Fast and Precise Cache Performance Estimation for Out-Of-Order Execution

Douma, R.J.; Altmeyer, S.J.; Pimentel, A.D.

Published in:

Citation for published version (APA):
Fast and Precise Cache Performance Estimation for Out-Of-Order Execution

Roeland J. Douma, Sebastian Altmeyer, Andy D. Pimentel
Computer Systems Architecture Group, University of Amsterdam, Netherlands
{r.j.douma, altmeyer, a.d.pimentel}@uva.nl

Abstract—Design space exploration (DSE) is a key ingredient of system-level design, enabling designers to quickly prune the set of possible designs and determine, e.g., the number of the processing cores, the mapping of application tasks to cores, and the core configuration such as the cache organization. High-level performance estimation is a principle component of any system-level DSE: it has to be fast and sufficiently precise. Modern out-of-order architectures with caches pose a significant problem to this performance estimation process, as no simple one-to-one mapping of the number of cache misses and resulting cycle time exists.

We present a high-level cache performance-estimation framework for out-of-order processors. Evaluation shows that our prediction method is on average 15 times faster than cycle-accurate simulation, while our estimates only show an average error of below 3.5%, reduce the pessimism of a naive high-level performance estimation by around 66%, and still maintain a high fidelity. Our approach thus enables quick yet accurate performance estimation and extends the applicability of system-level DSE to out-of-order processors with caches.

I. INTRODUCTION

As the complexity of modern embedded systems increases, the development of these systems becomes more and more challenging. Design space exploration (DSE) methods are thus used to cope with the increasing complexity, to speed-up the development process and thus, to reduce the time-to-market. Automated exploration tools quickly prune the set of possible architectures and select the set of (pareto) optimal candidate architectures according to multiple objectives, such as performance, costs, size and power consumption. The system specification automatically selected by the exploration tools range from high-level information such as the number and the type of the processing cores or the mapping of application tasks to cores to more low-level specifications such as the cache hierarchy configuration. As the pruning of the set of candidate architectures is already required at an early development stage, a high-level performance estimation of the target application is necessary. Such an estimation has to fulfill two opposing requirements: it has to be fast (to evaluate a sufficient set of candidates) and precise (to select the correct candidates). Caches are nowadays an integral component of most embedded systems where performance is critical. On the downside, caches require a significant amount of area on the chip and can increase the system’s energy consumption[1]. Choosing the right cache size and right cache parameters is thus a paramount task of any DSE, and the aforementioned high-level performance estimation must account for caches.

In case of processors with in-order execution, it is sufficient to compute the number of hits and misses as a simple one-to-one mapping from misses to cycles exist. To this end, an abundance of cache miss estimation techniques, such as stack-distance histograms and cache-miss equations, exist and fast and accurate performance estimation is available.

In contrast, modern out-of-order architectures with caches pose a significant problem to the performance estimation, as no simple one-to-one mapping of the number of cache misses and cycles exist: the actual address trace can change depending on the selected cache configuration and pipelining of outstanding memory requests change the average memory latency. A naive estimation of the number of cycles to execute an application purely based on the number of hits and misses (as valid for in-order processors) thus provides unreliable results. Hybrid approaches using source-code instrumentation and relatively fast source-level simulation exist, but typically are limited to in-order execution. Consequently, the currently available performance estimation techniques are either precise (based on slow cycle-accurate simulation) or fast (based on hit/miss ratios), but not both at the same time.

We present a cache performance estimation framework for out-of-order processors. The framework is split in two phases: a setup phase and a DSE-phase. In the setup-phase, we perform cycle-accurate simulations for two cache configurations, extract stack-distance histograms and the memory-overlap of the target application. This information serves as input for the DSE-phase which is then used to quickly estimate the performance of the target application for the candidate cache configurations. This two-step approach thus enables quick yet accurate performance estimation to be used for system-level DSE.

Our framework improves upon related work in the following aspects: The framework is

• fast during the actual DSE using a two-phase approach,
• precise as it explicitly models the effect of out-of-order execution on the memory performance, and
• agnostic of the target architecture as it only relies on the existence of a cycle-accurate simulator, but does not require an in-depth analysis of the hardware.

