Optimizing Cache Performance and Predictability

Using different Strategies to improve the interior Structure and to manage the Cache's Behavior in Real-Time Systems

Vanessa Kern
vanessa.kern@fau.de
Friedrich-Alexander Universität Erlangen
Erlangen, Deutschland

ABSTRACT

In today's multi-core systems each processor has its own on-chip private cache. In addition, they all access a bigger shared cache which can yet lead to problems. As the space in the shared cache is restricted, data is likely to be evicted often when multiple processors access the cache simultaneously. Therefore, the question arises how to optimize caches to get an improvement in performance and to take advantage of them in terms of speed. Besides, in real-time systems, it is important that a computation is available at a certain time and therefore, it is essential that the cache's interior data is predictable at any time.

First of all, it should be considered, achieving an improvement in data and data structures themselves. Manipulating the size of data and the order of code execution has an impact on localities and thus can improve the system's performance.

Major points are different cache management strategies with a focus on cache partitioning and cache locking. Their effect and the behavior of the cache will be presented which show a possibility to control the content inside the cache and to make it more predictable. In order to avoid eviction of data from the cache and to minimize the flow of data from the main memory, the right scheduling is very important. Thus, much attention is payed to the right timing and the criteria under which events are scheduled.

1 INTRODUCTION

Caches are mostly a gain of time, because cached data is available without accessing the main memory. However, in today's multi-core systems, shared resources need a special treatment as multiple processors can access them at the same time. Especially efficient cache usage is a very important topic for research because data that is currently used by one core can be evicted by another processor that references a different address in main memory which is mapped to the same cache line. This leads not only to a longer execution time but also requires more energy as data needs to be fetched from main memory more often.

However, eviction does not only occur if there are multiple cores. Considering data for the execution of a single task of one processor, these data structures are often stored side by side in main memory and thus, are mapped to the same cache line. Yet, if they are too large in size to fit inside together, it leads to cache misses and they will evict each other after each execution. For instance, if two arrays are called inside a loop and are mapped to the same cache line due to their offset, eventually, they evict each other because they do not fit inside a cache line together. This is called *cross interference* which results in writing back these data structures to main memory and retrieving them again each time. It consumes more energy and leads to a worse system performance.

This underlines the importance of a suitable alignment since data that is stored side by side with the current element in main memory, will probably be accessed in near future as well. This is called *spatial locality* which means that for instance, all elements of an array should be cached because they are likely to be used if an element of the array is referenced. In addition, a problem occurs if data that is highly reused, like run variables, gets replaced and needs to be restored oftentimes. This leads to cache misses and thus, the performance deteriorates. Therefore, it should be taken profit of so-called *temporal locality*. It states that addresses that have been referenced once are probably accessed in near future again. So, by ensuring that this data stays cached for a certain time, this can also prevent unnecessary eviction. Therefore, localities should be considered by designing a technique that improves the structure of data itself inside the main memory. It should be taken advantage of these localities and thus, achieve a better performance since data is evicted from the cache less often.

Space in caches is limited as bigger ones would be quite costly. Thus, this space has to be managed carefully. In case the content exceeds the size of the cache, *thrashing* would occur. Thrashing means, that there are multiple main memory accesses that compete over the same cache line and thereby constantly evict each other. Managing the content that is stored inside the cache at any time should, therefore, be a target to aim at.

All problems and difficulties named thus far have in common that multiple cache misses occur and that data gets easily evicted though it still may be needed. Without proper countermeasures, the system's performance is severely affected and decreases. Besides, in real-time systems it is essential that tasks are executed until a certain deadline. Multiple cache misses extend the duration of execution and it is difficult to define an execution time for each task and therefore, the worst case has to be assumed. Additionally, it is not predictable when tasks are executed and where data resides inside the system. Therefore, the cache requires a management that primarily, reduces cache misses to achieve a better performance and secondly, improves the predictability by managing the limited space inside of caches. It needs to be well known where data is at which time to guarantee that all data needed is available on a certain moment.

Therefore, in Section 2, different techniques will be described that attempt to take advantage of spatial and temporal locality, in order to reduce the number of cache misses and to achieve an augmentation of efficiency. Besides, two strategies to improve the predictability

of the cache by controlling the content are presented in Section 3. Using these, cache thrashing can be prevented and additionally, the performance can be improved. Finally, in Section 4, a scheduler is presented which combines the introduced strategies and decides on a mechanism to schedule multiple tasks.

