Impact of Resource Sharing on Performance and Performance Prediction: A Survey

Andreas Abel, Florian Benz, Johannes Doerfert, Barbara Dörr,
Sebastian Hahn, Florian Haupenthal, Michael Jacobs, Amir H. Moin,
Jan Reineke, Bernhard Schommer, Reinhard Wilhelm

Saarland University, Saarbrücken, Germany

Abstract Multi-core processors are increasingly considered as execution platforms for embedded systems because of their good performance/energy ratio. However, the interference on shared resources poses several problems. It may severely reduce the performance of tasks executed on the cores, and it increases the complexity of timing analysis and/or decreases the precision of its results. In this paper, we survey recent work on the impact of shared buses, caches, and other resources on performance and performance prediction.

1 Introduction

Multi-core processors are increasingly considered as execution platforms for embedded systems since they offer a good energy-performance tradeoff and seem to support transitions from federated to integrated system architectures in the automotive and avionics domains. Many applications implemented on such multi-core platforms are safety- and some also time-critical. A critical issue is the reduced predictability of such systems resulting from the interference of different applications on shared resources. These interferences can be at least of two kinds: Several applications may request a resource at the same time, but the resource can only admit one access at a time. As a consequence, an arbitration mechanism may delay the request of all but one application, thus slowing down the other applications. This is the case of resources like buses, typically called *bandwidth resources*. On the other hand, one application may also change the state of a shared resource such that another application using that resource will suffer from a slowdown. This is the case with shared caches, which fall into the class of *storage resources*. Most of the treatments of the interferences on shared resources found in the literature consider the detrimental effect of interferences. In the case of shared caches, however, the interference of one application A_1 on another co-running application A_2 could even speed up A_2 if A_1 would perform the right cache prefetching for A_2.

Interference on shared resources makes worst-case execution time (WCET) analysis of applications more difficult since a task or a thread can no longer be analyzed for its timing behavior in isolation. All potential interferences slowing down (or speeding up) the task under analysis have to be considered. This leads

to a combinatorial explosion of the analysis complexity, as all possible interleavings of different threads have to be analyzed. For that reason, currently, no sound timing-analysis method for multi-core platforms with shared resources exists.

This survey considers several aspects of the execution of sets of tasks on multi-core platform that have to do with the interference of the tasks on shared resources. One question is how the actual performance of tasks is slowed down by other co-running tasks. The other is how to compute bounds on the slow-down in order to derive guarantees for the timing behavior. A major problem is the increased complexity of this task compared to the single-task single-core case.

Caches are a particular case of *storage resources*. Several approaches exist for the treatment of shared caches in attempts to derive timing guarantees. Cache partitioning eliminates the interference between tasks. Static analysis of non-partitioned shared caches attempts to safely bound the interference. The definite comparison between these two approaches has yet to be done.

Buses are instances of *bandwidth resources*. Several protocols exist for the arbitration of shared buses, which can be classified as either time-driven, event-driven, or hybrid combinations of both. Static analysis can be used to determine good slot assignments in time-driven protocols like TDMA, and it can be used to determine bounds on the access delays in event-driven state-based protocols like FCFS and round robin.

1.1 Ways to derive guarantees

In order to guarantee the timeliness of tasks in a hard real-time system, one needs upper bounds on the execution times of the tasks.

As long as a task executes in isolation on a multi-core system (without co-running tasks), existing techniques for timing estimation could be applied. In case of parallel workloads (with co-running tasks), a sound approach for timing analysis of multi-core systems has to take into account the interferences, as described in detail in Sections 2 and 3.

Approaches to determine upper bounds on execution times of tasks on multi-core processors can be classified into two groups:

- Approaches achieving *performance isolation* by hardware and/or software techniques, e.g. by employing TDMA arbitration of busses. Performance isolation implies *timing composability* and permits the use of standard single-core timing analysis techniques with minor modifications. While this makes timing analysis comparatively easy, the challenge in such approaches is to make efficient use of shared resources by partitioning them appropriately for the given workload.
- Approaches analyzing the mutual effects of co-running tasks on each other's execution time. Such approaches require new timing analysis techniques that differ greatly from those employed in the single-core single-task case.

Different methods have been proposed or are pursued to derive guarantees for the timeliness of sets of tasks in a parallel workload setting when performance

isolation is not given. First, there is the classification according to whether the software is analyzed or executed.

- The *static analysis* of a whole set of concurrently executed applications may deliver a sound and precise guarantee for the timing behavior. The problem is the huge complexity of this approach.
- *Measurement-based* methods are in general not able to derive guarantees, neither in the single-core nor in the multi-core case.

The particular contribution to the execution-time bounds of the interference on shared resources can be dealt with in different ways:

- The *Murphy* approach assumes maximal interference on each access to a shared resource [1]. This assumption can be easily integrated into existing single-core timing analysis techniques. The *Murphy* approach will clearly give sound, but the most pessimistic execution-time bounds.
- The *slowdown factor* approach attempts to explicitly quantify the worst-case impact of the interferences in shared resources on the timing of a task caused by co-running tasks. The obtained slowdown factor can then be used to obtain an estimate on the execution times of a task in a parallel workload from an estimate in the isolated case. Existing approaches aim at quantifying the slowdown of a task in the worst case by measurement-based techniques. These measurement-based approaches employ so-called *resource-stressing benchmarks*, which are constructed for particular resources to produce the maximal slowdown on co-running tasks due to conflicts on this resource. In Section 4, we will see that attempts in this direction may be both unsound and overly pessimistic. Resource-stressing benchmarks are, in general, independent of the application that is slowed down by co-running these benchmarks. Therefore, one single resource-stressing benchmark can hardly slow down the application in the worst way. The fact that resource-stressing benchmarks might not be sound is demonstrated by an imaginary application-specific *worst companion* (see Section 4).
- Finally, the static analysis of a whole set of concurrently executed applications may deliver a sound and precise guarantee for the timing behavior. The problem is the huge complexity of this approach. To reduce analysis complexity, existing static analysis approaches separate analysis into two phases: The first phase determines a bound on the execution time of each task in isolation and a characterization of its resource-access behavior. The second analysis phase then uses this characterization to bound the impact of interference on the execution times of all tasks. The sum of the two bounds for each task then yields an estimate of the task's worst-case execution time. Such approaches are discussed in Section 2. For soundness, all of these approaches rely on *timing compositional* hardware architectures [2], which permit to account for the cost of interference in such a compositional way. Unfortunately, many existing hardware architectures exhibit domino effects and timing anomalies and are thus out of the scope of such an approach.

