Cache-aware Interfaces for Compositional Real-Time Systems*

(Invited paper)

Linh Thi Xuan Phan
University of Pennsylvania
linhphan@cis.upenn.edu

Meng Xu
University of Pennsylvania
mengxu@cis.upenn.edu

Insup Lee
University of Pennsylvania
lee@cis.upenn.edu

ABSTRACT

Interface-based compositional analysis is by now a fairly established area of research in real-time systems. However, current research has not yet fully considered practical aspects, such as the effects of cache interferences on multicore platforms. This position paper discusses the analysis challenges and motivates the need for cache scheduling in this setting, and it highlights several research questions towards cache-aware interfaces for compositional systems on multicore platforms.

1. INTRODUCTION

Modern real-time systems are highly complex; for instance, a car nowadays contains more than 100 microprocessors that run thousands of software functions. As new functionalities are being added and new technologies – such as multicore processors – are being adopted, this complexity is only going to increase. To meet this trend, the real-time systems community has developed scalable timing analysis techniques based on *compositional* reasoning [8]. The idea is to break a complex system into individual components, which are scheduled hierarchically on the platform. These components are analyzed individually at first; then, an *interface* is generated for each component that captures the component's timing and resource requirements. By choosing the interfaces carefully, it is then possible to combine them into larger interfaces that cover bigger and bigger subsystems, so that it eventually becomes possible to derive the properties of the system as a whole.

Despite several promising results on compositional analysis, most existing research assumed an idealized platform in which all overheads are negligible. In practice, however, there are many sources of nontrivial overheads – such as cache effects, preemptions, context switches, and interrupts – that can substantially interfere with the execution of tasks. Ignoring these overheads when analyzing the system can lead to wrong results: at run time, the system can miss deadlines, even when the analysis may suggest otherwise [6].

To bridge this gap, recent work has begun to consider *overhead-aware* compositional analysis (see e.g., [3, 5, 6, 10]). The idea is to account for the overheads that tasks of a component experience and incorporate them into the component's resource demands, which are then used to derive the component's interface. This approach works well for uniprocessor platforms, and it has also been extended to multicore platforms with private caches [10]. However, the cache-aware compositional analysis on multicore platforms with shared caches remains an open problem.

In the following sections, we discuss the challenges in analyzing the overheads in the presence of shared caches and motivate the need for cache scheduling in this setting. We then outline several research directions towards cache-aware interfaces and their realization on modern multicore platforms.

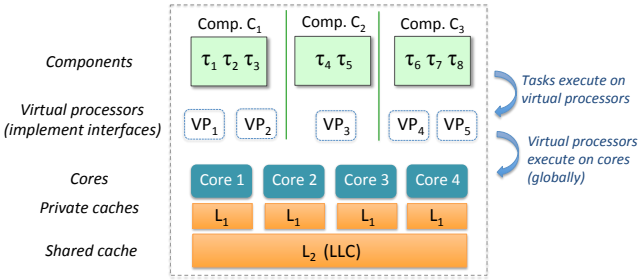

Figure 1: A compositional system on a multicore platform. Here, the resource demands of tasks in a component are abstracted as an interface. This interface is further transformed into a set of virtual processors, which supply resources to the tasks of the component. The virtual processors of all components are globally scheduled (e.g., as servers) on the cores. Each core has a private cache, and all cores share a shared cache.

2. ANALYSIS CHALLENGES

To illustrate the challenges, we use a two-level compositional system that is scheduled on a multicore platform with a last-level shared cache (LLC), as shown in Fig. 1.

Challenge #1: Concurrent cache accesses. On a multicore platform, tasks running simultaneously on different cores may concurrently access the memory regions that are mapped to the same cache sets of the LLC. When this happens, they may evict each other's content from the cache, thus resulting in cache misses. Precisely accounting for the overheads due to these concurrent cache accesses is highly challenging: in the worst case, these accesses may interleave with one another, and thus the tasks keep polluting each other's cache content. Without fine-grained information about the layouts of the tasks in the LLC and their access patterns, which are typically not captured in the tasks' specifications, it seems necessary to assume that *every* access to the LLC is a cache miss. With this assumption, one can perform the cache-aware analysis using the same method for private caches [10]. However, since this approach effectively considers the overhead of an LLC access to be the overhead of accessing the memory, it can result in unacceptable analysis pessimism.

