleph^{miss,S}_{a,\iota_F i}$	Activity vector for cache misses of size $\lvert S\rvert = 2048$ bit
a	Application
A	Set of SPEC 2006 test benchmark suite applications I employ for evaluation; $A =$ { astar, bzip2, gcc, gobmk, h264ref, hmmer, lbm, mcf, milc, povray }
$A \backslash \{a\}$	Set of applications without application a
$\lvert A\rvert$	Number of applications in set A

Continued on next page

Symbol	Meaning
A'	Set of applications $\in A$ that are executed in parallel on the architecture presented in figure 20. Note that each application $\in A'$ is executed on a separate core and the number of cores equals the number of applications, i.e. $\psi = \|A'\|$
API	Application programming interface
β	Base of an exponential function
c_a, $c_{a,j}$	Application that gets co-scheduled with application a; $c_{a,j} \in C_a$
C_a, $C_{a,j}$	Set of applications $C_a = \{c_{a,1} \ldots, c_{a,\|C_a\|}\}$ that get co-scheduled with application a; in my evaluation, $\{a\} \cup C_a$ is the set of applications that concurrently get co-scheduled and permanently contend for shared cache
$\mathbf{C}_a$, $\mathbf{C}_a^{\psi}$	Set of candidate co-schedules for application a; $\mathbf{C}_a = \mathbf{C}_a^{\psi} = \{C_{a,1}^{\psi}, \ldots C_{a,\|\mathbf{C}_a\|}^{\psi}\}$; $C_{a,j}^{\psi} = \{c_{a,1}, \ldots c_{a,\psi-1}\}$ refers to one of several sets of applications that are co-scheduled with a; note that I refer to $C_{a,j}^{\psi}$ and C_a interchangeably; generally, $\mathbf{C}_a^{\psi} = \mathfrak{C}(A \backslash \{a\}, \psi - 1)$
$\mathfrak{C}(A, \psi)$	Operator that returns all possible combinations of elements in set A that have length $\psi \leq \|A\|$; $\mathfrak{C}(A, \psi) = \left\{ \{a'_1, a'_2, \ldots a'_{\psi}\} \mid a'_i \in A \backslash \{a'_1, a'_2, \ldots a'_{i-1}\} \right\}$
$\|\mathfrak{C}(A, \psi)\|$	Number of all combinations of elements in set A of length ψ; $\psi \leq \|A\|$
$cseq(\delta, \nu)$	Circular sequence; a circular sequence $cseq(\delta, \nu)$ is a sequence $seq(\delta, \nu)$ whose first and last reference map to the same cache line and there is no other reference in the sequence that maps to that cache line
CMP	Chip multiprocessor; chip multiprocessing
δ	Distance in a stack distance histogram
$\delta_{a,\iota_F\, i,s}^{max,S}$	Index of the last entry of a stack distance histogram $H_{a,\iota_F\, i,s}^{sd,S}$ that is different from 0; generally $1 \leq \delta_{a,\iota_F\, i,s}^{max,S} \leq \alpha$
F	Window size, also named interval size or interval width; relates to the number of *instructions* an application executes in an interval; $F \in \boldsymbol{F}$
$\boldsymbol{F}$	Set of interval widths; $\boldsymbol{F} = \{2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}, 2^{25}, 2^{26}, 2^{27}, 2^{28}, 2^{29}\}$
DRAM	Dynamic random access memory

