On The Accuracy of Cache Sharing Models

Vlastimil Babka
babka@d3s.mff.cuni.cz

Peter Libič
libic@d3s.mff.cuni.cz

Tomáš Martinec
martinec@d3s.mff.cuni.cz

Petr Tůma
tuma@d3s.mff.cuni.cz

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics, Charles University
Malostranské náměstí 25, Prague 1, 118 00, Czech Republic

ABSTRACT

Memory caches significantly improve the performance of workloads that have temporal and spatial locality by providing faster access to data. Current processor designs have multiple cores sharing a cache. To accurately model a workload performance and to improve system throughput by intelligently scheduling workloads on cores, we need to understand how sharing caches between workloads affects their data accesses.

Past research has developed analytical models that estimate the cache behavior for combined workloads given the stack distance profiles describing these workloads. We extend this research by presenting an analytical model with contributions to accuracy and composability – our model makes fewer simplifying assumptions than earlier models, and its output is in the same format as its input, which is an important property for hierarchical composition during software performance modeling.

To compare the accuracy of our analytical model with earlier models, we attempted to reproduce the reported accuracy of those models. This proved to be difficult. We provide additional insight into the major factors that influence analytical model accuracy.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: *Modeling techniques, Measurement techniques*; B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Performance, Measurement, Experimentation

Keywords

processor caches, performance modeling, cache models

ICPE'12, April 22–25, 2012, Boston, Massachusetts, USA

1. MOTIVATION

The motivation for this paper originates with our work on software performance modeling. In software performance models, the performance of a software system is typically derived from the performance of the constituting components and the component interactions [7]. The interactions may be explicit, such as method invocation or message passing, or implicit, such as competing for a shared resource. Often, memory caches are one such resource.

Memory caches significantly improve the performance of workloads that have temporal and spatial locality by providing faster access to data. An accurate software performance model may therefore need to capture the performance impact of multiple workloads competing for memory caches [2].

Analytical models that estimate the cache behavior for a combination of workloads have been developed. These models target applications such as processor design or workload scheduling [8, 9, 13, 10]. For application in software performance modeling, additional requirements exist:

- Software performance models typically deal with time. The cache model therefore needs to calculate the timing penalties, rather than only the cache miss counts or the cache miss ratios.
- Software performance models often contain hierarchically composed components. The cache model should therefore permit the corresponding evaluation of hierarchically composed workloads.

Our first contribution is a cache model that reflects these requirements. The model belongs to the family of analytical cache models that use stack distance profiles[1] to describe the workload behavior. As a step towards hierarchical composition, our model calculates the resulting stack distance profile of a combination of workloads in addition to the timing penalties.

An important property of a cache model is its accuracy. The evaluation of accuracy is hindered by two factors: the difficulty in obtaining precise model inputs and the difficulty in creating controlled experimental conditions. Existing research differs in how these difficulties are addressed –

[1]A stack distance profile tells the probability of accessing a given number of different cache lines in between consecutive accesses to the same cache line.

striving for more precise inputs and more controlled conditions often implies significant overhead and limited realism, but imprecise inputs and uncontrolled conditions can also be detrimental.

Our second contribution is an evaluation of the modeling accuracy in realistic settings. In particular, we improve the existing methods for collecting the model inputs on real hardware, impose no additional control over the execution phases of the competing workloads, and compare the modeling results with measurements on real hardware.

Because the accuracy of a model depends on multiple factors, interpreting particular accuracy results is difficult. This prevents direct comparison of the accuracy results reported in previous research. It also complicates estimating how much of the reported accuracy is preserved in different settings.

As our third contribution, we use experiments to isolate the individual accuracy factors and provide insights into how the experimental settings impact accuracy. We also develop a method for collecting the stack distance profiles that mediates the influence of the workload profile stability and the replacement policy approximation on accuracy.[2]

The paper is structured as follows. We start by introducing our cache model in Section 2. We explain what tools we use to collect the model inputs in Section 3. In Section 4, we present the evaluation results, first in an overall view and then focusing on the major accuracy factors. In Section 5, we outline the relationship of our model to existing research, both in design and in evaluation. We summarize our contribution in Section 6.

2. CACHE SHARING MODEL

The cache sharing model for a cache with associativity A takes as an input for each workload x the stack distance profile sdp_x, the mean number of cache accesses per instruction API_x, the number of instructions per processor clock cycle when running in isolation IPC_x, and the cache miss penalty in processor clock cycles mp_x. The output consists of modified stack distance profiles, sdp'_x, and modified IPC values, IPC'_x, when each workload is running in parallel with other workloads. The profiles and IPC values can also be combined to get a single profile and IPC value representing the combined workload, for use when the models are composed in hierarchies.

For sake of brevity, we describe the model for two parallel workloads, denoted 1 and 2. The extension for multiple parallel workloads is straightforward.

In principle, the modified stack distance profile of workload 1, when running in parallel with workload 2, is derived in three steps done for each stack distance d, $1 \leq d \leq A$:

1. Calculate the average time (in processor clock cycles) between two consecutive accesses to the same cache line, where the second access has a stack distance d. We call this *reuse time* of stack distance d and denote as r_d.

2. Estimate the number of distinct cache lines accessed by the other workload during the time r_d.

3. Increase the stack distance of accesses for the original stack distance d by the number estimated in step 2.

We now describe the individual steps in detail.

Step 1. To obtain r_d, we observe that during the reuse time, the workload has to access $d-1$ distinct cache lines to build up the distance of the reused cache line in the LRU stack, and then access the reused line. With the assumption of independent accesses, this equals the mean time to achieve cache set occupation from 0 to d distinct cache lines, and can be derived from the stack distance profile as follows.

We model the process of cache set occupation of a workload as a Markov chain, with $d+1$ states representing the number of occupied cache lines between 0 and d. The transition matrix $\mathbf{P}_d$ is defined:[3]

- $p_{0,1} = 1$
- $p_{i,i} = hit(i), 1 \leq i < d$
- $p_{i,i+1} = 1 - hit(i), 1 \leq i < d$
- $p_{d,d} = 1$
- $p_{i,j} = 0$ otherwise, $0 \leq i \leq d$, $0 \leq j \leq d$

In the above, $hit(i)$ is the probability of a cache hit with i lines occupied, and can be derived directly from the stack distance profile:

$$hit(i) = \frac{\sum_{j=1}^{i} sdp_1(j)}{\sum_{j=1}^{A+1} sdp_1(j)} \tag{1}$$

With the state d being absorbing, we can calculate the expected number of steps t_d from the initial state 0 to the absorption in state d [11]:

$$t_d = \sum_{j=0}^{d-1} \left[(\mathbf{I} - \mathbf{T})^{-1} \right]_{0,j} \tag{2}$$

where $\mathbf{I}$ is a d-by-d identity matrix, $\mathbf{T} = [p_{i,j}]$, $i, j \in \{0, \ldots, d-1\}$.

The mean number of accesses between reuses of cache lines with stack distance d is equal to t_d and the reuse time (in processor cycles) can be derived:

$$r_d = \frac{t_d}{API_1 \cdot IPC_1} \tag{3}$$

Step 2. To determine the interference from the second workload, we first calculate the average number of accesses by the second workload during the reuse time of the first workload:

$$a_d = r_d \cdot API_2 \cdot IPC_2 \tag{4}$$

To calculate how many *distinct* lines exist in a_d consecutive accesses by the second workload, we use the transition matrix of the second workload with the number of occupied cache lines ranging from 0 to the cache associativity A, denoted as $\mathbf{Q}_A$. We derive the probability vectors

[2]See Section 4.5 for workload profile stability and Section 4.6 for replacement policy approximation.