1.2 Terminology

We will need a few terms for this survey. The *(individual) access delay* is the interval between the time of an individual access request and the time when this request is granted. This can sometimes be determined by measurement. The *worst-case access delay* is the access delay in a worst-case scenario. It is typically derived from the arbitration protocol and some system parameters, e.g. the number of concurrently executed tasks and the slot size in bus protocols. The *overall access delay* is the sum of all the access delays encountered during a task's execution. An upper bound on the overall access delay is sometimes used in combination with a WCET estimate to compute a worst-case response time (WCRT) estimate.

2 Bandwidth Resources, in particular Shared Buses

In computer architectures, buses are used to transfer data between different components of a computer. The components allowed to access a particular bus are called the *accessors* of the bus. In order to reduce cost and complexity of the overall system, often only one accessor is allowed to access the bus at a time. In this case, the limited bandwidth of such a *shared bus* is shared solely along the dimension of time. We only consider such buses in the following.

Several accessors may attempt to access the bus at the same time and thus cause a *bus conflict*. Arbitration mechanisms resolve bus conflicts by only granting access to one accessor at a time. There are three main classes of resource arbitration mechanisms. *Time-driven arbitration* uses a predefined bus schedule, which assigns time slots of fixed size to particular accessors. This is commonly referred to as *TDMA* (Time Division Multiple Access). *Event-driven arbitration* mechanisms decide at runtime, which accessor is granted resource access next. These decisions usually depend on the access histories of all accessors. Prominent examples are *Round Robin* or *FCFS* (First-Come-First-Serve) resource arbitrations. *Hybrid arbitration* mechanisms split their arbitration period into segments of fixed length. Static segments use time-driven resource arbitration and dynamic segments use event-driven resource arbitration. A member of this class is the *FlexRay* [3] bus protocol used in the automotive industry [4,5].

One can distinguish between *synchronous* and *asynchronous* bus accesses. Requests for *asynchronous access* to a shared bus can be buffered until they are finally granted by the arbiter. This way, the requesting application can immediately continue its execution. Applications that only rely on asynchronous resource accesses can achieve the same performance as in combination with a dedicated bus provided sufficient buffer sizes, enough bandwidth, and sufficiently delayed accesses to the results. In contrast, requests for *synchronous access*, e.g. memory load requests, block the requesting application until they are granted by the arbiter. The blocking of requests for synchronous bus accesses leads to additional stalls of the requesting processing unit. Such bus interference effects may decrease the performance of a processing unit compared to a dedicated bus scenario on average as well as in the worst case.

In a multiprocessing environment, a processing unit may access a shared bus *explicitly* or *implicitly*. Explicit accesses are performed by commands occurring in the program, and can therefore be more easily analyzed than implicit accesses. A first-level cache, private to a processing unit, for example, accesses a shared memory bus for cache reloads. The cache-coherence protocol for private caches also accesses the shared bus. Implicit accesses are harder to track by static analysis than explicit accesses since they are subject to more uncertainty.

2.1 Analysis Approaches and Task Model

Naive bounds on the WCET and WCRT of tasks assume bus access requests to always be granted immediately. They are obviously not sound in the presence of bus interferences. A common approach is to first derive naive bounds on the WCET and to account for possible interference effects in an additional analysis step focussed on the bus interference.

Literature on worst-case timing analysis for systems with shared bus resources and synchronous bus accesses is mainly concerned with this additional analysis step. The common assumption is that adding the access delays to the naive timing bounds will lead to sound WCET bounds. This can only be guaranteed for compositional timing models of hardware platforms [2].

To simplify the analysis of bus interference, all surveyed approaches adopt a *task model* based on so-called superblocks. Each task is described as a sequence of superblocks (see Figure 1). Upper bounds on the amount of computation time ($exec_i$) and the number of resource accesses (μ_i) per superblock (s_i) serve as input and are used as common abstraction of tasks [6]. The bounds on the amount of computation time are assumed to be the result of a naive WCET analysis ignoring completely the time needed to access the bus. For simplicity, our figures assume that a bus access always takes one time unit to be served once it is granted. In the presence of implicit bus accesses it can be a major challenge to derive tight bounds on the number of resource accesses of a superblock. This is an open research problem.

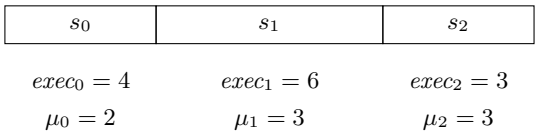

Figure 1. A task consisting of three superblocks

The approaches in the literature differ in the class of resource arbitration considered. While Pellizzoni et al. [7,8] are concerned with event-driven resource arbitration mechanisms, Schranzhofer et al. [6] treat TDMA resource arbitration. The determination of safe timing bounds in the presence of a shared bus and a hybrid arbitration mechanisms is described in Schranzhofer et al. [9].