The paper is structured as follows: Section II reviews the related work and Section III introduces the required terminology and notation. In Section IV we present the performance-estimation framework and evaluate the proposed techniques in Section V. Section VI concludes the paper.

II. RELATED WORK

The estimation of cache performance has been an intensive subject of research in the past decades. We distinguish two types of approaches: static analysis and simulation based approaches. The two most prominent examples of the former type are the cache-miss equations by Ghosh et al. [2], and the stack-distance computation [3]. Cache-miss equations provide a high-level representation of the cache behaviour. They are used in compiler frameworks to estimate the effect of optimizations. Estimation techniques based on the stack distance [3] provide
more accuracy, but suffer from the memory overhead. Several methods have been proposed to enable an efficient computation \(^4\)–\(^7\) or approximation \(^8\) of the stack distance. Stack distances and cache miss equations, however, only provide the number of misses and hits for different cache configurations. While this is sufficient for simple in-order architectures that stall in case of a cache miss, the performance of out-of-order architectures is not sufficiently described by this metric alone.

Simulation techniques provide an alternative to the static analyses, but are either restricted to in-order processors \(^9\)–\(^11\) or to the simulation of only one cache configuration at a time, thus resulting in an unacceptable execution time \(^12\).

A different set of performance estimation techniques is source-level simulation based on instrumentation \(^13\)–\(^15\). The source code is instrumented with memory access annotations and timing information, enabling the performance estimation via source-code simulations instead of using a cycle-accurate low-level simulator. These approaches thus still require explicit cache simulation and do typically not take the effect of out-of-order execution behavior. However, this approach still requires explicit cache simulation.

To the best of our knowledge, no fast and accurate high-level performance estimation for out-of-order processors with caches exist. DSE techniques thus either do not support these systems or have to restrict precise performance estimation to a small subset of the design space \(^17\).

III. BACKGROUND AND NOTATION

In this section, we introduce the necessary notation and concepts needed in the remainder of the paper. We assume a target architecture with disjoint instruction and data caches that are both connected to main memory via a shared bus.

A. Caches and Address Traces

We concentrate on the following set-associative caches with LRU replacement: i.e., caches that are partitioned into \(s\) cache sets, where each memory block of size \(l\) (i.e., cache line size is \(l\)) maps to exactly one of the cache sets. Each cache set in turn may contain up to \(k\) different memory blocks at once, where \(k\) is referred to as the associativity of the cache. The cache size is thus given by \(l \cdot s \cdot k\).

We will later on use the concept of a perfect cache, which denotes a fully associative cache that is large enough to store the complete data of a task/program. A perfect cache thus only exhibits cold misses, but no conflict or capacity misses \(^5\). Note that direct-mapped and fully associative caches are special cases of set-associative caches with \(k = 1\) or \(s = 1\), respectively.

A cache configuration or cache setup \(\zeta\) is a tuple \((k, s)\) consisting of the associativity and the number of cache sets. For the sake of simplicity, we assume a fixed line size \(l\) throughout the paper. The set of all cache setups is denoted by \(\mathcal{C}\) with the perfect cache \(pc\) as special instance of a cache setup, i.e.: \(\mathcal{C} = \{(k, s)| (k, s) \in \mathbb{N}\}\) \(\cup\) \(\{pc\}\). Architectural restrictions or a pre-selection of candidate cache setups allows us to reduce the complete set of cache setups \(\mathcal{C}\) and to focus only on a subset \(\mathcal{C}' \subset \mathcal{C}\).

