Work-In-Progress: Migration Hints in Real-Time Operating Systems

Phillip Raffeck, Peter Ulbrich, Wolfgang Schröder-Preikschat
Department of Computer Science, Distributed Systems and Operating Systems
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

Abstract—**Task migration is a potent instrument to exploit multi-core processors. Like full preemption, full migration is particularly advantageous as it allows the scheduler to relocate tasks at arbitrary times between cores. However, in hard real-time systems, migration is accompanied by a tremendous drawback: poor predictability and thus inevitable overapproximations in the worst-case execution-time analysis. This is due to the non-constant size of the tasks' resident set and the costs associated with its transfer between cores. As a result, migration is banned in many real-time systems, regressing the developer to a static allocation of tasks to cores with disadvantageous effects on the overall utilization and schedulability.**

In previous work, we successfully alleviated the shortcomings of full migration in real-time systems by reducing the associated costs and increasing its predictability. By employing static analysis, we were able to identify beneficial migration points and thus generate static schedules migrating tasks at these identified points. In ongoing work, we extend this approach to dynamic scheduling by providing information about advantageous migration points to an operating system which then makes migration decisions at runtime.

I. Introduction and Related-Work

Fully exploiting the potential of multi-core platforms in real-time systems with hard deadlines still is a challenging task. An important part of this problem is the effective utilization of cores. For example, a task set τ_a, τ_b, and τ_c with adverse processor utilization characteristics of 70, 70 and 40 percent (i.e., 180 % utilization) is infeasible to schedule on a system with two cores (i.e., 200 % capacity) when statically allocated.

This issue is addressed by dynamic allocation and migration of workload, respectively. Such migration can be realized at the task, job, and instruction level [1]. The former two variants are relatively easy to implement as the migration occurs only at scheduling-unit boundaries. However, they still fail to find a feasible schedule for the previous example, as the potentially adverse size of scheduling units remains. Migration on instruction level or *full migration*, on the other hand, facilitates the relocation of work at virtually any time. It thus allows splitting up jobs and distributing their utilization across cores. An abundance of multi-core scheduling algorithms [1], [2] rely on such fine-grained migration to exploit the potential of multi-core systems. Here, a widespread assumption is that migration works much like preemption and that overheads are practically constant. However, migration comes at considerable costs in practice, as the operating system not only has to preempt a task but also transfer its resident set (i.e., active working set) between core-local memories. In contrast to core-local preemption, these costs are non-constant and highly dependent on the point of migration [3], [4]. Their overheads may even jeopardize deadline tardiness and feasibility [5]. To nevertheless comply with the assumption of constant migration costs, worst-case execution time (WCET) analysis is forced to assume a pessimistic bound on the resident-set size and the resulting migration costs at any time of a task's execution.

One possible solution to the problem is the choice of an advantageous hardware platform. Especially for unified memory architectures (UMAs) with multi-level cache hierarchies, Bastoni et al. [6] have shown that, due to the consistent memory latencies, preemption and migration costs are comparable and dominated by cache-related costs. In particular, in UMA systems, the resident-set size is largely negligible for the estimation of migration costs. However, these considerations do not apply to non-unified memory architectures (NUMA) [4] and NUMA-like systems [3] that lack cache coherency or use other forms of inter-core communication. For the Infineon AURIX™ platform (TC277), for example, we measured substantial migration overheads of up to 2.5 times for inter-core reads. Moreover, as for the flat memory hierarchy, the transfer cost scale linearly with resident-set size.

A way to increase predictability and reduce costs is to restrict task displacement in the first place. For preemption, thresholds for the apt placement of preemption points were studied [7], [8]. Anderson et al. [9] extended this concept to restricted migration, but lack a implementation. Automatic analysis and generation of multi-core systems [10]–[12] for non-preemptive scheduling has been studied. Sarkar et al. [5] proposed hardware-assisted migration, which can hide latencies but does not solve the fundamental issue of WCET overapproximation. Jahn et al. [13] proposed an optimized transfer of memory pages to reduce latencies. None of these approaches facilitates full migration and tight WCET bounds.

A. Problem Statement

Assuming that migration costs correlate with the resident-set size on a given hardware platform, the design of hard real-time systems faces a dilemma: Migration should be avoided or limited to task/job-level migration to maintain temporal analyzability of tasks. However, the granularity of scheduling units is a crucial factor for schedulability as well as potential utilization. In contrast, full migration facilitates high utilization but comes with additional migration overheads. The fundamental issue is in the variability of indirect costs that depend on the resident-set size, which implies pessimism in the WCET

2576-3172/19/$31.00 ©2019 IEEE
DOI 10.1109/RTSS46320.2019.00056

bounds. We believe that there are two things necessary to solve the dilemma: (a) avoiding migration wherever possible and (b) increasing its predictability.

The first step in this direction is to cut scheduling units to a beneficial size by static code analysis. However, cutting code to match a certain size is a tedious process as the execution time is a non-functional property and thus hard to correlate with the source code. The fundamental challenge is to identify *split points* that are valid for all possible execution paths across branches and preserve the task's functional properties. Furthermore, choosing scheduling units only by size still suffers from the same issues as full migration, namely potentially high migration costs. A split point that results in optimally sized scheduling units may coincide with an unfavorably large resident set. In the worst case, the additional migration costs may again jeopardize the schedulability gained by changing the granularity in the first place. Here, the challenge is in the identification of split points that benefit both aspects.

Advantageous scheduling units with minimal migration costs facilitate high utilization and guarantee deadline adherence in the worst case. However, with jobs typically not exploiting their WCET migration may be omitted in the average case. Here, the challenge is operating-system support to leverage the knowledge inferred by static analysis and to restrict migration to the beneficial split points at runtime.