[3]The indices start from zero to match the states directly.

$\mathbf{D}(n) = \{D_0(n), \ldots, D_A(n)\}$, where $D_m(n)$ is the probability of the second workload accessing m distinct cache lines after n accesses, where $0 \leq m \leq A$, as follows [11]:

$$\mathbf{D}(n) = \mathbf{u}\mathbf{Q}_A^n, \tag{5}$$

where $\mathbf{u} = \{u_0, \ldots, u_A\}, u_0 = 1$ and $u_m = 0, m > 0$.

The vector $\mathbf{D}(a_d)$ thus gives the probability distribution of the number of distinct cache lines accessed by the second workload. For non-integer values of a_d, the vector is defined as an element-wise linear interpolation between the vectors $\mathbf{D}(\lfloor a_d \rfloor)$ and $\mathbf{D}(\lceil a_d \rceil)$.

Step 3. A partial distance profile pdp'_{1d}, describing the modified stack distances under sharing for accesses with original stack distance d, is calculated by taking the value $sdp_1(d)$ from the original profile and distributing it between distances d to $d + A$ proportionally to the values in $\mathbf{D}(a_d)$. Since accesses with resulting distance larger than $A + 1$ are always misses, we treat them as having the distance of $A+1$:

$$\begin{aligned} pdp'_{1d}(d+i) &= sdp_1(d) \cdot D_i(a_d), 0 \leq i < A+1-d \\ pdp'_{1d}(A+1) &= sdp_1(d) \cdot \sum_{i=A+1-d}^{A} D_i(a_d) \end{aligned} \tag{6}$$

After repeating the three steps for each $d, 1 \leq d \leq A$, the resulting distance profile sdp'_1 can be constructed by adding up the partial distance profiles pdp'_{1d} created in each repetition:

$$sdp'_1(i) = \sum_{d=1}^{A} pdp'_{1d}(i), 1 \leq i \leq A+1 \tag{7}$$

Cache misses in the original profile, $sdp_1(A+1)$, are simply added to $sdp'_1(A+1)$.

To determine IPC'_1, we consider how the ratio of misses per cache access increases due to the decrease of cache hit probability $hit(A)$ to $hit'(A)$, which is calculated from sdp'_1 using Eq. 1:

$$\Delta MPA_1 = hit(A) - hit'(A) \tag{8}$$

These extra cache misses can be translated to extra CPI using the workload cache miss penalty mp_1:

$$\Delta CPI_1 = \Delta MPA_1 \cdot API_1 \cdot mp_1 \tag{9}$$

The resulting IPC'_1 of the first workload thus follows:

$$IPC'_1 = ((1/IPC_1) + \Delta CPI_1)^{-1} \tag{10}$$

To determine sdp'_2 and IPC'_2, we repeat the whole process with the roles of the first and the second workload exchanged.

In equations 3 and 4, we have used the IPC values from isolated execution, although the workloads are in fact mutually influencing their IPC by executing in parallel over the shared cache. We solve this issue iteratively – in each iteration, new sdp'_x and IPC'_x values are calculated using the IPC'_x values from the previous iteration, starting with the isolated IPC_x inputs. Note that the stack distance profiles of the individual workloads are not modified during the iterations – in all steps, the isolated sdp_x profiles are used as inputs. We use a simple ϵ stability criterion on the IPC'_x values to determine solution convergence.

Finally, for composition purposes, the composed stack distance profile is calculated by element-wise averaging the sdp'_x profiles weighted by the memory access frequencies derived from IPC'_x and API_x.

3. EVALUATION TOOL SUPPORT

We evaluate the accuracy of the cache model by carrying out multiple experiments where various workload combinations are executed in parallel on cores that share a cache. The performance of the workload is both modeled and measured and the values are compared. Here, we describe the experimental workloads and the two techniques we use for collecting the stack distance profiles. Because any profiling technique can influence the collected profile, we use two different techniques to understand this influence on the accuracy of the model.

3.1 Experimental Workloads

Our workload combinations are pairs of workloads adopted from the SPEC CPU 2006 benchmark suite [1], supplemented with FFT and LZW calculations. The workloads execute in our experimental framework [3], which separates the initialization and the calculation parts of each workload and runs the calculation parts repeatedly to achieve steady cache sharing conditions. Even though the workloads can have multiple phases with different behavior, we avoid explicit synchronization that would make only particular phases compete against each other.

Depending on various technical properties of the code, getting a benchmark to work in our experimental framework may require a significant effort. To reduce this effort, we only use a subset of the SPEC CPU 2006 benchmark suite, but ensure that the included workloads have sufficient variation in their stack distance profiles. The subset consists of the 401.bzip2, 429.mcf, 444.namd, 458.sjeng, 462.libquantum, 470.lbm, and 473.astar benchmarks, with modifications adjusting the range of the accepted inputs as follows:

- 401.bzip2 was extended with a parameter for configuring the amount of data compressed, in addition to the file containing the data. Eight configurations are used, with the with amount of data set to either 1 MB or 2 MB and the input file set to one of *dryer.jpg*, *input.program*, *text.html* from 401.bzip2 inputs, and *100_100_130_cf_a.of* from 470.lbm inputs.
- 429.mcf uses a randomized subset of the *inp.in* input.
- 444.namd uses the *namd.input* input.
- 458.sjeng uses a subset of the *test.txt* input.
- 462.libquantum was modified to accept the input as arguments rather than reading it from a file. We use two input combinations, $(143, 25)$ and $(39, 25)$.
- 470.lbm was extended with a parameter for configuring the number of iterations performed per invocation. We use the *100_100_130_cf_a.of* input and one iteration per invocation.

- 473.astar uses the *lake.bin* with *lake.cfg* input.

Together with the FFT and LZW calculations, we have 139 workload combinations.

3.2 Stack Distance Collection: Valgrind Extension

The first technique we use for stack distance profile collection extends the cache profiler Cachegrind, which is a part of the Valgrind instrumentation and dynamic analysis tool [12]. When executing an application, Cachegrind observes the memory accesses and dynamically simulates a two-level cache hierarchy that is configured to match the host platform. The number of cache accesses and cache misses can be either reported for the whole program, or reported in more detail for program function and source line.

To simulate the cache hierarchy, Cachegrind maintains a LRU stack of cache line addresses for each cache set. Our extension added counters for all stack distances to produce the stack distance profiles.

To minimize the framework overhead, we use Valgrind Client Control trapdoor mechanism, which can instruct the simulator to start and stop counting, save the results, and reset the counters. We use this mechanism to let the simulator populate the LRU stacks during the warmup cycles, and only count the access distances during the measurement cycles. After measurement, the counters for each set are added together to form a single average profile – modeling with a separate profile for each set is also possible, but not investigated here.

Valgrind ignores hardware prefetching, because it only intercepts memory accesses at software level.

3.3 Stack Distance Collection: Stressmark Workload

The second technique we use for stack distance profile collection uses the Stressmark workload, described in [13]. Stressmark is designed to heavily use a configurable subset of the cache. This reduces the effective cache size for any other workload that runs with Stressmark. Observing the hardware performance event counters for multiple effective cache sizes allows us to derive the stack distance profiles and the miss penalties for the other workload.