A cache address trace \(T\) of size \(n\) is an ordered sequence \([m_1, \ldots, m_n]\) of memory blocks \(m_i \in \mathcal{M}\), where \(\mathcal{M}\) is the set of all memory blocks. The set of all traces is given by \(\mathcal{T}\). The stack distance \(sd_s\) (\(s\) denotes the number of sets) \(^3\) is the number of distinct memory elements mapping to cache set \(s'\) accessed in between an access to a memory element that also maps to \(s\) and the previous access to the same memory element, with \(\infty\) denoting that there is no prior access:

\[
sd_s(m_i, m_1, \ldots m_{l-1}) = \begin{cases} \{m_j| i < j < l \land cs(m_i) = cs(m_j)\} & \text{if } m_i \neq m_j \\ \infty & \text{else} \end{cases}
\]

where \(cs(m)\) denotes the cache set to which memory element \(m\) maps. The condition \(cs(m_i) = cs(m_j)\) thus ensures that the stack distance of element \(m_i\) only considers memory elements that compete for the same cache set as \(m_i\).

Table I summarizes the definitions used throughout the paper. To simplify the notation, we omit the subscripts whenever the parameters are sufficiently defined by the context.

B. Stack Distance Histogram Computation

To derive the number of cache misses, we will later use stack distance histograms \(^3\)–\(^7\) within our performance estimation framework. These stack distance histograms only depend on the number of cache sets and the cache line size (which we assume to be fixed). It is therefore sufficient to compute one stack distance histogram per distinct number of cache sets, irrespective of the associativity of the cache configuration. The computation of several stack distance histograms for different numbers of cache sets can be done in parallel.

Consider for example the following address trace \(A, B, B', A', B'', A''\) \(A, B, A', B', B, A''\)
cold misses

where we assume that all addresses map to distinct cache lines. The first 6 accesses are thus cold misses in any cache setup and have a stack distance of \(\infty\).

In a fully-associative cache, three accesses have a stack distance of 5 \((A, B, B')\), and two accesses have a stack distance of 4 \((A', A'')\). Only the last access to \(B\) has a stack distance of 2. In case of the set-associative cache, we assume that all references to \(A, A', A''\) map to the same cache set, but to distinct cache lines. The same holds for \(B, B', B''\). Moreover, we assume that \(A, B\) all map to distinct cache sets. Thus, to determine e.g., the stack distance for \(A\), only accesses to \(A, A', A''\) need to be taken into account. The number of cold misses is unchanged, five accesses have a stack distance of 2 \((A, B, A', B'', A'')\). The last access to \(B\) has a stack distance of 1. This example shows that the stack distance can significantly change if the cache setup changes.

IV. Cache Performance Estimation

System-level DSE requires fast, yet precise estimation of the execution time of the target application on the candidate architecture. Exhaustive evaluation of all cache configurations
using simulation is thus not an option; the simulation runs would consume more time than budgeted for the complete DSE.

We propose a method that relies on a minimal number of cache simulations needed to predict the performance of all candidate cache setups from the set \( C \), and a limited number of trace analyses. We divide the set of candidate setups into \( q \) congruent sets \( C_s \), where \( q \) is the number of distinct numbers of sets within \( C \):

\[
q = |\{s| (\_s) \in C\}|
\]

(2)

\( C_s \) contains all candidate cache setups with \( s \) cache sets:

\[
C_s = \{\zeta| \zeta = (\_s) \in C\}
\]

(3)

Instead of performing \( |C| \) simulations, our approach relies on the results of only 2 simulations in total (one to extract the address trace and one to estimate the memory overlap, a performance metric explained in detail in Section IV-B) and \( q \) trace analyses, which can be performed in parallel.

The first simulation assumes a perfect cache. This simulation provides us with the number of cycles, \( \text{cyc}_{pc} \), and the address trace \( T \). The second simulation is performed for a reference setup \( \zeta' \in C \), as explained later on.

