When Misses Differ: Investigating Impact of Cache Misses on Observed Performance

Vlastimil Babka, Lukáš Marek and Petr Tůma

Department of Software Engineering
Faculty of Mathematics and Physics, Charles University
Malostranské náměstí 25, Prague 1, 118 00, Czech Republic
E-mail: {vlastimil.babka,lukas.marek,petr.tuma}@dsg.mff.cuni.cz

Abstract—Although modeling of memory caches for the purpose of cache design and process scheduling has advanced considerably, the effects of cache sharing are still not captured by common approaches to modeling of software performance. One of the obstacles is lack of information about the relationship between cache misses, which the cache models usually describe, and the timing penalties, which the performance models require. Following earlier work that has shown how cache misses do not quite account for timing penalties, we report on extensive experiments that investigate the connection between cache sharing and observed performance in more depth on a real computer architecture.

Keywords—processor caches; performance modeling; resource sharing;

I. INTRODUCTION

Due to the significant difference between processor core speeds and memory module speeds, memory caches play an important role in the observed software performance. By employing workload locality properties, caches partially compensate for the difference in core and memory speeds, and are therefore an essential part of contemporary computer architectures.

When multiple workloads share a cache, observed software performance is necessarily influenced – Jiang et al. [2] show that for each of 15 workloads randomly selected from the SPEC CPU2000 suite, there is a cache sharing workload combination under which it runs more than half as long as alone. The underlying causes of the effects vary, and – besides the obvious contention for cache capacity – also include more subtle mechanisms such as cache trashing during context switches in hardware interrupts, reported e.g. in [3] as up to 8% increase in execution time, or replacement algorithm disruption during context switches, reported e.g. in [4] as up to 10% increase in cache misses.

Given their magnitude, the effects due to cache sharing need to be included in software performance modeling. This, however, is not the case so far – contemporary approaches to software performance modeling simply assume constant duration of atomic operations, even when those durations are likely to be influenced by cache sharing. One reason for the absence of cache sharing in software performance modeling is that the available cache models are typically concerned with cache misses, relevant to processor design or cache management, but not the associated timing penalties, relevant to software performance modeling.

Pursuing our larger goal of including cache sharing effects in software performance modeling, we extend our earlier work that has shown the relationship between cache misses and timing penalties to be weaker than expected [5]. We do this by conducting experiments that identify various individual factors contributing to performance effects under cache sharing.

Since our motivation is associated with software performance modeling, which needs to consider the existing hardware, rather than cache design, which usually proposes a new one, we take care to execute measurements on a real platform rather than on a simulator. While this helps to make our results practically relevant, we should point out that it also makes attribution of the observed effects to the involved mechanisms particularly difficult.

The paper is structured as follows. We outline the research context by surveying the recent software performance modeling contributions in Section II-A and the selected cache models in Section II-B. In both areas, we focus only on cache sharing, analyzing the assumptions made with respect to sharing effects and timing penalties. In Section II-C, we briefly recapitulate our earlier work showing the weaker than expected relationship between cache misses and timing penalties. Section III introduces artificial workloads that isolate factors contributing to the timing penalties, and shows how real workloads are influenced by these factors. Finally, we conclude with a short discussion in Section IV.

II. RESEARCH CONTEXT

A. Software Performance Models

Software performance modeling is tied to earlier research in software architectures, which postulates that software consists of components interacting through interfaces. The interaction typically assumes the form of function invocations. Given the durations of all invocations, and an information on what incoming invocations cause what outgoing invocations, a performance model can be created.

By looking at recent case studies in software performance modeling, we can see that the effects due to cache sharing are not explicitly covered – the only property captured by the models that is sensitive to cache sharing are the invocation
durations, and these are assumed independent even when they would likely be influenced by cache sharing.

Out of 15 recent case studies from venues including WOSP, SIPEW, SIGMETRICS, five studies [6–10] simply assume the invocation durations are provided by the developer. Ten studies [11–20] go further by suggesting, either directly in the text or indirectly in the references, that the invocation durations are collected by measurement. In all cases, the exact manner in which the invocation durations are collected is considered mostly out of scope of the study, with estimates and measurements being apparently acceptable approaches.