2 OPTIMIZATION OF CODE AND DATA

First of all, data itself can be restructured in memory to improve cache efficiency. There are two techniques, *Data Layout Optimizations* and *Data Access Optimizations*, which aim to modify data to fit inside the cache and adapting the execution order to the accessible memory that is cached.
Data layout optimizations are not essential, however, they provide a way to improve the arrangement of data structures inside of main memory. Using *Array Padding*, the problem cross interference can be solved by simplifying the allocation to the cache lines. It provides padding to each array which modifies the offset of their addresses in order to map them to different cache lines. Therefore, spatial locality of applications can be improved because data residing close to each other in main memory, is not mapped to the same cache line anymore [6].
Besides, data access optimizations are in charge of transforming code in order to improve temporal locality. An example of a transformation is called *Loop Interchange* which tries to change the order of multiple nested loops. The reason is that arrays are stored in main memory in either *row-major-order* or in *column-major-order*. Consequently, in case a block is retrieved and stored inside the cache, the order of the containing adjacent elements will be of great importance. As an array can not be cached entirely due to limited space, only parts of it are inside the cache. So, in case the program accesses the array in the opposite order, cache misses eventually occur and data is evicted although it still is required. Therefore, loop interchange swaps the indices of iteration variables to adapt the order the array is accessed to the order it is stored inside the cache [6]. As a result, temporal locality is improved, as the inner loop accesses the element that lies next to the previous element in memory which is already cached [5].
In general, these strategies are an improvement in terms of localities. Yet, they are selected and implemented by the programmer and, using heuristics, it has to be decided carefully in which case those optimizations should be applied. Either, they provide an effective interior structure or they also cause overhead which has to be weighed against each other.
Nevertheless, these strategies are insufficient since they only modify the input, yet do not change anything about the cache's behavior. Therefore, different cache strategies that affect and structure the limited space inside need to be analyzed.

3 CACHE MANAGEMENT STRATEGIES

The focus hereby lies on the interior structures of caches. It is important to avoid processors to randomly access the cache because it would become inpredictable which data is modified by which processor and which data is still needed. Instead it should be well known which data is stored in which part of the shared cache as well as the time it needs to stay inside. Also, it is necessary to prevent processors from overwriting data that is still referenced by another processor. Both the following strategies, cache partitioning and cache locking, are options that provide a way to manage the limited space and prohibit the access of multiple processors to the same cache line. They are often combined and frequently used in real-time systems, because they make execution time more predictable, as it is well known when data is available and by which time a task is executed.

3.1 Cache Partitioning

In order to optimize the usage of limited space in caches, one major aspect is cache partitioning. As a condition, the used hardware has to leave the decision to the operating system, where data is mapped on in the shared cache [2]. The operating system is therefore responsible for possible eviction of data. It is also assumed that the cache is a set associative cache [10].
The principal idea of cache partitioning is to divide the cache into multiple partitions and to assign a number of parts to each processor. Thereby, the strategy consists of two different interdependent aspects. Firstly, a mechanism, that performs the cache partitions by deciding the size of the parts. Also, it selects the input for each partition which means that it allows only the assigned processor to write data to a certain cache part and controls that the size of the input does not exceed the allocated part. Secondly, it consists of a policy that provides the data which is stored inside the cache lines that represents the input. Thus, this data comes from the main memory and it is demanded that it fits inside the cache partitions so it can easily be swapped in and out if required [7].

Operating System Components. As it comes to partitioning, for the sake of simplicity, we consider it to be divided way-based which means that it is parted into the size of cache lines [8]. In general, the cache we are considering is a shared resource, so these parts can be accessed by all processors. Therefore, it is important that the data from the main memory that can equally be referenced by all cores has to be parted as well, as it is intended that partitions from the main memory can be swapped in and out of the cache easily.
Data inside these partitions usually is needed to execute certain applications or different tasks that reference the same space in main memory [3]. So, if one task should be executed, the partition containing the required data can be easily loaded inside the cache as a whole. Especially, if another task that references the same data in main memory is executed, the partition has not to be stored into the cache again as already all data needed is available. So, in other words, the partitions consist of applications and groups of different tasks that reference the same space in main memory.
These partitions are called *Operating System Component* (OSC) which will be of great significance for the scheduling that will be analyzed later on. Concerning the size of those components, it should not exceed the size of the biggest shared cache as it would cause thrashing even if only one component resides inside the cache. Also it should not be too small because it would be equal to a random memory access otherwise. The optimal size of a component is a multiple of the cache way size [1]. These components can be swapped in and out easily, so that necessary data for a certain task resides inside the cache when it is needed [2].
After the partition, those cache slots need to be distributed to all