During execution, Stressmark strives to occupy a particular number of ways in all cache sets by accessing the cache in a random sequence, going through all sets once for each way to be occupied. This access pattern results in a flat stack distance profile. By observing the shared cache access and miss counters, we calculate the average number of truly occupied ways across all sets as the accessed number of ways times the observed hit rate. By repeating the Stressmark execution for different number of occupied ways, we can observe the miss rate of the profiled workload under different effective cache sizes, which allows us to compute the stack distance profile.

Because the Stressmark workload competes with the profiled workload in all sets, it may not be able to achieve a high number of truly occupied ways against more intensive workloads. This means observations corresponding to small stack distances may be missing. For those distances, we distribute the remaining accesses equally. For observed distances that are not integers, we use linear interpolation.

We make two modifications to Stressmark to improve accuracy. To estimate the cache miss penalty, we use only misses due to sharing, rather than all missess. Specifically, we measure how much the processor clock cycle counter and the shared cache miss counter increase under sharing. The penalty is calculated as the cycle count increase over the miss count increase, with the least-square-error linear regression applied to calculate a single penalty value for each workload. We also use the page coloring-based memory allocator from [5] to ensure that in our Stressmark implementation, all cache sets are accessed and no conflict misses occur.

The Stressmark-based tool interacts with hardware prefetching. In particular, the hardware performance event counters used to determine Stressmark cache occupancy do not distinguish between cache misses due to prefetch hits and prefetch misses. Therefore, either both kinds of cache misses are counted or none are. Stressmark can also suppress prefetches that would otherwise happen [4, 2].

3.4 Stack Distance Collection: Discussion

Despite the instrumentation overhead, collecting profiles with Valgrind is actually faster than using Stressmark, which has to repeat the measurement for each stack distance from zero to the cache associativity. However, Valgrind is not a cycle-accurate simulator. Therefore, we collect the instruction throughput and miss penalty values using Stressmark even when we obtain the stack distance profiles with Valgrind.

4. EVALUATION RESULTS

To evaluate the modeling accuracy, we independently model and measure cache sharing on selected workload combinations, and compare the model predictions to the measurements. Our measurements are obtained on a Dell PowerEdge 1955 system with two Quad-Core Intel Xeon processors (Type E5345, Family 6, Model 15, Stepping 11, Clock 2.33 GHz), 8 GB DDR2-667 memory, Intel 5000P memory controller, Fedora Linux 8, gcc-4.1.2-33.x86 64, glibc-2.7-2.x86 64. The workloads run on cores that have a private 32 KB 8-way set associative instruction L1 cache, a private 32 KB 8-way set associative L1 data cache, and share a 4 MB 16-way set associative L2 unified cache.

In our experiments, we disable hardware prefetching, because neither of our stack distance profile collection techniques is hardware prefetch aware. If needed, the performance impact of hardware prefetching can be evaluated separately from the performance impact of cache sharing [4, 2].

In the following, MR stands for miss rate, IPC stands for instruction throughput, *isol* stands for values measured when running an isolated workload, *real* stands for values measured when running a workload combination, and *pred* stands for values predicted by the model.

4.1 Overview: IPC

As an overview, we plot the relative slowdown prediction, $(IPC_{isol} - IPC_{pred})/IPC_{isol}$, against the relative slowdown measurement, $(IPC_{isol} - IPC_{real})/IPC_{isol}$, for all workload combinations. A workload combination is displayed as two points, one for each workload. A dotted line indicates when the predicted slowdown equals the measured slowdown. Figure 1 plots the predictions based on the Valgrind profiles, Figure 2 plots the predictions based on the Stressmark profiles.

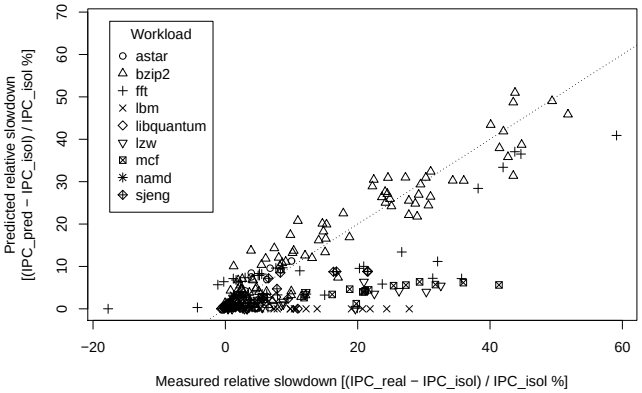

Figure 1: IPC slowdown prediction against IPC slowdown measurement, Valgrind profiles.

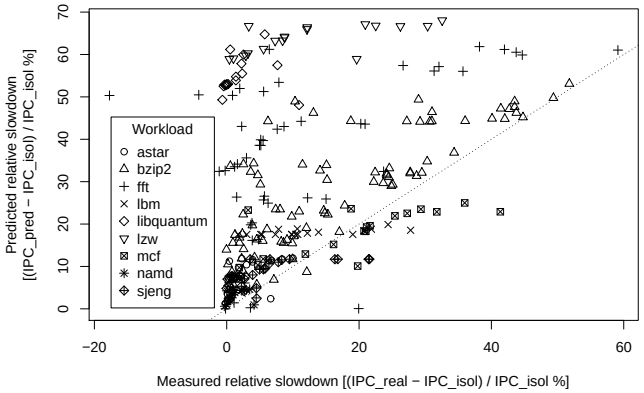

Figure 2: IPC slowdown prediction against IPC slowdown measurement, Stressmark profiles.

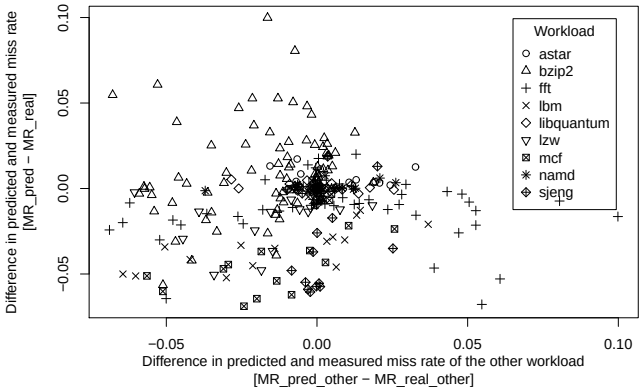

Figure 3: MR prediction error depending on the MR prediction error of the other workload in combination, Valgrind profiles.

The workloads most sensitive to parallel execution are some variants of fft and bzip2. With the Valgrind profiles, their slowdown is predicted with relatively acceptable error, however, for most of the other sensitive workloads, the slowdown prediction is too optimistic. Overall, the Valgrind profiles have a median prediction error of 3.5 %, inter-quartile range 1 % - 8.5 %, where the prediction error is calculated as $|IPC_{pred} - IPC_{real}|/IPC_{real}$. When considering only the upper quartile of workload combinations by sensitivity to paralel execution, we have 70 workloads whose slowdown exceds 19 %. For this group, the median prediction error is 14 %, inter-quartile range 7 % - 27 %.

With the Stressmark profiles and the workloads most sensitive to parallel execution, the slowdown prediction is somewhat pessimistic. Overall, the Stresmark profiles have a median prediction error of 9.6 %, inter-quartile range 4.7 % - 30 %. For the same group of sensitive workloads as above, the median prediction error is 10 %, inter-quartile range 6 % - 27 %. Importantly, the prediction is very pessimistic even for the least sensitive workloads.

The figures indicate that there are also workloads that appear to speed up in parallel execution. Although not intuitive, this is indeed possible, especially where a mostly writing workload executes in combination with a mostly reading workload. In this situation, the reading workload may take over part of the penalty for evicting dirty cache lines from the writing workload [2].

