Extending Static Scheduling Algorithms for Moldable Tasks towards Dynamic Scheduling of Multiple Applications with Soft Real-time Properties

Extended Abstract

Jörg Keller

Parallelism and VLSI Group, FernUniversität in Hagen, Germany

Abstract: We investigate scheduling of multiple parallel applications with real-time constraints and the goal of energy minimization onto a multicore processor with frequency scaling. We propose a method to employ recent static scheduling algorithms for parallelizable tasks and adapt them in order to create a dynamic scheduler for the situation where applications may terminate, new applications may start, or applications change their behaviour, e.g. their degree of parallelism, at runtime. We list achievements so far and open problems.

1 Introduction

We consider real-time computing in the sense that a number of tasks (with different computational requirements) must be periodically invoked in order to meet real-time constraints in the form of minimum required throughput or maximum latency for reaction. Worst-case execution time (WCET) analysis of task code together with static scheduling of tasks has been employed for many years to ensure meeting the imposed real-time constraints. In recent years, several developments have widened the scope of real-time computing.

- Processor cores can be frequency scaled also in embedded systems, with the possibility to reduce power and energy consumption, but with the consequence that execution speed of tasks can vary, which complicates static analysis and scheduling.

- Even embedded microprocessors comprise multiple cores, in order to increase processing capability while maintaining moderate frequency and thus power consumption levels. This increases complexity of scheduling, even if interferences between tasks (like contention on shared memory) do not occur.

- Tasks have become more and more complex, so that tasks themselves might have to be parallelized in order to meet latency constraints. This again complicates scheduling, especially if the degree of parallelism is a parameter to be determined by the static scheduler. Also, the parallel efficiency of a task for different core counts must be analyzed statically in addition to the sequential WCET.

- The consolidation of embedded systems to reduce the number of processing elements puts multiple applications onto one multicore. Thus, if not all applications are always active, the situation occurs that the workload changes which necessitates a change of the (so far static) schedule. Similarly, an application which might be used in two different environments might change its behaviour while running, which also changes the workload, with similar effect.

We investigate the problem how far static scheduling with all its advantages can still be used when all the above extensions are taken into account. Some of the challenges have already been addressed. For example, there are several static scheduling algorithms that compute energy-optimal or close-to-optimal schedules for independent, parallelizable[1] tasks with a common deadline on multicore processors with frequency scaling [3–5]. Some of these algorithms assume discrete frequency levels and allow arbitrary power profiles of cores, so that their processor model is quite realistic. By assigning frequency levels per task, the algorithms overcome some of the limitations from a small number of available frequencies. Thus, as long as the different tasks of an application have a common periodicity[2] those scheduling algorithms can be used. While those algorithms require the parallel efficiency functions of the parallel tasks as an input, there are approaches to use profiling on codes in order to compute a table of parallel efficiencies for different core counts [2]. Multiple applications, where

[1] The tasks are assumed to be *moldable*, i.e. they do not change their degree of parallelism during runtime.

[2] If not, it might be possible to achieve that situation by finding a small least common multiple of the individual deadlines and duplicating tasks accordingly.

all applications are present all the time, could be treated by considering all of their tasks together.

Despite all these achievements, the last item in the list above requires dynamic adaption. If an application terminates, the slots for its tasks remain empty, and the resulting idle times could be used to improve energy efficiency by slowing down other tasks. If an application starts while others are already running, the additional tasks from the new application must be mapped to cores, and some of the frequencies must be increased in order to handle the increased workload. Finally, an application that modifies its behaviour can be considered a combination of the previous two cases: the old behaviour terminates, the new behaviour starts. At least, a partial problem has been addressed in [1]: For a single parallel application, compiler analysis is available to determine different phases of the application, where a phase represents a period of fixed performance requirements and parallel efficiency. The application is instrumented so that it informs the operating system scheduler when it enters a new phase. The information comprises the future performance requirement and degree of parallelism, so that the scheduler can adapt frequencies and core allocations.