cores. Each processor is assigned certain ways of the shared cache and it is guaranteed that no other processor is able to write new data in that space. However, within their assigned space, each core has to allocate their own components because this is not done by the operating system [2].
There are different approaches to allocate the ways to the processors: statically and dynamically. On the one hand, during static cache partitioning the number of blocks is computed and determined in advance which means that each core gets a fixed number of ways. This is advantageous, in case there is a smaller amount of processors because thereby each core can access more cache ways. On the other hand, dynamic cache partitioning can reallocate the cache ways every tens to hundreds of seconds during run time [7]. It analyses which core references which cache way the most and allocates this way to the core. For the decision, there are different algorithms that decide which processor is assigned which way. One possibility is a look-ahead algorithm that computes the references of all applications to each way and allocates the number of ways to the processor that it requires. It is convenient for multiple processors because depending on each core's actions, they can be assigned the suitable amount of cache ways they need. Thus, an improvement in performance can be achieved as well as a fair allocation. Each processor is assigned a space inside the cache that it references the most and consequently, less data is evicted [9]. Still, dynamic cache partitioning causes some overhead, however, it is an ambition to reduce it further in future work [7].
One negative point is that the hardware has to be well-known in advance and needs to leave the decision to the operating system, where data is mapped on in the shared cache. However, on the contrary, contention of shared resources can be reduced, since the operating system has more knowledge about the content of the cache [2].
As a result, the content inside the cache is predictable at any time, since only one processor can write data to a certain part. Therefore, it has an impact on the execution time, because it can be controlled which data resides inside at which time [2]. Accordingly, it can be estimated how long an execution takes which is advantageous in real-time systems since it is possible to guarantee that all tasks are executed in time. In addition, eviction of data can thus be prevented and cache miss rates can be reduced in real-time systems.

3.2 Cache Locking

Another strategy is cache locking. The principle idea is to decide on parts of the cache that should be locked, which means that the data inside can not be touched by any other processor than the one that referenced it first. Therefore, data is not evicted until the processor is done with execution and the part is unlocked again which decreases the cache misses. Mostly, it is a hardware-based management but often is controlled by the operating system [2].
In order to reduce the cache miss rate, there are different mechanisms that can be used to lock data of different granularity to the cache, so it can not be evicted. Either selected data at granularity of memory blocks or cache ways can be locked to the cache. The last one is called *lockdown by way* which describes locking way-sized parts to the cache. Also, in case a function is highly reused, it is possible to lock the referenced data inside the cache [2]. So for instance, when a method is called, it is locked to the cache as long as it is being executed. By analyzing the calling frequency, the duration of the locking can be determined. However, it is more common that only basic blocks or cache ways are locked [8].
Regardless of which granularity is chosen, with an analysis of already processed applications, it can be identified which blocks have the highest miss rate. Those blocks should be locked to the cache as they are referenced more often. On the one hand, by buffering these blocks and their addresses in a miss-table that is sorted by descending order, blocks that should be locked to the cache can be selected. On the other hand, those blocks that have a lower miss rate can be evicted [8].
As a counter needs to constantly be stored while execution, storage overhead arises. Yet, this counter is required because locking decisions can be made accordingly. Another problem occurs if the whole cache is locked in which case, additional cache lines need to be stored in a buffer. Eventually, they have to be written back to main memory or they cause latency overhead [8].
However, cache locking has a positive impact on energy efficiency, since less cache misses occur. Thereby, conflict misses can be reduced because data that is used more frequently has been prevented from eviction and thus needs to be retrieved less often from main memory. Additionally, it increases the predictability of the cache's behavior because due to cache analysis, it is predictable when misses and hits occur and it is well-known which data is available at which time [8].
As a conclusion, cache locking is suitable for real-time systems as it is possible to control which data resides inside the cache and thus execution times can be estimated. Besides, overhead mostly is inevitable in order to compute the cache line that should be locked. However, comparing it to the improvement that can be achieved, it is an acceptable price to pay. In combination with cache partitioning, it can be realized that the cache's partitions that only are accessible for one core, can be locked for a certain time to prevent the same processor from being able to evict this block. Cache misses in theory can be entirely prevented, however, there still is need for a scheduler that decides on the order of execution in real-time systems.