The \( q \) trace analyses are needed to compute a histogram of the stack distances for a given \( s \):

\[
H_s(x) : \mathbb{N}^\infty \rightarrow \mathbb{N}
\]

(4)

Using the histograms, the number of cold misses for cache setup \( \zeta = (k,s) \) is given by:

\[
\text{miss}_{\zeta_{cold}} = H_s(\infty)
\]

(5)

The number of conflict or capacity misses by:

\[
\text{miss}_{\zeta_{warm}} = \sum_{k' \geq k \land k' \neq \infty} H_s(k')
\]

(6)

And the total number of misses is given by:

\[
\text{miss}_\zeta = \text{miss}_{\zeta_{cold}} + \text{miss}_{\zeta_{warm}}
\]

(7)

A. In-Order Execution

In an in-order processor, a cache miss means that the processor stalls until this cache line is retrieved from main memory. This means that the number of cycles required for a given cache setup can be approximated as follows:

\[
\text{cyc}_\zeta = \text{cyc}_{pc} + \text{miss}_{\zeta_{warm}} \cdot \gamma
\]

(8)

Where \( \gamma \) is the nominal memory latency of the system.

For the sake of simplicity, we assume unrestricted and immediate access to all potentially shared resources such as a bus and only concentrate on the execution cycles of the target application in isolation. Even though this assumption may be considered a strong restriction, it allows us to focus on the problem at hand and is common in the related work.

B. Out-of-Order Execution

Out-of-order processors do not stall in case of a cache miss, but execute other instructions while waiting for data. To fully exploit the advantages of out-of-order execution many processors also allow multiple outstanding memory requests to be handled simultaneously. Consequently, the effective memory latency \( \hat{\gamma} \) is often significantly smaller than the actual time to transport data from main memory to the cache; and the number of cache misses alone is not sufficient to estimate the performance anymore.

1) Illustrated Example: We illustrate the effect of out-of-order execution on the memory latency with an example depicted in Figure 1. We assume two cache configurations: Configuration 1 is a direct mapped cache with 2 sets; Configuration 2 has an associativity of 2 and also 2 sets. The second cache is thus twice as large as the first one.

We consider the address given in Section III-B and we assume again that all letters map to pairwise distinct cache sets, but all the accented letters map to different cache lines (but the same cache set). The cache is initially cold, i.e., empty, which means that the first 6 accesses in this trace are all misses for all cache configurations. The trace with corresponding hits, misses and cycle counts is shown in Figure 1.

The memory latency is 100 cycles and two outstanding memory requests can be overlaid with a combined memory latency of 150 cycles (instead of 200 for an in-order processor). Note that this is an oversimplification purely for the purpose of the example. For the sake of simplicity, we also assume that if two memory requests overlap they have to finish before another request can start.

Although Figure 1 shows the entire address trace, for this example we will only focus on the final part of the trace, since the first 6 accesses are all cold misses. The processor with the direct mapped cache has 6 cache misses and spends a total of 450 cycles waiting on memory. The average memory latency is thus 75 cycles. The processor with the 2-way set associative cache has only 5 cache misses but waits for a total of 400 cycles for data. The average memory latency is thus 80 cycles.

The example illustrates the imprecision of Eq. (8) when applied to out-of-order processors. The nominal latency \( \gamma \) is 100 cycles while we observe values of 75 and 80 cycles instead. The naive estimation of the execution cycles is thus not valid for DSE. Instead of the simple one-to-one mapping of the number of misses and the execution time, the average memory latency depends on the actual cache configuration.

2) Effective Memory Latency: As Eq. (8) is restricted to in-order processors, we propose an adapted version for out-of-order execution. We multiply the number of warm misses \( \text{miss}_{\zeta_{warm}} \) of cache configuration \( \zeta \) with the effective memory latency \( \hat{\gamma} \) instead of the nominal latency \( \gamma \):

\[
\text{cyc}_\zeta = \text{cyc}_{pc} + \text{miss}_{\zeta_{warm}} \cdot \hat{\gamma}
\]

(9)

Unfortunately, the effective memory latency is not constant, but depends on the cache configuration as we have seen in the example.

In the following, we show how we can compute an approximation of execution cycles \( \text{cyc}_\zeta \) for cache configuration \( \zeta \) based on the memory overlap, a metric which we compute using a reference cache configuration \( \zeta' \).