Compared to existing research, which rarely reports errors larger than units of percent, the results on Figures 1 and 2 might appear to be inaccurate. In the related work section, we explain why this is a misleading impression. To justify our claim, we have used the same inputs to perform validation with our implementation of the model in [13], and – although we can never safely exclude potential implementation issues – we have obtained similar modeling accuracy. The details are not included in the text due to lack of space, however, we publish both the sources and the data for interested readers.

4.2 Overview: Miss Rate

We continue with plotting the miss rate prediction error, $MR_{pred} - MR_{real}$, against the miss rate increase of the workload in concurrent execution, $MR_{real} - MR_{isol}$. The predictions based on the Valgrind profiles are in Figure 4, the predictions based on the Stressmark profiles are in Figure 5.

With the Valgrind profiles, the model appears to predict the miss rate with a resolution of about 0.05. The roughly linear cluster of points to the lower left suggests that increases in the miss rate below this resolution are not predicted by the model. We attribute these effects to the detailed behavior of the cache replacement policy.

With the Stressmark profiles, the model appears more pessimistic for less sensitive workloads. This effect is exemplified with lzw, whose Stressmark profile indicates almost 40% of accesses with stack distance just below the cache associativity (see Figure 8). Since these accesses would be very likely to change from hits to misses under concurrent execution, an inaccuracy in this part of the profile would explain the observed effect.

To assess whether both workloads in a workload combination suffer from the same miss rate prediction error, or whether the model favors one workload against the other, we plot the miss rate prediction error of one workload in a combination against the miss rate prediction error of the

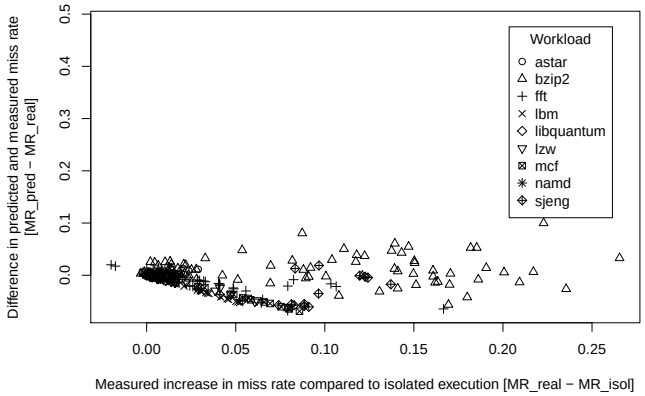

Figure 4: MR prediction error depending on MR increase, Valgrind profiles.

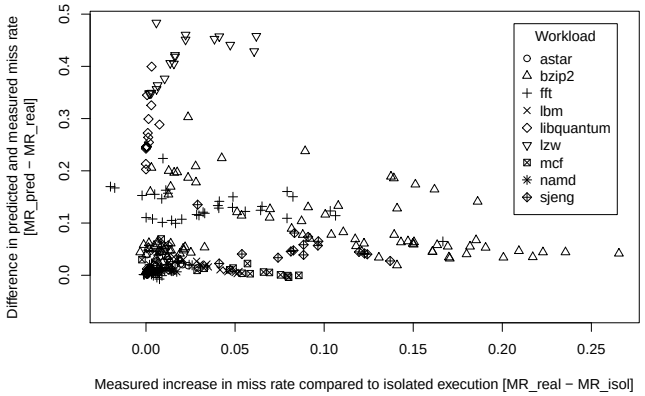

Figure 5: MR prediction error depending on MR increase, Stressmark profiles.

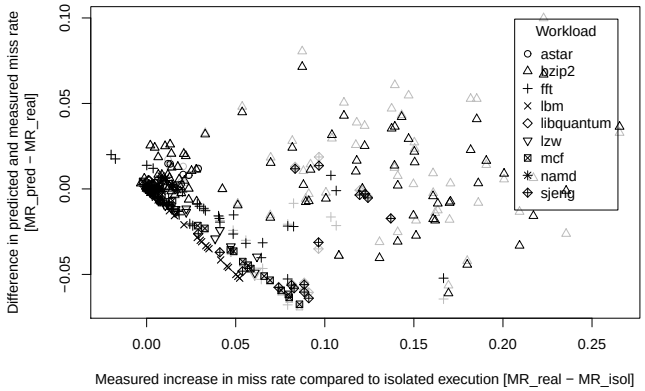

Figure 6: MR prediction error depending on MR increase, Valgrind profiles (grey points). Black points denote prediction with injected IPC_{real}.

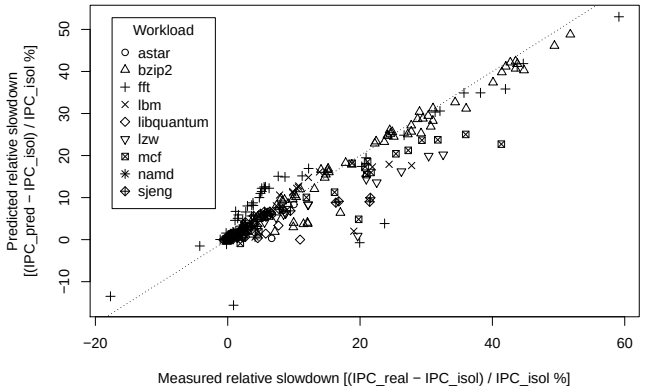

Figure 7: IPC slowdown prediction against IPC slowdown measurement, injected MR_{real}.

other workload. The results for Valgrind profiles on Figure 3 suggest that the error tends to impact both workloads equally, even though there are exceptions. Although not shown separately, the impact on the slowdown prediction is similar.

4.3 Miss Penalty Approximation

The cache model includes interaction between misses and penalties. Next, we investigate measurements that help isolate the effects of poor miss penalty approximation on the miss rate prediction. We do that by injecting IPC_{real} into the calculation where IPC_{pred} would otherwise be calculated.

There are two ways the injection can be performed. Either the iteration in the model is preserved and the measured instruction throughput is used as a seed, or the iteration is removed entirely. For the former option, there is no observable change in results, which confirms the iteration is not hindered by encountering local extremes. For the latter option, the results with Valgrind profiles are on Figure 6, with black points adjusted and grey points original. The difference is in fact relatively small.

4.4 Miss Rate Prediction

Analogously to the previous experiment, we investigate measurements that help isolate the effects of poor miss rate prediction. We do that by injecting MR_{real} into the calculation where MR_{pred} would otherwise be calculated, again removing the iteration entirely. The results on Figure 7 clearly indicate that an improved miss rate prediction would improve the modeling accuracy.

4.5 Workload Profile Stability

The previous experiments indicate the modeling accuracy is related more to the miss rate prediction than to the miss penalty approximation. The Valgrind profiles and the Stressmark profiles yield potentially different prediction results, and profile inaccuracies could explain some observed prediction errors. We therefore investigate the profiles themselves and the role of the assumption that the profiles are stable over time and similar across cache sets.

Our use of two workload profile collecting tools makes it possible to compare the profiles, Valgrind with Stressmark. While each tool has its drawbacks, and neither profile is therefore guaranteed to be correct, significant differences

should reveal potential profile accuracy issues. The comparison of the profiles is on Figure 8.