Challenge #2: Cyclic dependency between components' interfaces and tasks' overheads. In a compositional setting, an additional source of cache overheads comes from the interactions between the components' tasks and their interfaces (implemented as

virtual processors). To illustrate this, consider the following scenario in Fig. 1. Suppose τ_1 (higher-priority) and τ_2 (lower-priority) are running on VP_1 and VP_2, which are mapped to cores 1 and 2, respectively. Suppose further that VP_1 is now preempted by another virtual processor of another component, say VP_4. Then, the higher priority task τ_1 will migrate to VP_2 and preempt the lower-priority task τ_2. This leads to both cache-related migration and preemption overheads: (i) since VP_2 is mapped to core 2, τ_1 has to reload its useful cache content to the private cache of core 2 when it executes on VP_2; and (ii) when τ_2 resumes later, it will need to reload the useful cache blocks that have been evicted from the private cache and the LLC by τ_1. In other words, when a virtual processor of a component is preempted by another virtual processor, tasks running within the component may experience cache overheads. Likewise, we can show that tasks can also experience overheads when a virtual processor of their component has exhausted its budget and stops its execution.

The key challenge in the analysis of the above type of overheads is the cyclic dependency between the interface computation and the overhead a task experiences. For instance, to derive the overhead-aware interface for C_1, we need to compute the overheads that its tasks experience. To compute such overheads, we need to know (among others) how often each of its virtual processors is preempted by virtual processors of other components and how often it exhausts its budget. The former is not possible in a truly open environment, where information about other components is completely hidden from the analysis of C_1. The latter requires certain information about the interface of C_1, which is to be computed.

In our prior work for private caches, we resolved this cyclic dependency by assuming that the period of a component's interface is given a priori and that it is available to all other components [10]. However, efficiently deriving optimal cache-aware interfaces in the general setting, especially in the presence of the LLC, remains an open research problem. One potential approach is to use *parametric interfaces*, but this would typically result in a much more complex analysis.

Challenge #3: Dynamic concurrent resource supply patterns. The analysis of cache overheads in a compositional setting is further complicated by the dynamic resource supply patterns of the interfaces. In the traditional non-hierarchical setting, the platform always provides fully available resources with a fixed degree of concurrency m (equal to the number of cores), and thus it is possible to determine the resource demand of a task in its busy window based on its worst-case execution time (WCET) and m. In a hierarchical setting, however, the virtual processors of a component are not always available to the component, and their supply patterns can in fact vary significantly depending on their parameters as well as how they are scheduled at the next level. These variations and dynamic changes in the supply patterns of the interface make the resource demand analysis highly challenging, especially when considering their impacts on the overheads that a task experiences. Careful considerations and new abstractions of the resource supply patterns are therefore necessary to derive resource-efficient cache-aware interfaces.

Challenge #4: Interactions between the LLC and private caches. For platforms with multi-level caches, the interactions between different cache levels can create intricate effects on the overheads. For example, a task can only pollute another task's cache content in the LLC if it has already experienced a cache miss in the private cache of its core, which in turn could have been caused by itself (i.e., intrinsic cache miss), by a higher-priority task, or by a virtual processor of a component. Similarly, different cache policies also lead to different overhead scenarios. For instance, with strictly inclusive caches, whenever a useful cache block of a preempted task is evicted from the LLC, the corresponding block in the private cache is also evicted; later, when the task resumes, it will need to reload the cache block to both the private cache and the LLC, thus experiencing overheads at both levels. On the contrary, for exclusive caches, a task may experience a cache miss in the private cache but not in the LLC. Due to these intricate effects, simple extensions of existing results for a single cache level may lead to overly pessimistic or incorrect results.

3. CACHE SCHEDULING FOR COMPOSITIONAL SYSTEMS

An essential characteristic – and important benefit – of compositional systems is the ability to provide *resource isolation* among components. To a large extent, this isolation has been achieved for CPU resources through careful CPU scheduling and interface analysis. However, on a multicore platform, components can still interfere with each other in a complex manner via caches. As was discussed in Challenge #1, tightly accounting for such overheads in the interface computation is extremely challenging, and even if new analysis methods could be established that reasonably account for the cache overheads, the components themselves would still not be "free from interferences." While this may not be an issue from the schedulability point of view, it can create undesirable consequences, such as potential security attacks via caches.