In cases where estimates are used, it is unlikely that the effects of cache sharing would be captured faithfully in the invocation durations. In cases where measurements are used, the degree to which the effects of cache sharing are captured in the invocation durations depends on the workload used during measurement. Some studies [13, 18, 20] suggest using a simplified workload, while others [17] mention using a workload that resembles the software being modelled. Only rarely [14] are the invocation durations considered anything but constant.

To summarize, cache sharing effects are generally not captured by current software performance modeling practice, which tends to assume constant invocation durations independent of the cache sharing context. Directions that can provide progress include devising techniques to measure the invocation durations under cache sharing conditions resembling the software being modelled, and parametrizing the invocation durations in the models with cache sharing conditions.

B. Cache Sharing Models

The cache models that cover cache sharing greatly differ in complexity, mostly depending on the assumed application – models targeted at online cache management in scheduling tend to be less complex than models targeted at comparative evaluation in cache design.

Among the simplest cache models are models that apply analytical approximations. Models in [21–23] use various forms of curve fitting over observed miss rates and cache sizes to derive a model yielding miss rates under cache sharing. When needed, performance effects due to cache sharing are obtained assuming constant timing penalty per cache miss.

More complex models require knowledge of certain workload characteristics, such as stack distance profiles or reuse distance profiles. Models in [24–31] rely on knowledge of workload characteristics in addition to replacement policy and cache architecture to derive miss rates under cache sharing. Still, performance effects due to cache sharing are typically obtained assuming constant timing penalty per cache miss, except for [26], which points out that determining the penalty associated with a miss on a particular architecture is a complex unsolved problem before resorting to constant timing penalties.

The very few models that describe performance effects in terms of timing penalties rather than cache misses, and that do not convert misses to penalties by simple multiplication, include [32], which models the overlap of communication and computation by a linear factor. Specialized models that estimate cache access times do exist [33], but these models estimate the access times inherent to hardware design rather than the timing penalties perceived by software workload.

The issue of timing penalties on real processors has been marked for both lack of models [26] and lack of available information [34].

To summarize, cache sharing is generally understood in terms of competition for cache capacity and therefore naturally expressed in cache misses. When cache sharing models are to describe performance effects, straightforward conversion from miss rates to timing penalties is done. Notably, performance effects that are not due to misses are not modelled.

C. Cache Miss Penalties

Our earlier work [5] has shown the relationship between cache misses and timing penalties to be weaker than expected. Following other similar experiments, we have taken a subset of the SPEC CPU2006 benchmarks [35] and run them in pairs on a shared cache multiprocessor, examining the correspondence between the increases in cache misses and the increases in execution times.

The cores used in the experiments have only shared the memory hierarchy starting with the second level cache, the observed effects were therefore necessarily due to either cache sharing or memory bus, memory controller, and memory module sharing that occurs after missing in the cache.

Since the selected workloads were computation oriented, the interaction through the operating system was also small, confirmed by the fact that the difference between the wall clock time and the processor ticks spent executing the user space portion of the workloads was generally less than 1%.

The results of the experiments have shown that converting from cache misses to performance effects by considering fixed penalties is very imprecise. Although the conversion is somewhat complicated by prefetching, attempts to fit the observed results through linear regression only yield the $R^2$ coefficient of 0.6 or worse. Interestingly, fitting the observed results through linear regression appears to work better when cache misses in a single workload, rather than cache misses across all workloads, are considered. This suggests that the sensitivity of execution times to cache misses is stable within a benchmark but varies between benchmarks.

III. UNDERSTANDING CACHE SHARING EFFECTS

In [5], we have concluded that the sensitivity of execution times to cache misses differs from benchmark to benchmark.
To combine the cache sharing models, which deal with cache misses, and the software performance models, which deal with invocation times, we need to convert misses to times, and therefore to understand the observed effects.