4 OPTIMIZATION BY SCHEDULING

Scheduling is the most important point when it comes to optimization because all strategies named thus far are not that effective when they are not scheduled efficiently. Therefore, the following paragraph deals with a strategy for designing a scheduler. As there are many possible ways to do so, this work presents only one example that creates an efficient solution in real-time systems to improve the performance of caches. That includes a strategy to execute tasks without causing cache eviction and a cache profiler that computes a size for each task that will be the determining factor for the scheduling order.

4.1 Controlling Events

As already mentioned and described in Section 3.1, data can be gathered into components (OSC). In order to manage the data and the control flow, these OSCs need to be restricted which means that the interaction between those components has to be regulated.

That implies that direct function calls between components need to be completely stopped and mutual communication has to be controlled by the operating system. In case the interaction was not restricted, it could not be guaranteed that all referenced data would fit inside the cache without evicting each other, as one components could call another one that has to be stored inside the cache as well. Therefore, *events* are an efficient way to prevent direct communication between components. They are visible and accessible within the whole operating system. So, in order to subscribe to these events each OSC can define so-called *triggers*. After activating those triggers, a function is called which will start the execution of the OSC on that occasion [1]. As a condition, a system that can execute triggered functions is required. Consequently, the operating system maintains control over the communication between the OSCs the whole time.
To guarantee that no data is overwritten or read by another trigger, it is important, that only one trigger per component can be executed at once. Then, even data exchanges can be realized by creating a dependency between a trigger and another OSC, which implies that the content of the dependent component will not be touched as long as the trigger is activated [2].

Execution of components. Concerning the execution of triggers, it is split up in three phases: Firstly, the preloading phase which loads the OSC in unused parts of the shared cache after the trigger is activated. Secondly, the execution phase which executes the OSC without accessing the main memory. This can be realized as the operating system has more knowledge about the memory hierarchy and it is guaranteed that data remains cached. If the data has been overwritten, a *dirty* flag is set and it has to be written back to main memory. Lastly, the writeback phase writes back all data that has been flagged as dirty because they have been updated [2].
Without the right scheduling, the approach until that point would only cause that OSCs evict each other in the preloading phase if the size of the shared cache is exceeded. As a counter, cache locking and cache partitioning that already have been discussed in Section 3 are very effective solutions. On that account, the cache can be divided in multiple parts that can be assigned to different cores and those parts of shared cache can be locked, so that the components residing in this space are not evicted [2].
Concerning the scheduling, it has to be decided on the order of execution of the events. The question arises, what strategy should be followed to achieve a reduction of cache misses. It can theoretically be achieved if those components needed at that moment are present inside the cache and those which have not been referenced in a while are swapped out to give way to those components needed [1]. So, as already stated, when a component's trigger is activated, a task will be created by the system that will execute the component's method [2]. There are different strategies to schedule those tasks, one is presented that additionally supports multi-threaded tasks.

Scheduling multi-threaded tasks. The question arises whether an effective management of a scheduler can avoid thrashing which would have a negative impact on the system's performance. It is assumed that the components support multi-thread processing and that they consist of *multi-threaded tasks* (MTT). Each task has different jobs with their own deadlines which mark the time they should be executed. Using different co-scheduling choices, these deadlines can be prioritized which means that the deadline is set to the current time, in order to execute this task next on. Thereby, the scheduling strategy can be influenced. One possible and effective way to do that is described in [4].
Each MTT has a per-job *working set size* (WSS) which describes the amount of memory needed to execute this MTT. The required data resides inside the cache while the tasks are executed. Consequently, if the sum of all WSSs of the MTTs exceed the total size of the shared cache, thrashing occurs. That is why it has to be decided which job should be promoted, thus, which deadline should be set to the current time. Therefore, there should be scheduling decisions which control the total amount of WSSs residing inside the cache. To do so, jobs with the smallest WSS are promoted and their deadline is set to the current time. They are then scheduled in *earliest deadline first* (EDF) order because this method has shown to improve the schedulability for real-time systems [4]. In addition, the total amount that currently resides inside the cache is maintained and the scheduler is in charge of controlling that the total available space is not exceeded when scheduling a job. It is important that a job that has been scheduled stays promoted until it is done, in order to prevent it from being evicted from the cache too early and also to improve fairness. That means if promoted once, a job is not demoted until it is scheduled even though a job with a smaller WSS has been released. In case, a job can not be scheduled because there is not enough free space allocable, one core will be idled and this job will not be scheduled. A core can be idled, if other jobs can be delayed without violating their deadlines in order to schedule a *tardy* job. There is a potential loss in terms of parallelism. However, it can be compensated by the speed of caches which are improving the performance [4].
Each job has a release date and a deadline by default. Thus, one problem can arise, when one job is executed after that deadline. This can happen when other prioritized jobs have been scheduled first. A job is then marked as tardy. To prevent that tardy jobs are executed too late, it is important that those jobs always have to be prioritized over promoted jobs [4].
This strategy can simply be implemented without much overhead. It can be realized by creating two different run queues, one queue for all promoted jobs and one that is EDF-ordered. In the promotion queue the scheduler pins the jobs with the smallest WSS to the front, while the EDF-ordered queue is not to be modified at the beginning [4].