We first compute the miss-rate of the warm misses for the reference configuration \( \zeta' \):

\[
\text{miss}_{\zeta_{\text{warm}}} = \frac{(\text{miss}_{\zeta_{\text{warm}}})}{n}
\]

(10)

where \( n \) is the size of the address trace, i.e., the number of memory accesses, and the average number of cycles per
The design space, i.e., the set of candidate cache configurations, consists of two phases, the setup and the DSE phase, as shown in Figure 2. Within the setup phase, we perform the two simulations of the target application. One simulation assumes a perfect cache, and one assuming the minimal (i.e., reference) cache configuration \( \zeta_{min} \). These runs provide us with the number of execution cycles for these two cache setups \((cyc_{pc}, cyc_{\zeta_{min}})\) and the address trace. The address trace is then input to our stack histogram computation, where we derive a stack histogram for each set \( C_s \) (see Eq. (3)).

The last step within the setup phase consists of calculating the miss statistics for the reference cache setup \( \zeta_{min} \), i.e., the miss-rate \( \text{miss-rate}_{warm} \), the cycles per warm miss \( \text{cycles}_{pc \cdot \text{warm}} \), and the memory overlap \( \text{memory overlap}_{warm} \) as described in Eq. (13). By combining the memory overlap and the number of cycles of the perfect cache, we calculate \( cyc_{\zeta} \) for cache configuration \( \zeta \) using Eq. (14) as follows: \( cyc_{\zeta} \approx \frac{450 + 5 \cdot 100}{1 + \frac{6}{100}} = 900 \) which provides an exact estimate (which was to be expected as we have used configuration 1 as reference configuration).

Using the same stack distance histogram, we also calculate that setup 2 results in 6 cold misses, 5 warm misses and 1 hit. Using Eq. (10), we get the following miss-rate \( \text{miss-rate}_{warm} = \frac{6}{12} \) and using Eq. (14), we estimate the number of cycles as follows: \( cyc_{\zeta} \approx \frac{450 + 5 \cdot 100}{1 + \frac{6}{100}} \approx 834 \), which is only off by 16 cycles (error of less than 1.9%). Using the naive approach from Eq. (6), we get the following estimate \( cyc_{\zeta} \approx \frac{450 + 5 \cdot 100}{1 + \frac{6}{100}} = 900 \) which is 100 cycles off (error rate of 11.7%).

C. Framework

Our framework to estimate the cache performance consists of the following steps:

1. Within the setup phase, we perform two simulations of the target application.
2. Within the DSE phase, we perform a performance estimation framework.
3. We then revisit the example depicted in Figure 1 to illustrate the presented concepts. The figure shows that a run of the simulator with a perfect cache takes 450 cycles and the run of the simulator with a direct mapped cache takes 900 cycles (i.e., \( cyc_{\zeta_{min}} \)). We see using Eq. (5) and (6) that the execution with configuration 1, direct mapped, results in 6 cold misses and 6 warm misses. Thus, the warm miss-rate as given by Eq. (10) is \( 6/900 \approx 0.0067 \) and the memory overlap (Eq. (13)) is \( 6/900 \approx 0.0067 \). By combining the memory overlap and the number of cycles of the perfect cache, we can compute \( cyc_{\zeta} \) for cache configuration 1 using Eq. (14) as follows: \( cyc_{\zeta} \approx \frac{450 + 5 \cdot 100}{1 + \frac{6}{100}} = 900 \) which provides an exact estimate (which was to be expected as we have used configuration 1 as reference configuration).

Using the same stack distance histogram, we also calculate that setup 2 results in 6 cold misses, 5 warm misses and 1 hit. Using Eq. (10), we get the following miss-rate \( \text{miss-rate}_{warm} = \frac{6}{12} \) and using Eq. (14), we estimate the number of cycles as follows: \( cyc_{\zeta} \approx \frac{450 + 5 \cdot 100}{1 + \frac{6}{100}} \approx 834 \), which is only off by 16 cycles (error of less than 1.9%). Using the naive approach from Eq. (6), we get the following estimate \( cyc_{\zeta} \approx \frac{450 + 5 \cdot 100}{1 + \frac{6}{100}} = 900 \) which is 100 cycles off (error rate of 11.7%).