Fig. 1 illustrates the relationship between cache misses and timing penalties. A memory access can be initiated by a hardware prefetch or by a software demand. Once initiated, the access proceeds by searching the individual cache levels, ending up in memory when all the cache levels miss. A timing penalty is observed only when the instruction that accesses the data proceeds far enough in the pipeline to have to wait before the data becomes available. The distance between the moment when the access is initiated and the moment when the data is used, which is a property of both the workload and the platform, therefore determines the sensitivity of the execution time to the cache misses.

In the scenario illustrated on Fig. 1, cache sharing can influence the timing penalty in multiple ways – by competing for hardware prefetch, by competing for data storage capacity or request handling capacity of shared cache levels, and by competing for request handling capacity of shared memory bus.

Given the complexity of our experiment platform, there are many hypotheses that could explain the observed effects, but it turns out to be rather difficult to test them experimentally. Very often, the insight provided into the effect by the hardware performance monitoring support is limited, simply because there are no events that would be linked directly to the observed effects. Hypotheses are also often based on undocumented assumptions about the experiment platform, which might or might not be true. In fact, even some documented assumptions about the experiment platform might turn out not to be true, usually due to ambiguity in documentation.

To move forward, we have designed special artificial workloads for each hypothesis that attempts to explain the observed effects. The artificial workloads attempt to isolate and emphasize the mechanisms used in the hypothesis and therefore help understand what is happening. We then examine the behavior of the benchmark workloads in combination with the artificial workloads, or the behavior of a combination of multiple artificial workloads.

Note that due to a high number of hypotheses explaining the observed effects, the number of artificial workloads and conducted experiments is rather high. Here, only a selection of workloads and experiments leading to our conclusions is provided, and the experiments are described briefly. More details are given in [36] and a thorough description, including code listing, is provided in [37], a report of over 200 pages.

An experiment consists of multiple workloads running together over a shared cache. A workload whose timing is measured in the experiment is called measured workload, workloads included for the purpose of generating well defined interference are called interfering workloads.

In order to hit or miss in a cache, the artificial workload accesses addresses in a memory area of given size using predefined access patterns. To minimize the overhead of following a complex pattern, the patterns are stored as a chain of pointers that forms the very data to be accessed. This is further called pointer walk.

Note that the pointer walk is synchronous in nature. A new memory access can only be initiated when the previous one has completed. The pointer walk therefore emphasizes access latencies while limiting transfer rates. When request handling capacity of the cache or the memory needs to be exercised, the memory area used by the workload is split into evenly sized fragments and each fragment is initialized with a separate pointer chain. This multipointer walk workload then keeps as many pointers as there are fragments, advancing each pointer independently in a round-robin fashion, making it possible to initiate multiple independent memory accesses.

The workload construction is further complicated by unexpected platform dependent performance artefacts, such as the dependency of the access time on the offset within a cache line, or the dependency of the access time on the time since the accessed data was evicted from the private cache. Some of these effects are described in [36] and more thoroughly in [37], for the purpose of this paper suffice to say that some of the workload aspects were finetuned or randomized to obtain meaningful results.

To illustrate the results, we typically provide plots of values such as the duration of the measured operation or the value of a performance counter, typically plotted as a dependency on one of the experiment parameters. Absolute durations are expressed in processor clock cycles.

---

1Dell PowerEdge 1955, dual Quad-Core Intel Xeon E5345 2.33 GHz (Family 6 Model 15 Stepping 11), internal private 32 KB L1 caches, two 4MB L2 caches, each shared by two cores, 8 GB Hynix FBD DDR2-667 synchronous memory, Intel 5000P memory controller. Fedora Linux 8, gcc-4.1.2-33.x86_64, glibc-2.7-2.x86_64.
When averages are used in a plot, the legend of the plot informs about the details. The Avg acronym is used to denote standard mean of the individual observations – for example, 1000 Avg indicates that the plotted values are standard means from 1000 operations performed by the experiment. The Trim acronym is used to denote trimmed mean of the individual observations where 1% of minimum and maximum observations was discarded – for example, 1000 Trim indicates that the plotted values are trimmed means from 1000 operations performed by the experiment. The acronyms can be combined – for example, 1000 Avg Trim means that observations from 1000 operations performed by the experiment were the input of a standard mean calculation, whose outputs were the input of a trimmed mean calculation, whose output is plotted.