Time-driven Resource Arbitration An inherent property of time-driven resource arbitration is that the overall access delay experienced by a task on one processing unit is independent of the access requests issued by concurrent accessors. Thus, current approaches for worst-case timing analysis of systems with shared TDMA buses bound the overall access delay of a given task by considering its bus accesses and their possible distributions across a known bus schedule [6]. Optimization approaches try to find a pair of task assignment and bus schedule leading to a low WCRT bound [10].

The overall access delay of the processor under consideration (PuC) is maximized when it is blocked as long as possible by the bus arbitration. The PuC is blocked whenever it tries to access the bus in a time slot assigned to another accessor. Furthermore, it is blocked in a time slot assigned to itself if its access request is released too late to be completed in the current time slot.

The amount of overall blocking experienced by a superblock depends on how its bus accesses and its computation time are distributed within the superblock. Possible distributions are restricted by the given upper bounds on the number of bus accesses and the computation time per superblock. Intuitively, a bus access released early enough in a time slot assigned to the PuC does not lead to any blocking of the PuC. In order to maximize the overall blocking, a distribution of accesses and computation time in the superblock should avoid such access releases as far as possible.

Algorithms to calculate a WCRT bound for a superblock run through a sequence of points in time starting with the worst-case release time of the superblock. For each point in time, they decide if a bus access or a certain amount of computation time could lead to a globally maximized blocking. This decision determines which point in time has to be considered next. Uncertainty in this decision may lead to case splits. Termination is guaranteed as the maximal amount of resource accesses and computation time of a superblock will be consumed at some point in time. Such algorithms construct a distribution of bus accesses and computation time leading to a tight bound on the WCRT. To avoid case distinction and thereby reduce the complexity of the problem, it is possible to use coarse under- and overapproximations as intermediate result. This interval can be further refined using a binary search that excludes candidates guaranteed to be infeasible [6].

A superblock with its upper bounds on the computation time and the number of bus accesses is an abstraction of a fragment of a task. Obviously, these upper bounds may allow for different distributions of bus accesses and computation time across the superblock than those found in the original task. As a consequence, the worst-case distribution of accesses determined by an algorithm may actually be infeasible. The amount of pessimism introduced by the superblock abstraction remains to be evaluated.

In order to decrease the upper bounds on the WCRT of a task, it is beneficial to split it up into superblocks in a way that each superblock either only performs computations or only performs bus accesses. This is referred to as *dedicated access model* [6]. A deterministic execution model for time-critical systems

introduced by Boniol et al. [11] distinguishes between execution and communication slices in a similar way. The reason for the observed improvement is that the bounds on the WCRT of a superblock are the consequence of very unbeneficial interleavings of its bus accesses and its computation time with respect to the bus schedule's time slots.

However, it is not clear at which granularity superblocks should be abstracted from a task's code. Furthermore, it remains unclear which restrictions a task's code has to fulfill in order to be abstracted to superblocks following the dedicated access model. A simple possibility is to enforce a strict separation of computation and bus accesses by the programmer. This is, however, not possible in the presence of implicit bus accesses, which are not visible to the programmer and have to be detected by a consecutive static analysis.

Event-driven Resource Arbitration It is a common assumption in the literature that the latency of an individual resource access of the PuC can be bounded. Either an arbitration protocol provides this property to all accessors, or at least to the PuC. In the presence of event-driven resource arbitration, there are two approaches to bound the effect of interference, discussed in more detail below:

1. By taking into account the arbitration logic.
2. By taking into account the amount of competing resource accesses.

In both approaches, the common principle is to construct the worst-case scenario, that is the scenario that maximizes the latency of a request based on the knowledge about the interference.

Knowledge about the maximum number of bus accesses in a superblock of the PuC allows to bound its overall access delay: For round-robin arbitration, the worst-case scenario is that all other processing units are allowed to perform one access before the PuC is allowed to do so [7]. In FCFS arbitration, an interference bound has to additionally take into account the maximum number of access requests that can be released at the same time by concurrent processing units if this number is greater than one [8]. We call this *delay bounding based on the arbitration logic*. Of course these bounds are only tight provided that the concurrent processing units can really issue this number of bus requests while the superblock is executed. This remark leads us to the second bounding factor on the interference.

It is an inherent property of event-driven arbitration mechanisms, that a processing unit can only be blocked if and when another processing unit is requesting bus access. Furthermore, many event-driven arbitration mechanisms do not allow to interrupt and restart a bus access once it is granted. Provided that these two properties hold, a superblock on the PuC cannot experience more overall access delay than the time needed to perform all the accesses concurrent accessors might release while the superblock is executed [7,8]. We refer to this as *delay bounding based on the amount of concurrently requested access time*.

In order to bound the amount of concurrently requested access time for a given superblock, all possible interleavings of its bus accesses with concurrent

access sequences have to be taken into account. As this may be computationally infeasible in general, arrival curves [12,13] are introduced as an abstraction of the concurrent access behavior. They bound the maximal amount of access time that concurrent accessors might request in a time interval of given length. It is possible to obtain them from the superblock abstractions of the task on concurrent processing units [8]. Let function $R(t)$ describe the cumulative amount of access time requested by a concurrent accessor until time t. An arrival curve $\alpha(\Delta)$ bounds the amount of access time that can be requested in any time interval of length Δ:

$$\forall t_1 : \ \forall t_0 \leq t_1 : \ R(t_1) - R(t_0) \leq \alpha(t_1 - t_0)$$

The use of arrival curves for more than one concurrent accessor inherently overapproximates the experienced blocking. A longer running superblock can potentially suffer from more delay caused by a particular concurrent accessor than a quickly executed superblock. Therefore, the involved fixed point iteration has to pessimistically assume for each concurrent accessor that it can already profit from the delay caused by all other concurrent accessors. Yet, this loss of precision is bearable regarding the complexity of considering all possible interleavings of bus access sequences of the concurrent accessors.