The comparison of the profiles indeed reveals significant differences between Valgrind and Stressmark. The tools inherently differ in their sensitivity to the assumptions of stability over time and similarity across sets – where Valgrind continuously tracks accesses to all sets, Stressmark must compete one set at a time. To investigate this difference, we have developed a synthetic workload that is able to mimic another workload given its stack distance histogram and mean number of cache accesses per instruction, while enforcing both stability over time and similarity across sets. For brevity, we refer to this synthetic workload as Emulator.

The goal of Emulator is to perform independent memory accesses in each cache set that satisfy a given stack distance profile. The difficult part is avoiding access to data controlling Emulator, since that would distort the memory access pattern. Even a pseudocode description of Emulator is rather involved, we therefore only present the basic ideas and invite interested readers to peruse the sources.

In its most basic form, Emulator would need to keep track of the LRU stack for each set, and, for each access, randomly pick a distance from the profile, find the corresponding address in the LRU stack, perform the access and update the LRU stack. To avoid accessing the LRU stack, Emulator prepares the sequence of addresses to access beforehand. To further reduce overhead, the sequence of addresses to access is stored as a chain of pointers at the very addresses to be accessed [5]. Except for locating the first item of the chain, it can thus be traversed without overhead.

Finally, Emulator does not treat each cache set independently. Instead, the sets are assigned and ordered randomly in 16 groups of equal size. Each group has its own chain of pointers that defines the sequence of addresses to access, touching each set of the group once in random order. When executing, Emulator maintains 16 pointers, one for each group, to traverse all sets.

This design reduces the size of data controlling Emulator that has to be read during each iteration to 16 pointers, which on our platform translates into an overhead of accessing 2 extra cache lines per 4096 useful accesses. While this comes at the cost of introducing regularity of accesses among the sets, the independence of accesses within each set is preserved. The relative sizes of the access pattern and the caches also ensure that all accesses spill from the private L1 cache to the shared L2 cache, as is required.

We submit the Valgrind profiles of the original workloads to Emulator to create synthetic workloads with the same stack distance profile that are stable over time and homogeneous across sets. We then collect the Stressmark profiles of the synthetic workloads and use these profiles to model the original workloads. The results are indeed more accurate – across all workloads, the median prediction error is 5.7 %, inter-quartile range 2.4 % - 13 %, across the group of sensitive workloads, the median prediction error is 8 %, inter-quartile range 4.5 % - 15 %. The details are on Figure 9.

Interestingly, the process is not just a more complicated way of updating the Stressmark profiles to match the Valgrind ones – in fact, the Stressmark profiles of the synthetic workloads are not entirely similar to the Valgrind profiles of the original workloads. This is shown on Figure 11. We explain the reason for this difference and the related modeling accuracy next.

4.6 Cache Replacement Policy

So far, we have put aside the issue of the cache replacement policy. Although the details of the policy are not available in vendor documentation, the hardware most likely implements pseudo LRU in both the private L1 cache and the shared L2 cache. This actually impacts the workload profile collecting tools:

- When Valgrind collects a profile, it has to calculate when accesses spill from L1 to L2. Any deviation from strict LRU in L1 therefore impacts the profile, while deviations in L2 do not. In precise terms, when Valgrind collects a profile, it means that *the measured workload would have this profile in L2 on a platform where L1 uses strict LRU.*

- When Stressmark collects a profile, it has to calculate which position in the stack distance histogram corresponds to L2 misses observed at particular L2 occupancy. Any deviation from strict LRU in L2 therefore impacts the profile, while deviations in L1 do not. In precise terms, when Stressmark collects a profile, it means that *a workload with this profile executing on a platform where L2 uses strict LRU would cause the same number of L2 misses as the measured workload does on a platform where both L1 and L2 use pseudo LRU.*

Our cache model calculates the miss rates under the assumption of strict LRU. The validation compares these with miss rates measured on a platform with pseudo LRU. The mismatch between strict LRU and pseudo LRU exhibits itself differently with the two workload profile collecting tools:

- The Valgrind profile describes how the profiled workload would access L2 if L1 used strict LRU. Then, the model calculates what the miss rate would be if multiple profiled workloads shared an L2 with strict LRU. When using the Valgrind profiles, the model therefore calculates a miss rate on a hypothetical platform with strict LRU.

- The Stressmark profile describes how a hypothetical workload would have to access L2 on a platform with strict LRU to achieve the same miss rates against Stressmark as the profiled workload does on the platform with pseudo LRU. Then, the model calculates what the miss rate would be if multiple hypothetical workloads shared an L2 with strict LRU. When using the Stressmark profiles, the model therefore calculates a miss rate for the hypothetical workloads, which should achieve a similar miss rate on a platform with strict LRU as the profiled workloads do on the platform with pseudo LRU.

This explains the results from Section 4.5, where using the Stressmark profiles of the Emulator workloads, which in turn simulate the Valgrind profiles of the original workloads, yields better accuracy than using either the Stressmark profiles or the Valgrind profiles of the original workloads directly. When collecting the profiles of the original workloads, Valgrind copes better than Stressmark with workloads