As shown in Algorithm 1, at first, the scheduler looks at the first job of the EDF-queue. In case, it is a tardy job, it has to be scheduled next, otherwise, the first element of the promotion queue will be selected and scheduled next, providing that it does not cause thrashing [4]. According to its WSS and the current size of data residing inside the cache, a decision is to be made as described above.
One disadvantage is that in the specific case that a tardy job can not be scheduled next, because its WSS would not fit inside the cache and no core can be idled at that moment, then cache thrashing would occur and other data would be evicted [3]. However, that is an aspect that should be considered in later work.
On the contrary, this method effects an improvement of cache performances and additionally reduces task execution costs for

```
struct queue *promoted; *edf;
int curCacheSize; maxCacheSize;
// Mark if tardy
if edf->first->deadline < currentTime() then
    edf->first->tardy = true;
end
// Promotion
while edf->next != NULL do
    struct element *smallest = edf->first;
    if edf->next->wss < smallest->wss then
        smallest = edf->next;
    end
    push(smallest, promoted);
end
// Scheduling
if edf->first->tardy then
    if (curCacheSize + edf->first->wss) > maxCacheSize then
        idleCore;
    end
    schedule edf->first;
    curCacheSize += edf->first->wss;
else
    if (curCacheSize + promoted->first->wss) > maxCacheSize
      then
        idleCore;
    else
        schedule promoted->first;
        curCacheSize += promoted->first->wss;
    end
end
```

Algorithm 1: Order of scheduling. The task with the smallest WSS will be promoted by pushing it on the promotion queue. If a tardy job exists, it will be scheduled first. In case there is not enough free space and a core can be idled, the core will be idled and the job will be scheduled. Otherwise, it will also be scheduled, but thrashing will occur. If there is no tardy job, the first element of the promotion queue will be scheduled.

real-time tasks, since thrashing only occurs in rare cases. Therefore, for instance, extra computation can be supported for increasing a task's utility or it can be run with additional workload [3].

4.2 Computation of WSS

Since we now have an acceptable scheduler, the question arises how to define a correct WSS of each MTT and the answer comes with a cache profiler. A cache profiler only needs performance counters that are in many processors nowadays [4]. They count the amount of shared cache misses that occur while a job is being executed and calculate the per-job WSS [6]. Therefore, the counter has to be reset before a job is executed. The WSS results from the amount of total cache misses divided by the number of profiled jobs for an MTT and multiplied by the cache line size [4].

$$WSS = \frac{TotalCacheMisses}{NumberOfProfiledJobsForMTT} * CacheLineSize$$

Note that MTTs with a WSS that is bigger than the total size of the shared cache are not of interest.

Naturally, the only cache misses that occur primarily are when data is referenced the first time which leads to a compulsory miss. Additionally, the scheduler is not aware which job would cause thrashing because all WSSs of the tasks are initially unknown. In order to compute the WSS of a task, the profiler needs to be bootstrapped first. At the beginning, all WSSs are set to zero and the first jobs of each task are scheduled next. As there is no information about the size, these jobs will cause thrashing as they eventually exceed the size of the shared cache. In this case, the profiler resets the WSSs of the current jobs to the size of the cache and continues with the next jobs. Otherwise, if a job does not cause thrashing then its WSS can be calculated with the formula above. Thereafter, the jobs are scheduled as described in Section 4.1 which means that the job with the smallest WSS is prioritized and scheduled next. As a consequence, at that moment, jobs are mostly scheduled alone because they have a WSS that equals the size of the cache. Thus, no thrashing occurs and an approximate WSS can be estimated and updated. However, only if the WSS converges which means that the WSS is estimated a couple of times and the same value results repeatedly then the WSS is considered as correct [4].