In addition, arrival curves also incorporate concurrent access sequences that can possibly never be executed in parallel with a particular superblock. Therefore, arrival curves may introduce additional pessimism with respect to the concurrent access request behavior if that behavior exhibits a heterogenous structure along the dimension of time.

Hybrid Resource Arbitration For hybrid resource arbitration, it is more challenging to find a distribution pattern of bus accesses and computation time across the bus arbitration segments that leads to a globally worst response time. When constructing worst-case scenarios, the static and dynamic segments need to be taken into consideration jointly.

While it would be feasible to derive worst-case scenarios considering only the static or only the dynamic segments with the methods described above, the derived bounds would likely be very pessimistic.

To avoid high over-estimation, a dynamic programming approach presented by Schranzhofer et al. [9] enumerates all possible distributions of accesses in a superblock and interfering accesses of concurrent processing units across the segments of the bus arbitration.

2.2 Summary of the State of the Art & Open Problems

It is possible to determine safe bounds on the WCRTs of tasks for systems with synchronous accesses to a shared bus. Upper bounds on the naive WCETs and the number of bus accesses for each superblock are assumed as input. Approaches described in the literature treat different classes of bus-arbitration mechanisms.

Each of the approaches is backed by its own separate, rigorous formalism. It would be beneficial to identify more generic approaches to reason about all classes of arbitration mechanisms treated so far as well as possible future ones.

Furthermore, the current level of abstraction (superblock-based task model, arrival curves) is quite high and possibly results in severe overestimation of the actual interference. It would be valuable to examine the tradeoff between analysis efficiency and precision by studying more precise task characterizations.

All presented approaches require the bounds on the amount of computation time and the access times and delays to be compositional. Therefore most authors assume a timing-compositional hardware platform [2]. Such a platform allows for a precise description of its temporal behavior by a compositional timing model. Future work should consider the treatment of hardware platforms that currently only allow for a precise timing analysis in combination with models exhibiting timing anomalies.

3 Storage Resources, in particular Shared Caches

Caches are used to bridge the large latency gap between modern processor pipelines and main memory. They are small, low-latency on-chip memories that buffer a subset of the contents of the main memory. Cache replacement logic decides at runtime which memory blocks to store in the cache. Sequential programs running on single-core machines usually exhibit high *spatial* and *temporal locality* and thus experience a high cache-hit ratio. As a consequence, the average memory access latency is close to that of the cache in such a scenario. Precise and efficient static analysis of the behavior of a private cache is possible if the memory accesses of the application and the replacement logic are predictable [14].

Current and upcoming multi-core processors feature *private* first- and sometimes second-level caches and *shared* higher-level caches. In theory, large shared caches promise a more efficient use of expensive die area than an array of small private caches of the same aggregate capacity: their capacity can be shared flexibly according to the needs of the programs running on the connected cores. In addition, if applications running on different cores share data, they can communicate more efficiently through the shared cache than through main memory.

Unfortunately, the behavior of current shared caches is hard to predict statically. The reason for their unpredictability is the interference between accesses originating from different cores. As with buses, caches cannot serve multiple memory accesses completely in parallel. Pipelining of accesses is possible, but some "bandwidth interference" remains. However, the main challenge with shared caches is that the state of the cache depends on the precise interleaving of accesses from multiple cores. Accesses from different cores are typically served on a first-come first-served basis. Their interleaving thus depends on the relative execution speeds of the applications running on these cores, which—among other things—depend on their cache performance, which in turn depends on the cache state. This cyclic dependency between the interleaving of the accesses

and the cache state makes precise and efficient analysis hard or even impossible in general. Additional complications—not detailed in this survey—arise from coherence protocols, which need to be employed when applications running on different cores share memory. Now, even if a shared-cache analysis taking into account the interactions of multiple applications would exist, such an analysis would have to have precise knowledge of all applications, their mapping, and their scheduling. This would render independent and incremental certification impossible.

Maybe unsurprisingly, it has been observed that uncontrolled sharing of caches that does not take into account that cached memory blocks belong to different cores or applications is also detrimental from an average-case performance perspective. For example, an application with low temporal and spatial locality that generates many first-level cache misses can acquire a large portion of the shared cache without any benefit to system performance.

A common approach found in recent literature is to take into account the core or application a cache block belongs to when deciding which block to replace. Typically, the cache is conceptually partitioned among the cores, so that memory blocks compete for cache space only with other memory blocks of the same core. Within each partition a common replacement policy such as least-recently used is applied. The partition sizes are chosen with different objectives in mind, such as aggregate performance and fairness. In addition or even instead of varying partition sizes, some approaches also adapt the schedule of applications with these objectives in mind. Both decisions about scheduling and partition sizes can happen either statically or dynamically and are taken based on a static or dynamic characterization of the different core's memory access behavior.

3.1 Cache Partitioning

In the following, we provide a non-exhaustive survey of the vast existing literature on

- methods to partition caches,
- approaches to determine good partition sizes, and
- methods to schedule tasks taking into account a shared cache.

How to Partition a Cache
There are various approaches to partition caches. They can be distinguished by

- the partitioning scheme: set-based [15,16] or way-based [17,18], and by
- their implementation: in software [15,16] or in hardware [19,17,18].

In set-based partitioning, each core is given exclusive access to a subset of all cache sets. This can be realized both in hardware and in software. Hardware-based solutions can realize set-based partitioning by adapting the mapping of memory blocks to cache sets depending on the core that issued the memory access. If virtual memory is employed and the shared cache is physically-indexed, set-based partitioning can also be realized in software by *page coloring* [20]:

In page coloring, physical pages mapping to the same set of cache sets are assigned the same *color*. Then, the set of colors is partitioned among the different processes. Virtual pages belonging to a particular process are only mapped to physical pages of the colors assigned to that process.