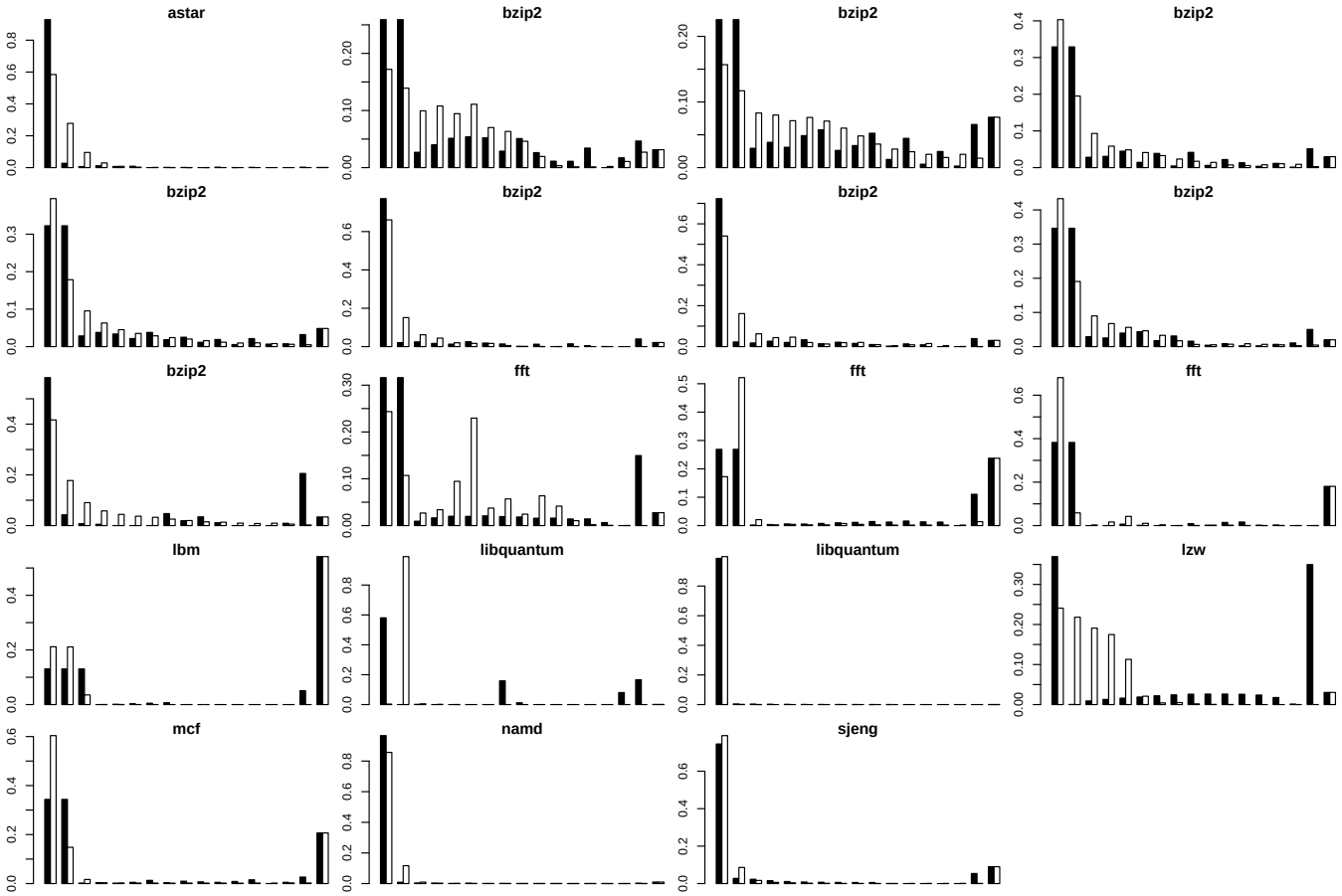

Figure 8: Stressmark profiles (black) and Valgrind profiles (white) of the original workloads. Distances grow from left to right. Multiple profiles of the same name denote the same benchmark with different choices of inputs.

whose profiles change over time and differ across sets. Simulating the original workloads with Emulator ensures stability over time and homogeneity across sets, important for the subsequent use of Stressmark. Finally, Stressmark calculates what profiles would cause the observed cache misses in a cache with strict LRU. These profiles fit the cache model, which estimates the cache misses under the assumption of strict LRU, better than the Valgrind profiles.

Finally, we illustrate the modeling accuracy in a situation where both the workload profile stability issues and the cache replacement policy issues are minimized. Modifying the previous experiment, we submit the Valgrind profiles of the original workloads to Emulator to create synthetic workloads, we then collect the Stressmark profiles of the synthetic workloads, and we use these profiles to model the synthetic workloads. The results, plotting the relative slowdown prediction, $(IPC_{isol} - IPC_{pred})/IPC_{isol}$, against the relative slowdown measurement, $(IPC_{isol} - IPC_{real})/IPC_{isol}$, are on Figure 10. Across all workloads, the median prediction error is 3.2 %, inter-quartile range 0.9 % - 6.1 %, across the group of sensitive workloads, the median prediction error is 3.8 %, inter-quartile range 1.6 % - 5.9 %. Since the results are for the synthetic workloads, they are no longer useful for modeling the original workloads, but they illustrate the accuracy achievable for workloads with stable profiles.

To conclude, our evaluation has shown the modeling accuracy achievable when the various simplifying assumptions about the workloads and the platform are satisfied only to a degree common in realistic settings. Furthermore, we have shown how enforcing the validity of the individual assumptions contributes to the modeling accuracy. The results indicate that the modeling accuracy is sufficient for application in software performance modeling, especially for workloads sensitive to cache sharing.

5. RELATED WORK

Of the models that estimate the cache miss ratio based on the workload description, we focus on models that rely on access distance profiles for workload description and are capable of handling combinations of workloads. More complex modeling approaches, such as full system simulation, belong to different application domain due to potentially large overhead. Less complex modeling approaches, such as overhead interpolation, were investigated in [6]. For each related model, we consider the internal structure and the evaluation approach separately.

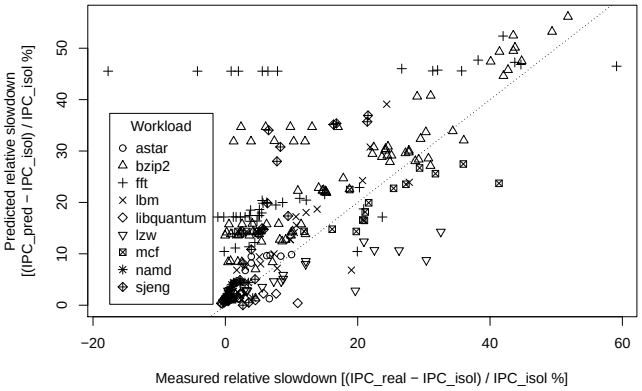

Figure 9: IPC slowdown prediction against IPC slowdown measurement of the *original workloads* with profiles collected by Stressmark on *synthetic workloads*.

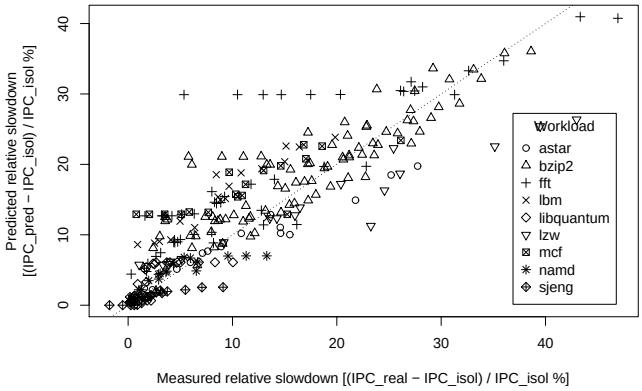

Figure 10: IPC slowdown prediction against IPC slowdown measurement of the *synthetic workloads* with profiles collected by Stressmark (also on synthetic workloads).

In our review, we sometimes have to adjust the discrepancies in terminology. Notably, when talking about the distance between consecutive accesses to the same cache line, we use the term *reuse distance profile* when the distance is expressed in memory accesses, and the term *stack distance profile* when the distance is expressed in unique cache lines.

5.1 Related Model Structure

The statistical model by Chandra et al. [8] is the oldest of the related models listed here. The computational approach follows the general outline of first estimating the time between two consecutive accesses to the same cache line by one workload, then calculating the distribution of accesses to unique cache lines in that time interval by the other workload, and finally determining what percentage of former hits will be turned to misses by those additional accesses.

The most notable difference between their model and ours is in the required inputs. Their model requires not only the stack distance profile, but also a distribution of the intermediate access counts for each distance in the stack distance profile, together denoted as circular sequence profile. In some points of the calculation, parts of the circular sequence profile are averaged over, the contribution of this additional information to accuracy is therefore not clear. The required inputs are also different from the provided outputs, preventing hierarchical model composition.

As another notable difference, their model does not consider the mutual influence between the competing workloads. This is related to the fact that their model only deals with the cache misses and not the timing penalties, and is thus unable to determine how much the additional cache misses due to sharing change the relative speed of the workloads.

Both our model and the model in [8] assume limited associativity. This assumption is important for their model since it uses a recursive formula to calculate the number of accesses to unique cache lines in a given time interval. Because the complexity of the formula grows exponentially with the interval length, the calculation over the entire cache, rather than over a single cache set, is not feasible in their model.

The throughput model by Chen et al. [9] includes a probabilistic cache contention model that extends the model in [8] in two major ways:

- The model compensates for situations where, due to a relatively large number of sets, the competing workloads are not likely to access the same sets in short time intervals. To do that, the model replaces the recursive formula that calculates the number of accesses to unique cache lines in a given time interval with a calculation based on additional input information.
- The model estimates how much the additional cache misses due to sharing change the relative speed of the workloads. Unlike our model, which iterates until convergence, the model in [9] only uses two repetitions.

In addition to the inputs required by the model in [8], the model in [9] also requires the average number of accessed sets and the stack distance profile for accesses in a given time interval, both for multiple groups of time intervals. Again, the required inputs are different from the provided outputs, preventing hierarchical model composition.