V. Evaluation

In this section, we evaluate the effectiveness of our performance estimation framework. We focus in particular on (i) the accuracy of our predictions and the improvement with respect to the naive estimation from Eq. (6) (both in terms of predicted misses and predicted cycles), (ii) the fidelity in the predicted order of candidate configurations, and (iii) the speed-up compared to the use of cycle-accurate simulation.

The design space, i.e., the set of candidate cache configurations, is as follows. We assume a line size of 32 bytes. The size of the cache ranges from 1 to 32 kB with a range of the
As benchmarks, we use the Mälardalen benchmark suite [18] (except for bsort100 due to a segmentation fault and sqrt which lacks a main-function), which gives 33 benchmarks and thus 66 scenarios in total when separately considering instruction and data caches. Due to space limitations, the graphs in this section only show a representative selection of the benchmarks, which contains the corner cases and spikes of the results. This means that results for the missing benchmarks lie in between those presented in the graphs. The gem5 [12], [19] instruction set simulator (ISS) serves as the cycle-accurate simulation within the framework (i.e. to obtain the cycle counts for the perfect and minimal cache configuration) and provides the cycle counts for all benchmarks and all cache configurations to evaluate the precision of our estimates. The simulator models a 32-bit ARMv7-A with a 5-stage pipeline which supports out-of-order execution. All simulations have been performed on an Intel QuadCore i7-2600.

A. Misses
Out-of-order execution may influence the actual address trace, such that the order of accesses depends on the actual cache set-up. We therefore first evaluate the number of misses predicted by our method (which is based on the address trace of the perfect cache) compared to the actual number of misses derived by the simulation (assuming the specific cache setup). We calculate the error using \( \text{error} = \left| 1 - \frac{\text{miss}_{pc,c}}{\text{miss}_c} \right| \) where \( \text{miss}_{pc,c} \) is the number of misses predicted by the trace from the perfect cache. The average error is shown in Figure 3. The figure shows that the average error in the number of misses of our estimation framework is below 5%. It also shows that the I-cache is more strongly influenced by the out-of-order behaviour than the D-cache. We note that in an in-order processor, the order of the memory operations would not have changed and we would have an error of 0%.

B. Validity
In Figure 4, we have visualized the average error in our predicted number of cycles compared to the actual number of cycles obtained using simulation. For readability we have omitted the error bars from Figure 4 which would have shown an average standard deviation of 2%.

We calculate the error using \( \text{error} = 1 - \frac{\text{cyc}_{pc,c}}{\text{cyc}_c} \) where \( \text{cyc}_{pc,c} \) is the number of cycles we predict. We have determined the memory overlap using the smallest cache configuration \( (\zeta_{\text{min}}) \) from our set of candidate configurations. This is a 1 kB cache with associativity 1, i.e., direct mapped.

The figure shows that for most benchmarks our predictions are close to the obtained values using simulation: From the 66 scenarios (of which only 34 are shown in the graphs due to the limited space), only 4 exhibit an average error of more than 5% and the majority (61) has an average error of below 3%. This indicates that our framework yields good results in predicting the number of required cycles.

The line graphs in Figure 4 show the improvement of our estimations compared to the naive approach from Eq. (6) on the secondary y-axis. From these results we can see that our approach is on average a factor 3.95 better for the prediction of the D-cache and a factor 6.95 for the I-cache.

The bars in Figure 4 show a large error of our predictions for two benchmarks (matmult and nsichneu). We have observed for these two benchmarks significant performance differences for the minimal cache setup \( (\zeta_{\text{min}}) \) compared to other cache setups (not shown). This results in a stark under- or over-approximation of the memory overlap, which in turn results in a larger error of our predictions. However, our framework still improves over the naive approach.

Note that we are agnostic to the target architecture and did not perform an in-depth analysis of the hardware. We only require the availability of a cycle-accurate ISS to predict the number of required cycles.