In way-based partitioning, the cache is partitioned along the cache ways: each core is given exclusive access to a subset of all cache ways. This can only be realized in hardware, by accordingly adapting the cache replacement logic.

Both schemes have their advantages and drawbacks: In particular in low-associativity caches, way-based partitioning only allows for a coarse-grained allocation of the cache space. Set-based partitioning, in particular its hardware-based variant, allows for a more fine-grained allocation of the cache space as the number of ways usually exceeds a cache's associativity. The main drawback of set-based partitioning is that changing partition sizes is expensive. Changing the coloring of pages or changing the mapping of memory blocks to cache sets implies that the cached data needs to either be invalidated or moved to their new locations in the cache. As a consequence, hardware-based solutions usually rely on way-based partitioning, in which changing partition sizes is cheap. Software-based solutions are forced to rely on set-based partitioning.

How to Determine Sizes of Partitions The choice of partition sizes has a strong effect on the performance of each corunning task, and thus the system performance. Multiple metrics have been proposed to measure system performance, the most common metric being *throughput*, i.e. the sum of the IPCs (instructions per cycle) of all corunning tasks. Other metrics combine performance with fairness goals.

To arrive at a good partition in terms of a particular performance metric, the effect of a partition size on each corunning task needs to be characterized in some form. The following characterizations have been used to drive the partitioning choice:

- *Miss ratio curves* [15] capture the miss ratio of a task in terms of the task's cache partition size. Unfortunately, miss ratios, i.e., ratios between cache misses and memory accesses, are not directly correlated with execution times. A high miss ratio does not imply that the task spends much time waiting for memory accesses. Thus, minimizing miss ratios does not necessarily maximize system throughput. A related metric proposed by Tam et al. [21] are stall-rate curves, which capture the share of overall execution time a task is stalled for memory.
- *Misses per 1000 instruction (MPKI) in terms of cache size* [18]. This metric is much more closely tied to the cache's impact on execution times. Still, it may not be perfectly correlated with the resulting performance in terms of cycles per instruction.
- Cycles per instruction (CPI) in terms of cache size [18]. This would be the ideal metric when optimizing throughput. However, in contrast to MPKI, it is hard to determine efficiently.

The vast majority of the literature is concerned with general-purpose computing, in which the set of tasks changes dynamically and is not known in advance. In this scenario, the characterization of the tasks needs to be performed at runtime. Qureshi et al. [18] propose dynamic set sampling as an efficient and fairly accurate mechanism to estimate the MPKI. The idea is to focus on a small subset of all cache sets and to extrapolate from the observations in this subset to the entire cache. On this small subset, the cache is augmented with additional tags, which are used to dynamically evaluate the effect of providing more cache ways to a particular task. With these additional tags, the hardware is able to count the potential number of cache hits due to each additional cache way.

In a hard real-time context, the task set is known in advance, and a reasonable characterization would be bounds on the WCET of each task in terms of its partition size.

Given the characterization of each of the tasks, dynamic approaches then predict from the past behavior of the tasks, which choice of partitioning is likely to maximize the chosen performance metric in the future. Depending on the partitioning scheme, a reconfiguration overhead needs to be taken into account. This may be particularly severe in case of software-based partitioning. Zhang et al. [15] discuss tradeoffs in a page coloring approach.

Scheduling Approaches In addition to choosing good partition sizes for a given workload, system performance may be improved by optimizing the schedule. This has been considered for general-purpose tasks and his particularly relevant in the hard real-time case, as all the necessary information is present at design time.

Zhao et al. [22] present an approach to *dynamic scheduling* that is based on their *CacheScouts* monitoring architecture. This architecture provides hardware performance counters for shared caches that can detect how much cache space individual tasks use, and how much sharing and contention there is between individual tasks. The authors describe three *dynamic* scheduling heuristics that utilize this architecture:

- Schedule a waiting task that has significant sharing with other already running tasks.
- Schedule a task that still has its working set left in the cache.
- Co-schedule tasks with a small and a large working set.

An approach to *static scheduling* in a hard real-time context is presented by Guan et al. [23]. They extend the classical real-time scheduling problem by associating with each task a required cache partition size. They propose an associated scheduling algorithm, *Cache-Aware Non-preemptive Fixed Priority Scheduling*, FP_{CA}. FP_{CA} schedules a job J_i, if

- J_i is the job of highest priority among all waiting jobs,
- there is at least one idle core, and
- enough cache partitions to execute J_i are available.

They propose a linear programming formulation that determines whether a given task set is schedulable under FP_{CA}. For higher efficiency, they also introduce a more efficient heuristic schedulability test that may reject schedulable task sets.

3.2 Summary of the State of the Art & Open Problems

Much work has been done in general-purpose computing to optimize average-case performance by cleverly partitioning shared caches. Much less has been done in the context of hard real-time systems. However, in the case of cache partitioning, hard real-time systems may in fact present a simpler problem: A major remaining challenge is that no real-time scheduling policy taking into account cache space demands is established. The work of Guan et al. [23] is a step in this direction. However, it takes the required cache partition size of each task as an input. Choosing partition sizes to optimize schedulability is an open problem.

In this survey, we have focussed on shared caches. Other storage resources that are increasingly shared in embedded systems are DRAMs and Flash memory. Recently, real-time memory controllers for SDRAM have been proposed that provide bandwidth and latency guarantees to their clients [24,25,26]. These controllers are based on a combination of predictable arbitration mechanisms, such as TDMA, and DRAM-specific access patterns that eliminate the dependence of latencies on the execution history.