A. Request Handling Capacity

We first consider the competition for request handling capacity of shared cache levels. Although the particular limits are usually not mentioned in vendor documentation, multiple mechanisms including cache banking or miss state handling registers can limit concurrent access to cache. Taking our experiment platform as an example, we can find a hint suggesting that it is only possible to handle one miss in a particular set at a time in the description of the L2_REJECT_BUSQ performance monitoring event [38] page A-15. When limits on concurrent access to the cache exist, cache sharing can introduce timing penalties due to reaching these limits.

To assess these limits and penalties, we perform an experiment that runs multipointer walks with random access pattern on two cores sharing a cache. The measured workload is configured to represent an application that relies on cache heavily. The size of the accessed memory area is set so that the measured workload either misses in the private L1 cache and hits in the shared L2 cache, or misses in the shared L2 cache, representing an application that mostly hits or misses, respectively.

The interfering workload is configured to represent two variants of an application that issues multiple independent memory accesses. In one configuration, a random access pattern is used together with memory area size that fits in the shared L2 cache together with the measured workload, so that both workloads mostly hit on access. In the other configuration, conflict misses are triggered in the shared L2 cache, yielding a workload that mostly misses on access without actually competing for much of the shared L2 cache capacity. To achieve this effect, the access pattern of each chain is created from addresses that map to the same associativity set, different chains use different sets. Because a set in the shared cache is selected by physical rather than virtual address, the experiment requires allocation using a page coloring strategy [36].

![Figure 2. FFT slowdown due to interfering workload that triggers shared L2 cache hits.](image)

Fig. 3(a) shows the slowdown of the measured workload when both workloads are configured to mostly hit on access. Depending on the number of pointers used by the workloads, and therefore the number of multiple independent memory accesses issued, up to 16% slowdown can be observed. Since the workloads are not competing for capacity and not prefetching, it is likely that this overhead can be attributed to the limited request handling capacity of the shared L2 cache.

Fig. 3(b) shows the slowdown of the measured workload when it hits while the interfering workload misses on access. The slowdown can be very significant, up to 106%.

Finally, Fig. 3(c) shows the slowdown of the measured workload that misses in the shared cache, with the interfering workload that hits. The effects appear to be the smallest of the three combinations considered.

With real workloads, using the same interfering workloads to highlight the effects of concurrent access to the cache is only possible when the real workloads trigger no misses and no prefetches. For workloads that do trigger misses, the interfering workloads could also compete for memory bus, and for workloads that do trigger prefetches, the interfering workloads could also compete for hardware prefetch. Both cases are discussed later, here we focus on 453 povray, which exhibits a miss rate of only 100 misses per second when run in isolation and therefore meets best the isolation condition. The slowdown of this benchmark is 3% when run in parallel with the hitting variant of the interfering workload, 21% with the missing variant, both using 64 pointers.

As another example of a real workload that triggers no misses and no prefetches, we use the FFT transformation as implemented by FFTW [39]. In the experiment, a transformation of a buffer that does not exceed 1 MB fits in the shared L2 cache, as evidenced by the L2_LINES_IN event counter, and exhibits up to 10% slowdown with the hitting variant of the interfering workload, 70% with the missing variant. The hitting variant is illustrated on Fig. 2.

Overall, the experiments show that a significant overhead can occur due to cache sharing even when there is no
competition for cache capacity or memory bus. This supports the hypothesis that there indeed is a limit on concurrent access to the cache, with accesses that miss occupying the cache longer than accesses that hit.

B. Hardware Prefetch Competition

The second source of cache sharing effects that we consider is competition for hardware prefetch. The experiment platform has two documented prefetch mechanisms to the shared L2 cache, namely the streamer prefetcher and the DPL prefetcher [40]. Prefetch misses should be handled with lower priority than demand misses and should track cores independently [40, page 2-38].

A concurrent workload could influence prefetches in two ways. First, the number of access patterns tracked by the DPL prefetcher is limited and memory accesses therefore compete for access pattern tracking. Second, prefetches can be discarded if there are too many outstanding misses. In both cases, the influence is related to presence of multiple outstanding requests, and therefore cannot be separated from the influence of competing for request handling capacity. We can, however, compare the difference in cache sharing overhead between workloads that are sensitive or insensitive to discarded prefetches.