Both our model and the model in [9] assume limited associativity. Using the additional input information, their model better describes the situations where the workloads are not likely to compete for the same sets, however, their model still expects that the stack distance profiles are the same for all sets. Given that the choice of the cache sets is typically not controlled by the workload, this expectation appears reasonable.

The model by Xu et al. [13] employs a computational approach based on the properties of a steady state, where the competing workloads occupy their fixed shares of the cache. First, the relationship between the length of a time interval and the number of unique cache lines accessed in the interval is expressed using a recursive formula similar to that in [8]. Next, it is observed that all the competing workloads take the same time to entirely occupy their cache shares, and that the sum of the cache shares is the cache size. This gives a set of equations that can be solved to yield the sizes of the cache shares.

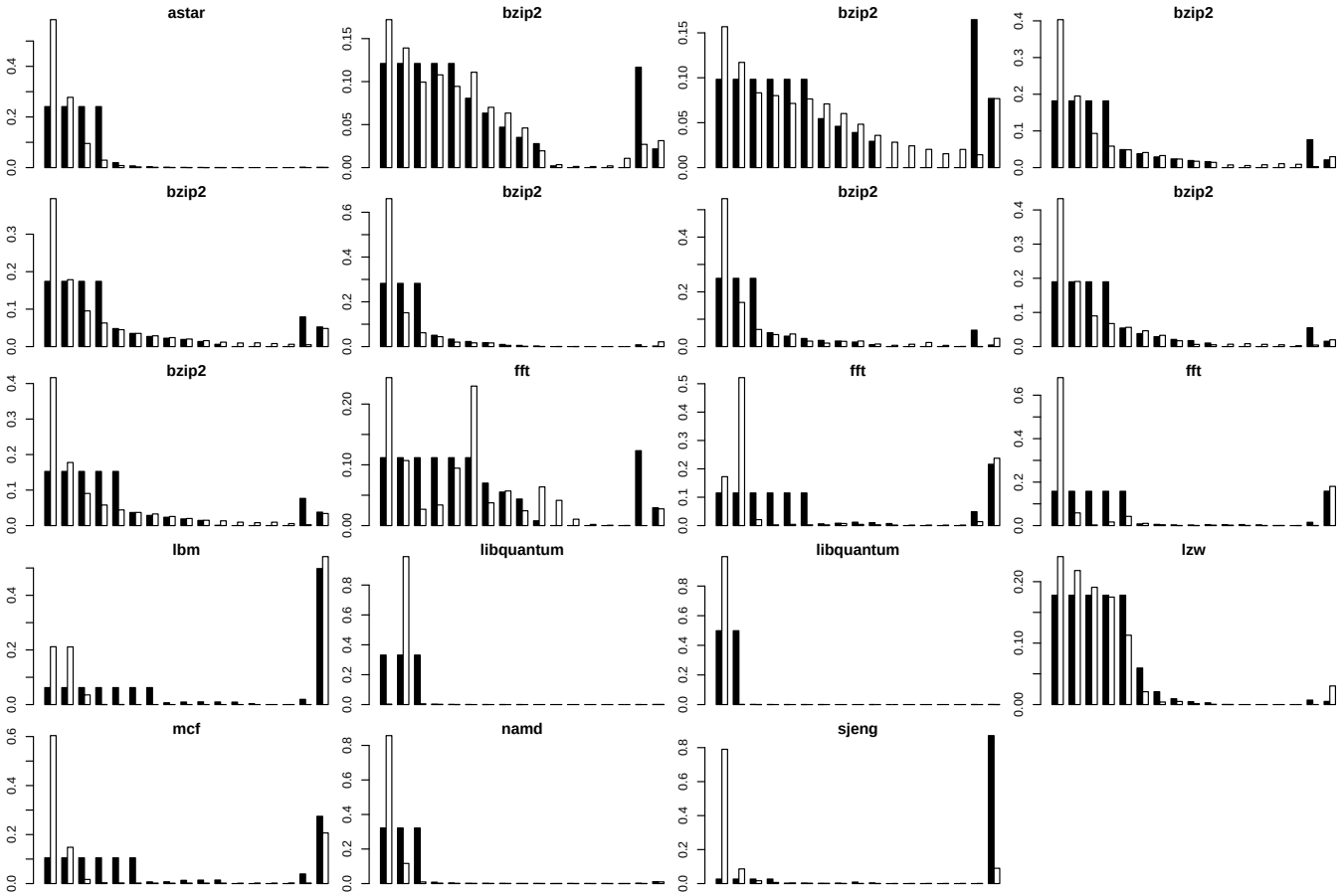

Figure 11: Stressmark profiles of the synthetic workloads (black) and Valgrind profiles of the original workloads (white). Distances grow from left to right.

Both our model and the model in [13] assume limited associativity, expecting that the stack distance profiles are the same for all sets. Their model also requires the same inputs as our model. Again, the required inputs are different from the provided outputs, preventing hierarchical model composition.

The model by Eklov et al. [10] relies on a statistical approach that estimates a stack distance profile from a reuse distance profile. First, scaled versions of the reuse distance profiles of the competing workloads are expressed as functions of the instruction throughputs. The scaled profiles are then merged into a single reuse distance profile. Next, a merged stack distance profile is estimated from the merged reuse distance profile. The merged stack distance profile is then split into the stack distance profiles of the competing workloads. Finally, the instruction throughputs of the competing workloads are expressed as functions of the stack distance profiles. This gives a set of equations that can be solved to yield the instruction throughputs.

Unlike our model, the model in [10] assumes full associativity. The impact of this difference on accuracy depends on the workloads, and is generally difficult to quantify.

The model in [10] requires similar inputs as our model, namely the reuse distance profiles where we require the stack distance profiles. Although technically different, the inputs are of similar nature and scale. Internally, their model performs profile conversions, merges and splits that involve approximations and possibly rounding,[4] which may impact accuracy. Although direct comparison is not possible due to technical differences, we believe that similar impact on our model is smaller, since it uses no rounding and fewer assumptions.

Finally, the required inputs are different from the provided outputs, but since the merged reuse distance profile is considered as an interim information in the model, it is possible that relatively minor modification would remove this obstacle to hierarchical model composition.

To summarize, we show how our model and the reviewed models differ in features such as the inputs and the outputs, the computational approach, the treatment of associativity, and the rounding and assumptions that impact accuracy. All of the models differ in some of the features, both compared with our model and compared among themselves.

5.2 Related Model Evaluation

Existing research also differs in how the modeling accuracy is evaluated, so much so that this basically makes the

[4]The issue of rounding is not discussed explicitly but the illustrations suggest it is likely done.

accuracy results incomparable. We explain some of the reasons by looking at how the model inputs are obtained and how the experimental conditions are controlled.

As far as the model inputs are concerned, the evaluations in [8, 9, 10] rely on simulators to obtain the workload profiles.[5] The use of simulators typically gives precise workload profiles, however, simulators often introduce overhead that is not acceptable in realistic settings, where the models would therefore work with potentially less precise inputs.

The evaluation in [13] determines the workload profiles that form the model inputs using experimental measurements on real hardware. The experimental measurements still carry some overhead, and the workload profiles are constructed from the measurements with some approximations.

Our evaluation is closer to [13] than the other reviewed evaluations since it also obtains model inputs using experimental measurements on real hardware.