4 Assessing the Impact of Resource Sharing on Performance by Measurements

As described earlier in Section 1.1, one possibility to estimate the timing behavior of tasks is by measurements. In this section, we discuss existing measurement-based approaches that aim at quantifying the slowdown a task experiences when different tasks are executed in parallel.

In a single-core setting, a measurement-based estimate is obtained by multiplying the longest observed execution time by a safety margin. However, it is not possible to directly extend such measurement-based timing analyses from the single-core to the multi-core setting.

First, consider timing analysis before tasks have been mapped to cores and scheduled. Given a set of k tasks executing on a system with n cores ($k > n$), there exist $\binom{k}{n}$ possible workloads that could be executed in parallel. Hence, without additional knowledge about the mapping and the schedule, it would be necessary to measure the slowdown of a task in each possible workload in order to determine its worst-case timing. This is very expensive by means of analysis time, especially as all measurements have to be repeated in case the task set changes. Therefore, it is desirable to determine workload-independent estimates of the slowdown. In case of timing analysis after the mapping process, this is not a problem.

Second, the choice of a proper safety margin becomes harder due to the increased variance in execution times. The safety margin is needed to compensate for the potential difference between the highest measured execution time and the actual WCET. With increasing variance, the safety margin must thus be increased as well. Thus the application of the safety margin might lead to a strong overestimation of the WCET in case the highest measured execution time is close to the actual WCET.

To obtain a workload-independent estimate of the slowdown, so-called resource-stressing benchmarks are employed as co-runners during the measurements. A resource-stressing benchmark is a synthetic program that tries to maximize the load on a resource or a set of resources. The interference a resource-stressing benchmark causes on a certain resource is meant to be an upper bound to the interference any real co-runner could cause. Therefore the slowdown a program experiences due to interferences on a certain resource when co-running with a resource-stressing benchmark bounds the slowdown that could occur with respect to this resource in any real workload.

Typically, resource-stressing benchmarks are independent of the application under consideration, but specific to an architecture. They are independent of the application so they can be reused for the analysis of different programs. The architecture under consideration must be taken into account for properly constructing e.g. a benchmark that maximally stresses the bandwidth to main memory: it must be guaranteed that none of the loads performed by the benchmark can be served by any of the caches. Thus the offset between the addresses has to be chosen in such a way that no cache line is accessed twice. For the proper choice of this parameter, architectural information about the width of a cache line is needed.

As an additional aspect, the obtained slowdowns can be used to estimate the timing predictability of an architecture. A significant slowdown implies a possibly high variance in the execution times of a task and thus disallows accurate timing analysis. Hence, it is also a measure for the suitability of a multi-core architecture for time-critical embedded systems.

In the context of hard real-time embedded systems, measurement-based approaches are inappropriate because they cannot derive safe, analytical guarantees. This is because it cannot be guaranteed that the actual WCET is encountered during the measurements. Even the highest measured execution time together with the safety margin is not necessarily an upper bound to the WCET. However, no static analysis that soundly accounts for all interferences in a multi-core architecture, has been proposed so far.

In Section 4.1, we present the existing approaches in more detail. In Section 4.2, we discuss the shortcomings of the measurement-based approaches and pose open questions.

4.1 Resource-Stressing Benchmarks

Radojković et al. [27], Nowotsch et al. [28] and Fernandez et al. [29] all employ resource-stressing benchmarks in order to quantify the slowdown of tasks in

a multi-core architecture. In all three approaches, the respective benchmarks are systematically constructed based on the characteristics of the resource to be stressed. A benchmark that aims at stressing, e.g. parts of the memory hierarchy should almost only involve memory operations and avoid local computations.

The approaches differ in the resources and architectures under consideration as well as in the respective evaluation techniques.

Radojković et al. [27] propose benchmarks that stress a variety of possibly shared resources, including functional units, the memory hierarchy, especially the caches at different levels and the bandwidth to the main memory. Experiments are carried out on three architectures with different shared resources in order to show the varying timing predictability, depending on the degree of resource sharing. One architecture offers hyperthreading, i.e. all resources including the pipeline are shared. For the second architecture, only the bandwidth to the main memory is shared between two cores whereas for the third architecture, the L2 cache is shared as well. In order to show the variance of the possible slowdown, measurements are taken for three different scenarios. In the first scenario, only resource-stressing benchmarks are executed concurrently to estimate the worst possible slowdown independently of the application. In the other cases, the application is either executed with another co-running application or with different co-running resource-stressing benchmarks.

Unsurprisingly, the more resources are shared, the larger is the possible impact of resource sharing on execution times. This makes techniques like hyperthreading impractical for systems with hard real-time constraints. The measurements for the different scenarios reveal that the slowdown due to co-running resource-stressing benchmarks drastically exceeds the slowdown measured in workloads only consisting of real applications. This implies that the workload-independent slowdown determined with co-running resource-stressing benchmarks yields a very imprecise upper bound to the slowdown in any real workload. Therefore, the analysis result might be not very useful in practice.

Nowotsch et al. [28] present benchmarks that stress the memory hierarchy, i.e. the bus to main memory and the caches. The slowdown is measured for workloads exclusively consisting of resource-stressing benchmarks. Several scenarios are considered, testing the influence of different memory configurations (static RAM vs. dynamic RAM) and cache coherency settings on the variance of the observed slowdowns. The overhead due to static coherency (i.e. only checks whether memory blocks in the local caches are coherent) is determined by enabling the coherency flag during measurements although there are no coherent accesses. The overhead due to dynamic coherency (i.e. not only checks, but also explicit memory operations enforcing coherency) is assessed by enabling the coherency flag during measurements in a setting where coherent accesses occur. Measurements are carried out on the Freescale P4080 multi-core processor which is used in avionics.