Recent advances in hardware- and software-based cache partitioning have brought a new solution within reach: instead of simply analyzing the cache as is, we can treat *cache as another schedulable dimension*. By breaking the cache into smaller pieces, e.g., using cache partitioning mechanisms, we can assign them to different tasks (or virtual processors, or components, or cores) at run time, and we can do so dynamically depending on how much cache space a task would need at a given point. This way, tasks running concurrently on different cores never access one another's cache space, thus completely eliminating the cache interference due to concurrent cache accesses. To realize this approach, the *CPU and cache schedulers would need to be aware of each other* to guarantee isolation while still being resource efficient.

In our prior work, we have explored this approach for real-time components [9], and our evaluation results show that it can help improve schedulability substantially compared to cache-agnostic scheduling. In the following, we discuss two important research problems and potential directions towards this approach for compositional systems.

Hierarchical cache partitioning: On today's hardware, one-level cache partitioning can easily be done using either software or hardware techniques, e.g., page coloring [2] or way-partitioning [4]. For example, using the page coloring mechanism, the cache can be divided into several disjoint partitions that each consist of a number of cache sets, which are mapped to different regions of the memory; by controlling the mapping of virtual addresses to machine addresses, the operating system can control which cache partition(s) a task can use. This approach can potentially be extended to enable hierarchical partitioning by adding intermediate layers of address translations. For instance, a two-level partitioning can be achieved by controlling the mapping from virtual addresses to the physical addresses and the mapping from physical addresses to machine addresses.[1] However, this has two direct implications: (i) the cache scheduler in each component would need to be aware of the mapping used by the next-level scheduler, i.e., complete isolation be-

[1]This approach may not work for systems with huge memory pages.

tween different scheduling levels is no longer achievable; and (ii) the cache allocation and reallocation become a lot more expensive.

In contrast, hardware techniques, such as way partitioning or Intel's Cache Allocation Technology [1], can provide very efficient cache allocation, but the total number of cache partitions is more limited than that of software-based techniques. It seems interesting to explore hybrid approaches that combine software and hardware mechanisms for different levels of the hierarchy, to achieve a larger number of partitions, while still maintaining low overhead and some degree of isolation between scheduling layers.

Static vs. dynamic allocation: A simple approach to achieving cache isolation in compositional systems is to perform static allocation across all levels of the hierarchy: the cache is statically partitioned to components, and each component's cache space is further partitioned to its sub-components (tasks). Since each task has its own cache space throughout its lifetime, it will never interfere with another task via cache, and the interface analysis can thus be done using existing techniques. However, this approach cannot always be feasibly applied in practice, e.g., when the tasks do not fit in the whole cache at the same time. When this occurs, the tasks have to receive fewer partitions (or even none) than they would require, which in turn result in much higher worst-case execution times (due to intrinsic cache misses, which they create on themselves). Static allocation can also severely under-utilize cache resources, because the cache partitions allocated to a task are wasted when the task is not executing or does not need all partitions.

An alternative approach is to only statically partition the shared cache among cores and apply the cache-aware compositional analysis for multicore systems with private caches [10]. However, this can still lead to high cache preemption and migration overhead when the virtual processors are scheduled globally on the cores.

The above issues can be solved by dynamically allocating cache resources to each task at run time. Dynamic cache allocation is beneficial, as it provides better flexibility and higher utilization of cache resources. However, it also presents several new challenges from both theory and systems perspectives. When preemption is allowed, tasks may still experience cache overhead – e.g., upon resuming from a preemption, a task may need to reload its cache content in the cache partitions that were used by its higher-priority tasks; as a result, the overhead analysis would need to consider the specific cache allocation strategy used at each layer of the hierarchy. Further, efficiently implementing dynamic cache reallocation is much more complex than static allocation, and software techniques such as page coloring typically have very high overheads in this case.

A promising direction is to integrate static and dynamic allocations using software techniques and hardware techniques, respectively. One potential solution is to perform static allocation at the component level and dynamic allocation at the task level. Not only can such an approach provide complete isolation among components (which is more critical than isolation among tasks of the same component), it can also be applied to systems with very high number of tasks. An interesting question for systems with more than two levels of hierarchy is to find the right combination of static and dynamic allocations that provides a good tradeoff between resource utilization and run-time efficiency.

4. CACHE-AWARE INTERFACES WITH CACHE SCHEDULING

In this section, we discuss the necessary extensions towards cache-aware analysis for compositional systems with cache scheduling.