Continued on next page

Symbol	Meaning
$E(\nu_{c_a,\iota_{F}i}(\delta))$	Expected number of memory accesses that application c_a performs in the time that application a, on average, takes to refer to δ different cache lines
$\gamma(s)$	Operation that transforms the address of a cache set to an address of a cache group; $\gamma(s) = (s-1) \cdot \frac{\lvert G\rvert}{\lvert S\rvert} + 1$
G	Set of groups; $G = \{g_1, \ldots g_{\lvert G\rvert}\}$
$\lvert G\rvert$	Number of groups; Activity vector method: $\lvert G\rvert = 32$; SDC method, variation 'lru set group': $\lvert G\rvert = 16$
$\mathcal{GG}(\psi, F)$	Gain vs. cost evaluation function
$H_{a,\iota_{F}i}$	Histogram representing any information related to application a, interval $\iota_{F}i$; if it is said that histogram $H_{a,\iota_{F}i}$ is of capacity $D, D \in \mathbb{N}^+$, then $H_{a,\iota_{F}i}$ can hold up to D elements $\in \mathbb{N}_0^+$; if $H_{a,\iota_{F}i}$ is said to be a *three dimensional* histogram of capacity $D \times N, D \in \mathbb{N}^+, N \in \mathbb{N}^+$, then $H_{a,\iota_{F}i}$ can hold up to $D \cdot N$ elements $\in N_0^+$
$H_{a,\iota_{F}i}(\delta)$	Operation that references histogram element at position δ;
$H_{a,\iota_{F}i}(\delta, \nu)$	Operation that references histogram element at position δ, ν in a three dimensional histogram;$1 \leq \delta \leq D$ and $1 \leq \nu \leq N$, unless otherwise noted
$H^{sd}_{a,\iota_{F}i}$	Stack distance histogram of capacity $\alpha + 1$; entries $\delta \in \{1, 2 \ldots \alpha\}$ hold the number of references of application a in execution interval $\iota_{F}i$ that map to the δ most recently used cache line of any cache set; $H^{sd}_{a,\iota_{F}i}(\alpha + 1)$ holds the number of misses of application a in $\iota_{F}i$
$H^{sd,S}_{a,\iota_{F}i,s}$	Stack distance histogram of capacity $\alpha + 1$ that holds stack distance information for application a, interval $\iota_{F}i$ that relate to cache set s
$H^{sd,ext}_{a,\iota_{F}i}$	Stack distance histogram of extended size $\alpha \cdot \lvert G\rvert$ for application a, execution interval $\iota_{F}i$
ι_F	Tuple of intervals of window size F each; $\iota_F = (\iota_{F}1, \ldots \iota_{F\lvert\iota_F\rvert})$
$\lvert\iota_F\rvert$	Number of intervals in case interval size = F; $\lvert\iota_F\rvert = \frac{2^{29}}{F}$
$\iota_{F}i$	Interval i in the tuple of intervals ι_F; $\iota_{F}i \in \iota_F$

Continued on next page

Symbol	Meaning
$\kappa^{\mathrm{L1C}}(m)$	Operation that extracts the *key* part of a memory address m for a private L1 cache; given a cache of way size w, then $\kappa^{\mathrm{L1C}}(m) = m/w$; generally, $\kappa^{\mathrm{L1C}}(m) \geq 0$
$\kappa_a^{\mathrm{L2C}}(m)$	Operation that extracts the *key* part of a memory address m of an application a for a shared L2 cache; given a cache of way size w, then $\kappa_a^{\mathrm{L2C}}(m) = m/w + h(a) \cdot 2^{\lvert\lambda\rvert \cdot \alpha^{\mathrm{L2C}} \cdot \lvert S^{\mathrm{L2C}}\rvert}$; $h(a) \in \{1, 2, \dots \lvert A\rvert\}$ is the operation that returns a unique hash for each $a \in A$; α^{L2C} is the associativity of the cache, $\lvert S^{\mathrm{L2C}}\rvert$ the number of cache sets, and $\lvert\lambda\rvert$ the size of a cache line; note that $\forall_{a_i, a_j \in A,\, a_i \neq a_j} : \nexists_{m_i, m_j} : \kappa_{a_i}^{\mathrm{L2C}}(m_i) = \kappa_{a_j}^{\mathrm{L2C}}(m_j)$ and $\forall_m : \kappa_a^{\mathrm{L2C}}(m) \geq 0$
$\varkappa_{a,\iota_{\boldsymbol{F}}i}$	Number of different keys of application a, interval $\iota_{\boldsymbol{F}}i$ that map to identical cache set
$\varkappa^S_{a,\iota_{\boldsymbol{F}}i,s}$	Number of different keys of application a, interval $\iota_{\boldsymbol{F}}i$ that map to the same cache set s
l	Word length in byte; $l = 4$
λ	Cache line
$\lvert\lambda\rvert$	Size of a cache line in byte
Λ	Set of cache lines $\{\lambda_1, \dots \lambda_{\lvert\Lambda\rvert}\}$
L1C	Level 1 cache
L1T	Level 1 TLB
L2C	Level 2 cache
L2T	Level 2 TLB
LRU	Least recently used
LSB	Least significant bit
μ	Cache miss
m	Memory reference, also named *memory address* or simply *address* or *reference*
$\lvert m\rvert$	Word length of an address in *bits*; $\lvert m\rvert = 32$
mod	Modulo operation
M_a	Tuple of memory references $M_a = (m_{a,1}, m_{a,2}, \dots)$ of an application a