C. Fidelity
As we have seen, our estimates are very close to the actual number of misses and execution cycles, but not exact. However, for early DSE it is often equally important to preserve the fidelity than to have precise predictions [20]. To show the fidelity of the results we use two common metrics: Spearman’s \( \rho \) [21] and Kendall’s \( \tau \) [22]. In short, Spearman’s \( \rho \) compares the rank of elements between two datasets. Where a value of 1 means the ranks match and a value of −1 means the ranks are reversed. Kendall’s \( \tau \) compares the relation of one item in the set to another. Again a value of 1 means the ordering matches and a value of −1 means a reverse ordering.

For the I-cache, the average value of Spearman’s \( \tau \) for the whole benchmark suite is 0.994 with a standard deviation of 0.004. The average of Kendall’s \( \rho \) is 0.983 with a standard deviation of 0.017. For the D-cache, the average of Spearman’s \( \rho \) is 0.956 with a standard deviation of 0.038. The average of Kendall’s \( \tau \) is 0.924 with a standard deviation of 0.062.

Our method is an approximation and is subject to an error margin, as can be seen from the figures. For cache setups that
perform within 1% of each other, our method can therefore not provide a proper judgement. This, however, is acceptable for early DSE as other design-criteria (such as hardware costs or energy consumption) are likely to outweigh such a minor difference. To take this into account, we also investigated the effect of allowing a 1% error in the number of calculated cycles in the calculation of the fidelity metrics. This means we allow shuffling of the ordering if the results are within a 1% range of each other. For the I-cache predictions this improves Spearman’s $\rho$ to an average of 0.999 with a standard deviation of 0.003 and Kendall’s $\tau$ to an average of 0.990 with a standard deviation of 0.012. The D-cache experiments show similar behaviour with Spearman’s $\rho$ improved to an average of 0.991 with a standard deviation of 0.015. Kendall’s $\tau$ is improved to an average of 0.954 with a standard deviation of 0.042. These results indicate that there are several cache configurations with very similar performance.

D. Speed-up

Finally, we evaluate the execution time of our approach compared to exhaustive simulation using gem5. We therefore compare the total execution time of the simulation of the 36 cache setups with the execution time of our framework, including both the setup-phase (executed once in total) and DSE-phase (executed once for each cache setup). In Figure 5, the speed-up of our approach is shown per benchmark. We have measured speed-ups between 6$x$ and 23$x$ with an average of 15$x$. This shows that for all benchmarks, we are significantly faster than a complete simulation of all cache configurations.

Figure 6 visualizes the point at which the overhead of the setup-phase amortizes and our approach is faster than simulation. The dotted blue line shows the normalized execution time when exploring cache configurations using simulation. The line with the upward pointing arrows represents the normalized time our framework requires for the benchmark $fir$ when predicting D-cache, i.e., the minimal speed-up. The line with the downward pointing arrows represents the benchmark $lms$ when predicting I-cache, i.e., the maximum speedup. The break-even points show the number of cache configurations where the prediction using our framework and using the simulations require the same execution time. The break-even points are at 6 cache configurations in case of $fir$ and D-cache exploration, and at 2 cache configurations in case of $lms$ and I-cache exploration. The average break-even point lies at 2.37, which is to be expected as our framework requires two runs of the simulator and limited additional overhead to compute the metrics.

VI. CONCLUSION

In this paper, we have presented a framework to estimate the performance of systems with out-of-order processors using caches. Our framework consists of two phases: a setup and a DSE phase. The setup-phase performs two cycle-accurate simulations and a trace analysis to extract relevant information metrics, after which the DSE-phase then derives accurate performance estimations of the target application for architectures with different cache setups within negligible execution time. Evaluation has shown that the framework is precise both in terms of estimated number of cache misses (average error of less than 4%) and estimated execution cycles (average error of less than 3.5%). Our framework improves upon a naive estimation based purely on the number of misses by a factor of 3.95 in case of I-caches and by a factor of 6.95 in case of D-caches. The fidelity in the ordering of candidate cache setups is near optimal with an error of around 5%. We furthermore have observed an average speed-up of 15$x$ compared to cycle-accurate simulation, which represents the only precise alternative. Our framework thus enables quick yet accurate performance estimation for out-of-order processors with caches to be used for system-level DSE.

REFERENCES