Component specifications. Since a task's WCET depends on the cache space it is allocated, to enable optimized cache allocation, each task should specify not only a single WCET for a specific number of cache partitions but potentially a set of WCETs with respect to different numbers of allocated partitions. To derive tight analysis of the cache overheads a task may experience, it would be useful to additionally specify other fine-grained cache-related information such as the memory access patterns and the working set size, as they can greatly influence the overheads. Since this extended specification will also make the analysis much more complex, it is interesting to explore different trade-offs and quantifications of the cache-related information of a task to achieve accurate analysis without being intractable.

Component-level schedulers. To fully utilize resources, the CPU and cache scheduling should be integrated, since a task or a virtual processor can only be executed when it has *both* types of resources. It seems intuitive at first to synchronize the CPU and cache allocations, i.e., whenever a task (or virtual processor) receives CPU resources to execute, it will also receive the required cache resources. However, this is not always possible: because the amounts of cache and CPU resources that are available may be different, it is possible that there are sufficient cores to schedule the ready tasks but not sufficient cache partitions, and vice versa. In addition, unlike CPU scheduling, fully dynamic cache scheduling (at every scheduling level) is difficult to achieve without incurring high run-time overheads. It is therefore necessary to develop new scheduling solutions specifically for compositional systems that can efficiently and effectively integrate the CPU and cache allocations.

Cache-aware interfaces. To enable cache scheduling at the component level(s), it is necessary to expose not only the CPU requirements but also the cache requirements on the components' interfaces. One approach is to use similar concepts to CPU resource models such as the multiprocessor periodic resource model [7], which specifies the total resource budget Θ that must be provided in each period Π with a maximum degree of concurrency of m. One challenge with this approach is that, because the interface itself does not impose constraints on the exact level of concurrency, the number of cache partitions a component receives at run time can vary between 0 and m. This is problematic for the scheduling within the component, because not all allocated cache resources can be used effectively, e.g., if each task requires more than the provided number of partitions. Alternatively, if we allow a task to execute even if there are fewer partitions than it requires and the partitions it receive can vary dynamically, the worst-case execution time of the task in this scenario becomes difficult to predict. Therefore, it seems necessary to impose additional constraints on the concurrency level of the cache resources that an interface provides. In cases where static cache allocation is performed at the component level(s), the cache information exposed on an interface can potentially be reduced to the total number of cache partitions the component requires (as the number of cache partitions a component receives is fixed).

Interface analysis. With cache scheduling, the overhead due to concurrent cache accesses is eliminated, which significantly reduces the analysis complexity. However, since static allocation across all levels may not be possible, tasks and virtual processors may share and access the same partition(s) over disjoint periods of time. As a result, the analysis also faces all the remaining challenges described in Section 2. In addition, the interface analysis must also consider the cache supply patterns of the interface and its

impact on the cache overhead, e.g., caused by the (extra) preemptions via cache. We also note that the cache-aware interface computation (a synthesis problem) is significantly more challenging than the cache-aware schedulability analysis, as it requires computation of the cache-aware resource demand functions of tasks under a specific interface model, as well as a way to generate the interface parameters that satisfy the component's demands.

5. CONCLUSION

Compositional analysis is an effective approach to scheduling and analysis of complex real-time systems. However, its benefits have not been fully realized on modern multicore platforms due to the lack of a theory that can accurately consider the effects of caches. In this position paper, we have discussed key challenges in this setting, and we outlined several research questions and potential directions towards a realistic cache-aware compositional analysis theory for multicore platforms.

Acknowledgement

This work was supported in part by ONR N00014-13-1-0802 and N00014-16-1-2195, and NSF CNS 1117185, ECCS 1135630, and CNS 1329984.

References

[1] x86: intel cache allocation technology support. http://lwn.net/Articles/622893/. Accessed: 2015-01-09.

[2] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):338–359, Nov. 1992.

[3] W. Lunniss, S. Altmeyer, G. Lipari, and R. I. Davis. Accounting for cache related pre-emption delays in hierarchical scheduling. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems (RTNS)*, 2014.

[4] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[5] R. M. Pathan, P. Stenstrom, L.-G. Green, T. Hult, and P. Sandin. Overhead-aware temporal partitioning on multicore processors. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[6] L. T. X. Phan, M. Xu, J. Lee, I. Lee, and O. Sokolsky. Overhead-aware compositional analysis of real-time systems. In *In Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[7] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2008.

[8] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, 2003.

[9] M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *In Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[10] M. Xu, L. T. X. Phan, I. Lee, O. Sokolsky, S. Xi, C. Lu, and C. Gill. Cache-aware compositional analysis of real-time multicore virtualization platforms. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, 2013.