Once the required inputs are available, the evaluation proceeds by independently modeling and measuring the effects of cache sharing on selected workload combinations. The results of modeling and measurement are compared to determine the modeling accuracy.

The workload combinations are almost always limited to pairs of workloads, except for [9], which uses up to 32 workloads of up to 4 different types. The workloads are almost always selected from the SPEC CPU benchmark suite, but the benchmark version and the configured input size varies. Typically, about tens of different workload combinations are used. Our evaluation roughly fits these general parameters – the somewhat limited choice from the SPEC CPU benchmark suite is compensated with a larger choice of the configured input sizes and the addition of other benchmarks, the total number of workload combinations we use is higher.

For a particular workload combination, how the combination is executed also matters. The workloads are known to have multiple phases with different behavior, and details such as synchronization and repetition can influence what phases end up competing for the cache:

- The experiments in [8] run the workloads once and stop the measurement when the shorter workload finishes.
- The experiments in [9] run a block of 400 million instructions at the start of the workloads.
- The experiments in [13] are described too vaguely to determine how the workloads were executed. The text suggests that only a single phase was considered for each benchmark, however, it does not say how the phases were identified and synchronized.
- The experiments in [10] run a block of 100 million accesses a constant distance from the start of the workloads.

Our experiments separate the initialization and the calculation parts of the workloads and run the calculation parts (which can contain multiple phases) repeatedly without synchronization. Compared to the reviewed evaluations, this increases the opportunity for different combinations of phases to compete for the cache, and reduces the risk of observing artefacts due to strict workload synchronization.

The evaluations in [8, 9, 10] measure the effects of cache sharing on a simulator. Only the evaluation in [9] complements these with measurements on comparable real hardware, the other two evaluations use simulator configurations that potentially limit realism – in [8], the simulator uses special set indexing to ensure uniform set utilization, in [10], the simulator uses in-order processing that can stabilize the miss penalty. Also, it is not clear whether the simulators implement the detailed behavior of the cache replacement policy, which can impact the modeling accuracy significantly.

Of the reviewed evaluations, only the one in [13] uses real hardware, as does our evaluation.

Finally, the reviewed evaluations differ in the way the modeling accuracy is reported. Using MR for miss rate and IPC for instruction throughput, and denoting isolated, modeled and measured values respectively with $isol$, $pred$ and $real$, we have:

- In [8], the error is defined as the miss rate prediction error scaled against the measured miss rate,
$$error = \frac{MR_{pred} - MR_{real}}{MR_{real}}$$
- In [9], both the miss rate prediction error and the instruction throughput prediction error are reported, but neither is defined. There are hints suggesting the calculation is similar to the previous case.
- In [13], both the miss rate prediction error and the instruction throughput prediction error are reported, but neither is defined and there are no hints suggesting what the calculation looks like.
- In [10], the instruction throughput prediction error scaled against the isolated instruction throughput is reported,
$$error = \frac{IPC_{pred} - IPC_{real}}{IPC_{isol}}$$

The problem with reporting the relative miss rate prediction error, also discussed in [10], is that in low miss rate workloads, a large relative error may translate into a small performance impact. When the ultimate purpose of modeling is assessing performance, the relative miss rate prediction error is therefore not a good metric. We believe that multiple error metrics should be used together to help highlight different accuracy aspects, as we do in our evaluation.

To summarize, we show that the accuracy results of the reviewed evaluations are mutually incomparable. Additionally, no reviewed evaluation is documented sufficiently for constructing a comparable evaluation of our model. Lacking the option to produce comparable evaluation results, we have decided to strive for realistic evaluation settings, which give practically relevant picture of the achievable accuracy. We also take care to publish complete documentation – not just this text, but also complete tool sources and complete data files – to permit comparable evaluations against our model.

[5]The evaluation in [10] also explores the option of profile sampling.

6. CONCLUSION

The paper makes multiple contributions to memory cache models, motivated by the aplication in software performance modeling, but also useful in other contexts. First, we have presented a new cache model from the family of models that take the stack distance profile of multiple workloads as input and estimate the cache miss ratio and the instruction throughput as output. Among the properties particular to our model is the ability to calculate the stack distance profile of a combination of workloads. Other properties of our model were contrasted with related work as well.

We pay a special attention to evaluating the modeling accuracy in realistic settings, in particular when the competing workloads execute on real hardware with no synchronization between execution phases. We demonstrate the achievable accuracy with a series of measurements that not only give the overall results, but also analyze the various accuracy factors. We show that in our model, the miss penalty prediction is somewhat less influential than the miss rate prediction. Among the important factors distorting the miss rate prediction are the workload profile stability and the differences between the assumed strict LRU policy and the implemented pseudo LRU policy.

To isolate the individual accuracy factors, we have developed a specialized workload that is capable of mimicking a given stack distance profile. We use the workload to demonstrate a subtle relationship between the tools used to collect the stack distance profiles and the modeling accuracy – to our knowledge, this relationship has not been identified in existing research.

Finally, we identify multiple issues that make the modeling accuracy evaluation results in existing research mutually incomparable. In our evaluation, we carefully avoid the identified issues and publish complete tool sources and complete data files to permit comparable evaluations against our model. The sources and the data files are available at http://d3s.mff.cuni.cz/benchmark.

7. ACKNOWLEDGMENTS

We would like to thank Peter F. Sweeney for comments that have helped us improve the final revision of this paper.

This work was partially supported by the Czech Science Foundation projects GACR P202/10/J042 and GACR 201/09/H057, and by the Charles University project SVV-2011-263312.

8. REFERENCES

[1] SPEC CPU2006. http://www.spec.org/cpu2006.

[2] V. Babka, L. Bulej, M. Děcký, J. Kraft, P. Libič, L. Marek, C. Seceleanu, and P. Tůma. Resource usage modeling, Q-ImPrESS deliverable 3.3. http://www.q-impress.eu, February 2009.

[3] V. Babka and L. Marek. Frameworks for measuring effects of resource sharing. http://d3s.mff.cuni.cz/benchmark, 2010.

[4] V. Babka, L. Marek, and P. Tůma. When misses differ: Investigating impact of cache misses on observed performance. In *Proceedings of ICPADS 2009*, pages 112–119, Shenzhen, China, December 2009. IEEE.

[5] V. Babka and P. Tůma. Investigating cache parameters of x86 family processors. In *Proceedings of the SPEC Benchmark Workshop 2009*, volume 5419 of *LNCS*, pages 77–96. Springer, January 2009.

[6] V. Babka and P. Tůma. Can linear approximation improve performance prediction? In *Proceedings of EPEW 2011*, volume 6977 of *LNCS*, pages 250–264. Springer, October 2011.

[7] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *IEEE Trans. Soft. Eng.*, 30(5):295–310, 2004.

[8] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of HPCA 2005*, pages 340–351. IEEE, 2005.

[9] X. E. Chen and T. M. Aamodt. A first-order fine-grained multithreaded throughput model. In *Proceedings of HPCA 2009*, pages 329–340. IEEE, 2009.

[10] D. Eklov, D. Black-Schaffer, and E. Hagersten. Fast modeling of shared caches in multicore systems. In *Proceedings of HiPEAC 2011*, pages 147–157. ACM, 2011.

[11] C. M. Grinstead and J. L. Snell. *Introduction to Probability*. American Mathematical Society, 1997.

[12] J. Seward et al. Valgrind. http://www.valgrind.org. Version 3.6.1.

[13] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. Cache contention and application performance prediction for multi-core systems. In *Proceedings of ISPASS 2010*, pages 76–86. IEEE, 2010.