The outcomes show that the time for concurrent accesses to DDR-SDRAM memory scales very badly with the number of concurrent cores, in contrast to SRAM. Concerning the different cache coherency settings, the results show that

even without coherent accesses, static coherency induces an overhead in execution time that should be considered in timing analysis. The impact of dynamic coherency strongly depends on the type of memory operation (read or write). In case of read with concurrent read, dynamic coherency does not cause any slowdown compared to static coherency. For write operations, the execution is slowed down significantly in case of dynamic coherency, regardless of the concurrent operation. This can be explained by the fact that after a write operation, coherency actions are required to keep the memory hierarchy consistent.

The benchmarks used by Fernandez et al. [29] focus on the memory hierarchy of the system, including the caches at different levels as well as the memory controller. In order to evaluate the influence of the underlying operating system on the slowdown due to shared resources, the measurements are performed once on Linux and once on the real-time operating system RTEMS. The overhead in execution time is determined for two types of workloads: task sets exclusively composed of resource-stressing benchmarks and task sets with one application and one co-running resource-stressing benchmark.

The outcomes of the experiments with benchmarks that exclusively stress the private L1 cache through store operations show that a considerable slowdown is produced. This is due to the fact that the L1 cache is write-through, thus store operations lead to bus traffic, L2 cache interference and memory controller accesses. The number of store instructions within an application is identified to be the dominant metric for slowdowns in this architecture. Employing write-back caches reduces this overhead but introduces new overhead arising from cache coherency protocols, as it has been described in [28]. The results produced for the different operating systems show that they have a non-negligible influence as well. For example, whereas the estimated miss rate is 11% for a certain application in a Linux environment, the same application shows a miss-rate of near zero on the RTEMS operating system. Similar to the outcomes of the other papers, the application-independent slowdown measured in workloads with only resource-stressing benchmarks exceeds the application-dependent one.

4.2 Summary of the State of the Art & Open Problems

The state of the art approaches provide measurement-based estimates on the slowdown of a task due to interferences on shared resources caused by co-running tasks. These estimates are then used to roughly classify components of a multi-core architecture with respect to timing predictability. There are two main concerns about the soundness of the measurement-based approaches employing resource-stressing benchmarks.

First, there is neither a proof nor a formal argument why a specific benchmark puts maximal load on a resource. For the storage resources in particular, it is very difficult to argue why their state is affected in the worst possible way. In case of several possible co-runners, these arguments become even more difficult because of the combined interferences.

Second, a resource-stressing benchmark is typically application-independent. An application with memory-intensive as well as computation-intensive parts is

neither slowed down maximally by a cache-stressing nor by a functional-unit-stressing co-runner. To account for this behavior, either a sound combination of slowdowns obtained by different benchmarks or an application-specific benchmark is needed. Obtaining a sound combination of slowdowns, which we call *compositionality issue*, is hard because resources are not independent of each other. For example, a program that stresses the memory bus does also necessarily stress the last-level-cache. But, a benchmark that stresses several resources simultaneously cannot stress one particular resource in the worst possible way at the same time. We call the application-specific benchmark that causes the maximal slowdown *worst companion*. In general, the construction of this worst companion might be infeasible due to limited control over the hardware itself. Nevertheless, application-specific benchmarks might produce worse slowdowns than application-independent benchmarks which disproves the soundness of these measurement-based techniques.

Beyond the open questions concerning the soundness of the approaches, the usefulness of the obtained slowdowns is questionable. While a resource-stressing benchmark can cause slowdown factors of up to 20, in reality, applications are co-located with other applications that typically cause much less interferences on shared resources. Radojković et al. [27] demonstrate that the slowdown an application experiences with co-running resource-stressing benchmarks is significantly higher compared to other co-running applications.

An orthogonal research problem is the construction of supporting benchmarks that e.g. use cache prefetching to speed up the application. Analogously to the worst companion, one could aim at the construction of a *best companion*.

5 Conclusions

This survey has given an overview of the impact of the interference on shared resources on performance and performance estimation. Goals of the described approaches were performance estimation or performance improvement. Mainly, two types of resources were considered, bandwidth resources, e.g. buses, and storage resources, e.g. caches.

Performance estimation methods for multi-core systems with shared buses attempt to derive bounds on the overall access delays under different arbitration protocols. All use cumulative abstractions such as the number of bus accesses during bounded-length phases in tasks in a simplified task model. Different arbitration protocols have different worst-case scenarios in which bounds on the access delays are computed.

Static analysis of the cache behavior of shared caches is highly complex due to the large number of potential interleavings to be considered. Cache partitioning among the different tasks on different cores is used instead. Methods known from the single-core case can then be used. Performance of statically shared caches will suffer for systems with dynamically varying demands on memory. Dynamic cache partitioning is the answer. Existing approaches consider shared

caches in isolation and ignore the effect cache reloads and write backs have on the necessarily shared bus.

Several approaches attempt to construct resource-stressing benchmarks. Such a benchmark for a specific resource is meant to cause the maximal slowdown on a co-running application. In an interval of bounded length any resource can only be stressed to a degree that is a function of the interval length. The presented ways to design resource-stressing benchmarks do this in an intuitive, but ad-hoc way. They are independent of any particular application. It seems clear that a larger slowdown of a particular application can be achieved by an application-specific *worst companion*, which better exploits the available time to exercise stress on shared resources. Thus the existing resource-stressing benchmarks do not exhibit upper bounds on the interference. On the other hand, they are overly pessimistic: a co-running application will seldom exercise such a large stress on a shared resource.