Continued on next page

Symbol	Meaning
$\|M_a\|$	Number of references in M_a
M_{a,ι_F}	$M_{a,\iota_F} = (M_{a,\iota_F 1}, \ldots M_{a,\iota_{F\,\|M_{a,\iota_F}\|}})$ is the tuple that holds $\|M_{a,\iota_F}\|$ tuples of memory references $M_{a,\iota_F i}$, where each $M_{a,\iota_F i}$ refers to memory references that originate from F instructions; $\forall_{F \in \boldsymbol{F}} : \sum_{i=1}^{\|M_{a,\iota_F}\|} \|M_{a,\iota_F i}\| = \|M_a\|$
$\|M_{a,\iota_F}\|$	Number of intervals of window size F in M_a; generally, $\forall_{a \in A} : \|M_{a,\iota_F}\| = \|\iota_F\|$
$M_{a,\iota_F i}$	Tuple of memory references $(m_{a,\iota_F i,1}, \ldots, m_{a,\iota_F i,\|M_{a,\iota_F i}\|})$ of application a in execution interval ι_{Fi}; holds memory references that originate from instructions $i \cdot F \ldots (i+1) \cdot F - 1$ of application a; $M_{a,\iota_F i} \subseteq M_a$
$\|M_{a,\iota_F i}\|$	Number of memory references of application a in interval ι_{Fi}; please note the difference: While $\|\iota_{Fi}\| = F$ refers to the number of *instructions* in an interval ι_{Fi} and is independent of any application, $\|M_{a,\iota_F i}\|$ refers to the number of *memory references* of an application $a \in A$ in execution interval ι_{Fi} and is specific to each $a \in A$
Mem	Main memory (random access memory)
MMU	Memory management unit
$MP(\psi, F)$	Mean penalty evaluation metric; evaluates cache contention prediction techniques
MRU	Most recently used
MSB	Most significant bit
$NMRD(\psi, F)$	Normalized Mean Ranking Difference; measure to evaluate general ranking performance
ν	Number of references to a cache or a cache set in a sequence $seq(\delta, \nu)$ or circular sequence $cseq(\delta, \nu)$;
ν_{max}	Maximum number of references that are considered in a sequence $seq(\delta, \nu)$ or circular sequence $cseq(\delta, \nu)$
$\overline{\nu}_{a,\iota_F i}$	Average number of memory references of application a in interval ι_{Fi} that occur when a accesses δ different cache lines per set
Ω	Relative offset/threshold applied to distinguish between set/groups of high activity and low activity respectively; $\Omega = \frac{3}{4}$

Continued on next page

Symbol	Meaning
$\Omega_{\boldsymbol{\Gamma}}^{acc,G}$	Threshold; number of references to a group of cache sets that are necessary to tag a group as 'highly active' with respect to memory references
$\Omega_{\boldsymbol{\Gamma}}^{acc,S}$	Threshold; number of references to a cache set that are necessary to tag a cache set as 'highly active' with respect to memory accesses
$\Omega_{\boldsymbol{\Gamma}}^{miss,G}$	Threshold; number of stand-alone cache misses that have to occur in a group of cache sets in order to tag that group of cache sets as 'highly active' with respect to cache misses
$\Omega_{\boldsymbol{\Gamma}}^{miss,S}$	Threshold; number of stand-alone cache misses that have to occur in a cache set in order to tag that cache set as 'highly active' with respect to cache misses
π	Penalty
$\pi_{C_a,\iota_{\boldsymbol{\Gamma}}i}$	Penalty introduced from co-schedule C_a to application a in execution interval $\iota_{\boldsymbol{\Gamma}}i$; $\pi_{C_a,\iota_{\boldsymbol{\Gamma}}i} = t_{C_a,\iota_{\boldsymbol{\Gamma}}i} - t_{a,\iota_{\boldsymbol{\Gamma}}i}$
$\boldsymbol{\pi}_{\mathbf{C}_{a,\iota_{\boldsymbol{\Gamma}}i}}(\rho)$	Penalty the candidate co-schedule in $\mathbf{C}_a$ that has ranking position ρ introduces to application a in execution interval $\iota_{\boldsymbol{\Gamma}}i$
ϖ	Memory page
$\varpi_a(m)$	Operation that returns the page address of a memory reference m of application a; $\varpi_a(m) = (m \ mod \ \lvert\varpi\rvert) + h(a) \cdot 2^{\lvert m\rvert - ld(\lvert\varpi\rvert)}$, where $h(a) \in \{1, 2, \dots \lvert A\rvert\}$ is the operation that returns a unique hash for each $a \in A$ and $\lvert\varpi\rvert$ is the page size; note that $\forall_{a_i,a_j \in A} : a_i \neq a_j \Rightarrow \nexists_{m_i,m_j} : \varpi_{a_i}(m_i) = \varpi_{a_j}(m_j)$ and $\forall_{a \in A, \lceil ld(\lvert A\rvert)\rceil < ld(\varpi)} : \varpi_a(m) \geq 0$
$\lvert\varpi\rvert$	Page size measured in byte; $\lvert\varpi\rvert = 2^{12}$
ψ	Parallelism applied when evaluating cache contention prediction methods; $\psi \in \boldsymbol{\psi} = \{2, 4, 8\}$; given ψ, the applied processor consists of ψ processor cores, each featuring a privat L1 cache, while all ψ cores share a common L2 cache, as it is demonstrated in figure 20
$\boldsymbol{\psi}$	Set of parallelisms applied in the evaluation; $\boldsymbol{\psi} = \{2, 4, 8\}$
$p_{C_a,\iota_{\boldsymbol{\Gamma}}i}$	Prediction of cache contention the applications in C_a introduce to application a in interval $\iota_{\boldsymbol{\Gamma}}i$