2 Proposed Approach

To tackle the problem for multiple applications, we propose to extend the current procedure of static scheduling and subsequent execution as follows.

Initially, a set of applications with known task behaviours is present. Thus, a static scheduling algorithm can be used to compute a schedule that allows to repeatedly invoke all the tasks with chosen frequencies such that all deadlines are respected and energy (per scheduling round) is minimized.

If an application terminates, we leave the schedule as it is, i.e. we create idle phases where the tasks of the terminated application have been mapped. This schedule still respects all deadlines, but is not necessarily energy-optimal anymore. We call this an *ad-hoc adapted* schedule.

If another application (with known task characteristics) starts at some time, then it is mapped to as many cores as possible, and the frequencies of these cores are increased to a level that ensures that the deadlines are respected. Such a level must exist if we assume that the system is not overloaded. Also this schedule is ad-hoc adapted.

If some application enters a new phase, it informs the runtime system about its new behaviour and is treated like a combination of a terminated and a new application, with the difference that the idle phases created by "terminating" the previous phase are preferably used to map the tasks for the new phase.

As the ad-hoc adapted schedules still respect all deadlines, the real-time properties of the system are maintained, only the energy per round may not be minimal anymore. Therefore, if the system has switched to an ad-hoc adapted schedule, it may invest some resources (in addition to invoking the application tasks) to compute a new energy-optimal schedule. As soon as this schedule is ready, it replaces the ad-hoc adapted schedule, and the system is energy-optimal again.

3 Open Problems

So far, the proposed approach has not been implemented, although implementations of some parts are available (cf. the literature in Sect. 1). Thus, the parameters for such a system to run smoothly must be determined. For example, so far it is not exactly known how much energy must be spent to compute a new optimized schedule. While there are indications that it is much cheaper to compute a schedule that only differs little from a previously computed schedule (in contrast to computing a schedule from scratch), this has to be tested. Also, the energy to compute a schedule has to be put in relation to the energy savings achievable when going from an ad-hoc adapted schedule to an optimal schedule again. Hence, if the ad-hoc adapted schedule is close to optimum, one could wait until several changes have assembled before computing a new energy-optimal schedule. Consequently, the frequency of changes in workload would be needed for an assessment. This could be achieved by benchmarking real-time applications with these characteristics, as far as they are available. Yet it could also turn out a chicken-and-egg problem, where systems with multiple real-time applications of changing behaviour only show up when appropriate scheduling algorithms are available. Finally, the accuracy of the scheduling forecast must be tested by implementing a prototype on a real machine. Ultimately, a production system would need operating system support for applying the above proposal, or even integration of such a proposal into the scheduler of a real-time operating system.

Acknowledgements

I would like to thank Christoph Kessler for several helpful discussions about this topic.

References

[1] Simon Holmbacka, Erwan Nogues, Maxime Pelcat, Sebastien Lafond and Johan Lilius. Energy Efficiency and Performance Management of Parallel Dataflow Applications. In: *Proc. Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2014, pp. 1–8.

[2] Christoph W. Kessler, Welf Löwe. Optimized Composition of Performance-aware Parallel Components. Concurrency and Computation: Practice and Experience, Volume 24 Issue 5, 2012, pp. 481–498.

[3] Nicolas Melot, Christoph Kessler, Jörg Keller, Patrick Eitschberger. Fast Crown Scheduling Heuristics for Energy-Efficient Mapping and Scaling of Moldable Streaming Tasks on Manycore Systems. ACM Transactions on Architecture and Code Optimization (TACO), Volume 11 Issue 4, January 2015, Article No. 62.

[4] Peter Sanders and Jochen Speck. Energy efficient frequency scaling and scheduling for malleable tasks. In: *Proc. 18th International Conference Euro-Par*, 2012, pp. 167–178.

[5] H. Xu, F. Kong, and Q. Deng. Energy minimizing for parallel real-time tasks based on levelpacking. In: *Proc. 18th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA12)*, 2012, pp. 98–103.