As a workload that benefits from prefetches, we use the multipointer walk with a linear access pattern. The total memory area is set to twice the shared cache size so that the workload triggers misses, which should in turn trigger prefetches.

The interfering workload is a random multipointer walk hitting in the shared cache, which consumes only a minimal cache capacity, but still competes for access pattern tracking and therefore potentially influences prefetches.

Fig. 3(d) shows the experiment results. A straightforward comparison with the random multipointer walk on Fig. 3(c) would indicate that the slowdown is indeed more significant in this experiment, but such a comparison is misleading. By checking the L2_LINES_IN performance event counter, we can see that prefetches are only discarded with 8 or more pointers used by the measured workload, and there is no increase in latency for this number of pointers evident in the results. The reason for the difference in relative slowdown of random and linear workloads is actually due to the fact that demand accesses in the linear workload are turned from misses to hits thanks to prefetch, and since a workload that hits progresses faster than a workload that misses, the same

---

2It is also possible to disable prefetching entirely using the MSR registers of the experiment platform. This would provide somewhat different results compared to using an interfering workload, which is unlikely to suppress prefetching completely.
absolute timing penalties appear relatively larger.

Given the fact that demand accesses in the linear workload hit, it makes sense to compare the experiment results with the random hitting workload in Fig. [3(a)] where the demand accesses also hit. We can see that the slowdown on the linear workload is more significant, a very notable difference is visible when the measured workload uses just one pointer. In this case, the limit on concurrent access exercised by the interfering workloads has almost no impact – from 23 to 24 cycles per step of the random pointer walk, but the slowdown of the linear pointer walk is 26% – from 47 to 59 cycles per step. This is very different from the previous experiments.

To determine the influence of competing for hardware prefetches on real workloads, we use the same interfering workloads as in the previous experiment – the observed effects are therefore a combination of competing for request handling capacity and competing for hardware prefetch.

The influence of the random multipointer walk with 64 pointers hitting the shared cache on the SPEC CPU2006 benchmarks is presented in Fig. 4 which shows the observed slowdown and the observed decrease of prefetch misses. The benchmarks are sorted by the prefetch miss rate observed when run in isolation, in descending order. We can see that generally, a larger slowdown goes hand in hand with a larger prefetch miss rate observed in isolation and a larger decrease in prefetch misses, but notable exceptions exist (e.g. 462.libquantum and 433.milc) and there is no obvious correspondence between the size of the slowdown and the size of the decrease in prefetch misses.

Note that for the three benchmarks with the lowest prefetch miss rate, we actually observe an increase of prefetch misses, but due to the low rate the difference in absolute numbers is also low. Counts of prefetch misses for the benchmarks with lower prefetch miss rate are also relatively less stable, all benchmarks with rate higher than 401.bzip2 have less than 1% CoV.

The difference in time for 464.h264ref, the largest of all benchmarks, is therefore not related to decreased prefetching, and since the number of demand misses and their absolute increase is also relatively low, most of this overhead should be attributed to the sharing of request handling capacity. The increase of prefetch misses for 465.tonto however cannot be explained by either low prefetch miss rate or unstable results.

We also add an experiment with FFT, with results on Fig. 6. For buffer sizes of 2 MB or more, the transformation does not fit in the shared L2 cache, and the performance counters indicate that most misses are prefetch misses when run in isolation, and more than half of them changes to demand misses as the number of pointers in the interfering workload increases. We can see that the impact of interference is much higher for buffer sizes between 2 MB and 8 MB, up to 27% with 4 MB buffer.

From the results of the experiments, we can conclude that a workload relying on prefetching is more likely to be influenced by sharing the cache with another workload, even when the workloads do not compete for shared cache capacity. It is, however, difficult to estimate the influence from the demand miss and prefetch miss counts provided by the performance event counters.