References

1. Ungerer, T., Cazorla, F., Sainrat, P., Bernat, G., Petrov, Z., Rochange, C., Quinones, E., Gerdes, M., Paolieri, M., Wolf, J., Casse, H., Uhrig, S., Guliashvili, I., Houston, M., Kluge, F., Metzlaff, S., Mische, J.: Merasa: Multicore execution of hard real-time applications supporting analyzability. IEEE Micro **30** (2010) 66–75
2. Wilhelm, R., Grund, D., Reineke, J., Schlickling, M., Pister, M., Ferdinand, C.: Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems **28**(7) (July 2009) 966–978
3. FlexRay Consortium: FlexRay. `http://www.flexray.com/`
4. BMW: BMW technology guide: Flex Ray. `http://www.bmw.com/com/en/insights/technology/technology_guide/articles/flex_ray.html`
5. Robert Bosch GmbH: FlexRay communication controller IP. `http://www.bosch-semiconductors.de/en/ipmodules/flexray/flexray.asp`
6. Schranzhofer, A., Chen, J.J., Thiele, L.: Timing analysis for TDMA arbitration in resource sharing systems. In: Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium. RTAS '10, Washington, DC, USA, IEEE Computer Society (2010) 215–224
7. Pellizzoni, R., Caccamo, M.: Impact of peripheral-processor interference on WCET analysis of real-time embedded systems. IEEE Transactions on Computers **59** (2010) 400–415
8. Pellizzoni, R., Schranzhofer, A., Chen, J.J., Caccamo, M., Thiele, L.: Worst case delay analysis for memory interference in multicore systems. In: Proceedings of the Conference on Design, Automation and Test in Europe. DATE '10, 3001 Leuven, Belgium, Belgium, European Design and Automation Association (2010) 741–746
9. Schranzhofer, A., Pellizzoni, R., Chen, J.J., Thiele, L., Caccamo, M.: Timing analysis for resource access interference on adaptive resource arbiters. In: Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium. RTAS '11, Washington, DC, USA, IEEE Computer Society (2011) 213–222
10. Rosen, J., Andrei, A., Eles, P., Peng, Z.: Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In:

Proceedings of the 28th IEEE International Real-Time Systems Symposium. RTSS '07, Washington, DC, USA, IEEE Computer Society (2007) 49–60
11. Boniol, F., Cassé, H., Noulard, E., Pagetti, C.: Deterministic execution model on COTS hardware. In: ARCS. (2012) 98–110
12. Cruz, R.L.: A calculus for network delay. IEEE Transactions on Information Theory **37**(1) (1991) 114–141
13. Wandeler, E., Thiele, L., Verhoef, M., Lieverse, P.: System architecture evaluation using modular performance analysis: a case study. Int. J. Softw. Tools Technol. Transf. **8**(6) (October 2006) 649–667
14. Reineke, J., Grund, D., Berg, C., Wilhelm, R.: Timing predictability of cache replacement policies. Real-Time Systems **37**(2) (2007) 99–122
15. Zhang, X., Dwarkadas, S., Shen, K.: Towards practical page coloring-based multi-core cache management. In: Proceedings of the 4th ACM European conference on Computer systems. EuroSys '09, New York, NY, USA, ACM (2009) 89–102
16. Suhendra, V., Mitra, T.: Exploring locking & partitioning for predictable shared caches on multi-cores. In: Proceedings of the 45th annual Design Automation Conference. DAC '08, New York, NY, USA, ACM (2008) 300–303
17. Nesbit, K.J., Laudon, J., Smith, J.E.: Virtual private caches. SIGARCH Comput. Archit. News **35**(2) (June 2007) 57–68
18. Qureshi, M.K., Patt, Y.N.: Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In: IEEE/ACM International Symposium on Microarchitecture. MICRO '06, IEEE Computer Society (2006) 423–432
19. Xie, Y., Loh, G.H.: PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In: Proceedings of the 36th annual international symposium on Computer architecture. ISCA '09, New York, NY, USA, ACM (2009) 174–183
20. Taylor, G., Davies, P., Farmwald, M.: The TLB slice – a low-cost high-speed address translation mechanism. SIGARCH Comput. Archit. News **18**(3a) (May 1990) 355–363
21. Tam, D., Azimi, R., Soares, L., Stumm, M.: Managing shared L2 caches on multicore systems in software. In: Workshop on the Interaction between Operating Systems and Computer Architecture. (2007)
22. Zhao, L., Iyer, R., Illikkal, R., Moses, J., Makineni, S., Newell, D.: CacheScouts: Fine-grain monitoring of shared caches in CMP platforms. In: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques. PACT '07, Washington, DC, USA, IEEE Computer Society (2007) 339–352
23. Guan, N., Stigge, M., Yi, W., Yu, G.: Cache-aware scheduling and analysis for multicores. In: Proceedings of the seventh ACM international conference on Embedded software. EMSOFT '09, New York, NY, USA, ACM (2009) 245–254
24. Akesson, B., Goossens, K., Ringhofer, M.: Predator: a predictable SDRAM memory controller. In: CODES+ISSS, ACM (2007) 251–256
25. Paolieri, M., Quiñones, E., Cazorla, F., Valero, M.: An analyzable memory controller for hard real-time CMPs. IEEE Embedded Systems Letters **1**(4) (2010) 86–90
26. Reineke, J., Liu, I., Patel, H.D., Kim, S., Lee, E.A.: PRET DRAM controller: bank privatization for predictability and temporal isolation. In: Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis. CODES+ISSS '11, New York, NY, USA, ACM (2011) 99–108

27. Radojković, P., Girbal, S., Grasset, A., Quiñones, E., Yehia, S., Cazorla, F.J.: On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. ACM Trans. Archit. Code Optim. (2012)
28. Nowotsch, J., Paulitsch, M.: Leveraging multi-core computing architectures in avionics. In: EDCC. (2012)
29. Fernandez, M., Gioiosa, R., Quiñones, E., Fossati, L., Zulianello, M., Cazorla, F.J.: Assessing the suitability of the NGMP multi-core processor in the space domain. In: EMSOFT. (2012)