For the profiler, it takes some time at the beginning until the WSSs leveled off to the correct size. However, afterwards it can result in better performance and avoiding of thrashing [4].

As a conclusion, the profiler is needed to compute the correct WSSs of all MTTs and thereby help the scheduler to decide which tasks should be promoted as well as to prevent cache misses [4].

5 EVALUATION AND DISCUSSION

To review cache optimization strategies in real-time systems, there are many techniques for optimizing the behavior of the cache that have their own advantages and disadvantages. The high cache miss rate that occurs without those strategies, underlines the need of a precise specification of the cache's behavior.

On the one hand, of course, each strategy causes overhead because different computations need to be done in order to improve the cache's interior structure.

On the other hand, however, the cache's behavior can be managed by scheduling tasks according to the presented algorithm. This can prevent cache misses which leads to a better performance and also can save energy, as less data has to be written back to main memory and fetched again in a short time. In addition, cache partitioning and cache locking are possibilities to control cached data, because it is well known where it is located at which time. Thereby, the duration of a task can be estimated which is advantageous in real-time systems because usually the worst case execution time has to be assumed. With these strategies, a reliable execution time can be estimated. Generally speaking, the presented strategies represent an overall positive way to get a better performance and to increase the predictability without too much overhead.

In the future, there should be more research on how to avoid one processor being idled if a job can not be scheduled due to the occurence of cache thrashing in order to increase parallelism.

REFERENCES

[1] Hendrik Borghorst and Olaf Spinczyk. 2015. Increasing the Predictability of Modern COTS Hardware through Cache-Aware OS-Design. In *Proceedings of the 11th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '15)*. 41–44. http://www.mpi-sws.org/~bbb/events/ospert15/pdf/ospert15-p41.pdf

[2] Hendrik Borghorst and Olaf Spinczyk. 2019. CyPhOS – A Component-Based Cache-Aware Multi-Core Operating System. In *Proceedings of the 32nd International Conference on Architecture of Computing Systems (ARCS '19)*. 171–182. https://doi.org/10.1007/978-3-030-18656-2_13

[3] John M. Calandrino and James H. Anderson. 2008. Cache-Aware Real-Time Scheduling on Multicore Platforms: Heuristics and a Case Study. In *2008 Euromicro Conference on Real-Time Systems*.

[4] John M. Calandrino and James H. Anderson. 2009. On the Design and Implementation of a Cache-Aware Multicore Real-Time Scheduler. In *2009 21st Euromicro Conference on Real-Time Systems*. 194–204. https://doi.org/10.1109/ECRTS.2009.13

[5] Mahmut T. Kandemir, Ibrahim Kolcu, and Ismail Kadayif. 2002. Influence of Loop Optimizations on Energy Consumption of Multi-bank Memory Systems. In *Proceedings of the 11th International Conference on Compiler Construction (Lecture Notes in Computer Science, Vol. 2304)*. Springer International Publishing, 276–292. https://doi.org/10.1007/3-540-45937-5_20

[6] Markus Kowarschik and Christian Weiß. 2003. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In *Meyer U., Sanders P., Sibeyn J. (eds) Algorithms for Memory Hierarchies. Lecture Notes in Computer Science*, Vol. 2625. Springer, Berlin, Heidelberg, 213–232. https://doi.org/10.1007/3-540-36574-5_10

[7] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and Ponnuswamy Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 367–378. https://doi.org/10.1109/HPCA.2008.4658653

[8] Sparsh Mittal. 2016. A Survey of Techniques for Cache Locking. *ACM Trans. Des. Autom. Electron. Syst.* 21, 3, Article 49 (may 2016), 24 pages. https://doi.org/10.1145/2858792

[9] Sparsh Mittal. 2017. A Survey of Techniques for Cache Partitioning in Multicore Processors. *ACM Comput. Surv.* 50, 2, Article 27 (may 2017), 39 pages. https://doi.org/10.1145/3062394

[10] Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. 2013. Outstanding Paper Award: Making Shared Caches More Predictable on Multicore Platforms. In *2013 25th Euromicro Conference on Real-Time Systems*. 157–167.