C. Memory Bus Contention

The last considered source of influence due to cache sharing is memory bus contention. Note that it is possible to compete for the memory bus even with very little competition for shared cache capacity – this happens when the competing workloads access different cache sets, or have working sets so large that they do not actually benefit from caching. Since competition for the memory bus is necessarily associated with frequent cache misses, its influence cannot be isolated from the effects of request handling capacity and hardware prefetch contention. On our experiment platform, we can, however, compare the slowdown that occurs when just the memory bus is shared, with slowdown that occurs when the L2 cache is shared.

To minimize competition for cache capacity, we again use the multipointer walk that triggers misses only in a limited number of associativity sets, as the interfering workload. We bind the interfering workload either to a processor core that shares both the L2 cache and the memory bus with the measured workload, or to a processor core that only shares the memory bus.

The slowdown of the SPEC CPU2006 benchmarks is shown on Fig. 5. The benchmarks are sorted in descending order of their total miss rates when run in isolation. It can roughly be observed that a higher miss rate results in a larger slowdown. Note the excessive slowdown of 189% for 462.libquantum, which can be attributed to flushing of dirty cache lines.

As would be expected, sharing the L2 cache in addition to the memory bus usually results in a larger slowdown for rea-
An explanation of this effect, supported by the value of the SPEC CPU2006 benchmarks, is shown in Figure 5. The slowdown of SPEC CPU2006 benchmarks, due to interfering workload triggering either misses in the shared L2 cache or memory reads on the shared memory bus.

IV. Concluding Remarks

Investigating the relationship between cache sharing and software performance, we have pointed out [5] that one of the missing pieces in the existing models is the relationship between cache misses and timing penalties. The frequent assumption of constant penalties does not hold for the experiment platform – and more, the assumption is difficult to relate to observed cache misses since these may include both useful and useless prefetch misses.

We have examined some potential causes of the weak relationship between cache misses and timing penalties, including competition for hardware prefetch, competition for request handling capacity of the shared cache, and competition for request handling capacity of the memory bus. Since it is difficult to attribute timing penalties to these causes through direct measurement, we have devised artificial workloads that emphasize one potential cause while suppressing others – e.g., a workload that triggers cache misses without actually evicting much of the cache capacity – and used these workloads to evaluate the timing penalties on the SPEC CPU2006 benchmarks.

To summarize the observations, we have shown that workloads differ significantly in their sensitivity to cache misses, and that besides cache misses, workloads can also be sensitive to other aspects of cache sharing, especially when the ability of the cache to handle multiple requests, or the ability of the cache to initiate prefetch requests, is stressed. Notably, some of these effects are evident even for workloads that do not share cache directly, such as the 21% slowdown of 453.povray due to limits on concurrent cache access coupled with coherency protocol traffic.

If cache sharing is to be included in models of software performance, the identified cache sharing effects need to be covered to achieve reasonable modeling precision. Although we do not provide such a model (yet), we believe our contribution is in better understanding of the effects, in the difficult experimental context of a real platform. This understanding has multiple immediately useful implications, such as guiding the configuration of cycle precise simulators to include effects that occur in practice. Also, the artificial workloads described in this paper serve to identify the sensitivity of software being modelled to the individual cache sharing effects, and provide the background workload when measuring invocation durations, as outlined in Section III.

Including cache sharing in models of software performance is still an open task that faces more unresolved issues, of which we list just two. One, many cache models are rather complex and including them into performance models would aggravate the already difficult state explosion issues and make the models computationally intractable. Two, any cache models require detailed workload information which might not be available at early software development stages, which is where the performance models are usually applied.

Due to space reasons, we have limited the presented experiments. For each of the potential causes of the weak relationship between cache misses and timing penalties, we only show enough experiments to illustrate the size of the potential impact, and to verify that the SPEC CPU2006 benchmarks are indeed susceptible. For further experiments, with more detailed description and more experiment platforms, please check out [37].

ACKNOWLEDGMENT

The authors would like to thank their fellow DSRG members for fruitful discussions and proofreading of the paper. This paper was partially supported by the Q-ImPrESS research project by the European Union under the ICT priority of the 7th Research Framework Programme, and by the Czech Science Foundation grant 201/09/H057.

REFERENCES

[1]