Continued on next page

Symbol	Meaning
$PPBAB$	Penalty Predicted Best vs. Actual Best
$PPBRS$	Penalty Predicted Best vs. Random Selection; equivalent to gain
ρ	Ranking position; $1 \leq \rho \leq \lvert\mathbf{C}_a^{\psi}\rvert$
$\rho^{\text{sim}}_{C_a,\iota_{\boldsymbol{F}}i}$	Simulated ranking position of candidate co-schedule C_a in interval $\iota_{\boldsymbol{F}i}$
$\rho^{\text{pred}}_{C_a,\iota_{\boldsymbol{F}}i}$	Predicted ranking position of candidate co-schedule C_a in interval $\iota_{\boldsymbol{F}i}$
RAM	Random access memory
s	Cache set
S	Set of cache sets; $S = \{s_1 \dots s_{\lvert S\rvert}\}$
$\lvert S\rvert$	Number of cache sets of a processor cache
$\varsigma(m)$	Operation that extracts the set address s from a memory reference m; given a cache of way size $\lvert w\rvert$ and line size $\lvert\lambda\rvert$, then $\varsigma(m) = ((m/\lvert\lambda\rvert)\ mod\ (\lvert w\rvert/\lvert\lambda\rvert)) + 1$; note that $1 \leq \varsigma(m) \leq \lvert S\rvert$
$seq(\delta, \nu)$	Sequence of memory references; δ represents the number of memory references within the sequence that map to the same cache set, but to a different cache line; ν represents the total number of memory references within the sequence
SMT	Simultaneous multithreading
$t_{a,\iota_{\boldsymbol{F}}i}$	Time application a spends on memory accesses in execution interval $\iota_{\boldsymbol{F}i}$, when a is executed in absence of any other application (stand-alone)
$t_{C_a,\iota_{\boldsymbol{F}}i}$	Time application a spends on memory accesses in execution interval $\iota_{\boldsymbol{F}i}$, when a is co-scheduled with applications C_a
TLB	Translation look-aside buffer
w	Cache way
$\lvert w\rvert$	Way size of a processor cache in byte
$\xi^{S}_{C_a,\iota_{\boldsymbol{F}}i,s}$	Masking element that indicates if the sum of the number of different keys of applications $\{a\} \cup C_a$ that are mapped to cache set s exceed associativity α of the cache ($\xi = 1$) or not ($\xi = 0$)
ζ	Stack; if it is said that stack ζ is of capacity $K, K \in \mathbb{N}^+$, then ζ can hold up to K elements

Continued on next page

Symbol	Meaning
$\zeta(\delta)$	Operation that references the element at stack position δ; if K is the capacity of the stack, then $1 \leq \delta \leq K$
$\zeta\{x\}$	Operation that returns the stack position of element x according to $$\zeta\{x\} = \begin{cases} \delta, & \text{if } \exists\ \delta, 1 \leq \delta \leq K:\ \zeta(\delta) = x \\ K+1, & \text{if } \nexists\ \delta, 1 \leq \delta \leq K:\ \zeta(\delta) = x; \end{cases} \quad (85)$$ if there is *more than one* δ with $\zeta(\delta) = x$, then the δ with the least value is returned; K is the capacity of the stack
ζ^S	Set of LRU stacks; $\zeta^S = \{\zeta_1^S, \ldots \zeta_{\lvert S\rvert}^S\}$; each ζ_s^S is of capacity α