B. Our Approach and Contribution

In this paper, we present MIGROS (anticipatory MIGration support in Real-time Operating Systems), our ongoing work to address the aforementioned issues. Its first pillar is the static code analysis of the system, which we partially described in our previous work [14]. At compile time, our analysis framework identifies potential split points with the desired granularity by a heuristic estimation of execution cost while ensuring that program semantics are preserved across all control-flow branches. Subsequently, the granularity of the resulting scheduling units is verified by a target-specific WCET analysis. We optimize both the scheduling-unit size and the associated migration cost simultaneously by extending the search for split point candidates to the vicinity of the optimal scheduling-unit granularity. This way, we can choose split points with beneficial resident-set sizes. Finally, our toolchain performs the actual cutting of code with the resulting units facilitating optimized static scheduling.

However, in a static schedule, any migration is always performed, even if it is unnecessary under the current execution conditions. Therefore, we propose OS support to leverage our split-points at runtime by what we call *migration hints*. As part of our ongoing work on MIGROS, we present three variants for such a hinted migration mechanism: *job-level migration*, *migration as exception*, and *on-demand migration*. Thereby, we strive to mitigate migration overheads further and improve the overall system performance for the average case.

II. System Model and Assumptions

We consider a hard real-time system with m cores that allows full preemption and full migration. We define the latter to permit migration at each instruction of the application code but to prohibit migration during the execution of system calls and other operating-system code. A set τ of n sporadic tasks with occurrence rate T_i is scheduled on m processors. Each task τ_i contains a set of l *scheduling units* J (a.k.a. jobs) and has a processor utilization U_i. The number of tasks n, the period and their respective *worst-case execution time* (WCET) C_i determine the theoretical schedulability. A system is theoretically schedulable on m cores if the total utilization of all tasks is less or equal than the number of cores m.

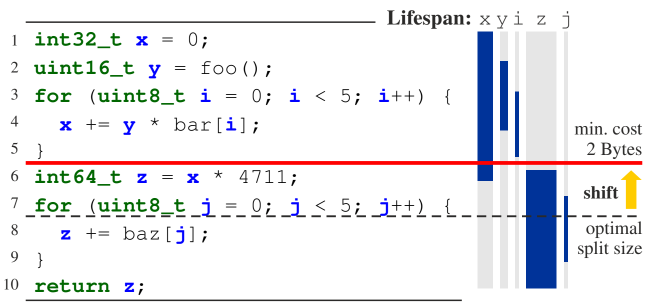
As a non-functional requirement for the static analysis, we assume that upper bounds on the number of iterations for all loops are given. Additionally, cache-related overhead is already covered in the timing analysis by the overapproximations required to incorporate preemption effects and delays. We further assume that the target hardware features a RISC processor with in-order execution and explicitly managed, non-coherent caching mechanisms, for instance, scratchpads. Generally, we assume that on the considered hardware platform, migration costs increase with the resident-set size.

With the notion of a *resident set*, we refer to the currently active (i.e., alive and resident in memory) part of a process's working set, for example, local variables or the state of the stack. For reasons of generality, we consider the set elements as bytes. This fine-grained view can ultimately be mapped to any type of higher-order memory management (e.g., lines, pages). We restrict this definition to comprise only core-local data and assume that all other data is globally accessible.

III. Static Analysis and Tailoring

In previous work [14], we presented an entirely static approach for the identification of beneficial migration points using the proposed concept of a *split-point graph*. This work constitutes the starting point for our dynamic migration approach presented in this paper. Therefore, we quickly summarize our previous work in the following. As a first intermediate step, we leverage static source-code analysis and knowledge about the semantics of the targeted operating system to derive all truly existing scheduling units and their respective control-flow graphs. Subsequently, we infer additional information to assess the suitability of potential split points, which is the size of the active resident set as well as a heuristic execution-time estimation. Finally, we embed this information in a split-point graph, where edges correspond to the possible split points and nodes represent all instructions between them.

We then use the split-point graph to identify program points with minimal migration cost, where splitting yields scheduling units of suitable size. To achieve this, we assess each possible split point according to two criteria: distance (δ) to the intrinsic split point and the associated resident-set size (ω), that is, migration costs. *Intrinsic split point* hereby refers to the program point where a migration-cost–agnostic algorithm only considering the execution time would split the scheduling unit. We can instrument our analysis to consider the scheduling-unit size in two ways: It either searches for precisely one split point



```
 1 int32_t x = 0;
 2 uint16_t y = foo();
 3 for (uint8_t i = 0; i < 5; i++) {
 4   x += y * bar[i];
 5 }
 6 int64_t z = x * 4711;
 7 for (uint8_t j = 0; j < 5; j++) {
 8   z += baz[j];
 9 }
10 return z;
```

Figure 1: Working set and lifespan of automatic variables, showcasing the need to simultaneously consider both scheduling-unit size and resident-set size to obtain optimal split points with minimal overhead.

making a task set schedulable or aims to identify multiple, about equidistant split points.

Figure 1 illustrates the interplay of the two optimization parameters δ and ω. In the example, splitting between lines seven and eight would lead to the optimal target size of the resulting scheduling units. However, in this case, we suffer from significant migration costs, as `j` and `z` have to be transferred. By employing the size of the resident set at a given program point to additionally consider the migration cost, our proposed algorithm identifies the split point between lines five and six. This split point has a minimal distance from the intrinsic split point, while simultaneously just the intermediate result stored in `x` has to be transferred.

By employing the proposed approach, we are able to improve the schedulability of statically scheduled systems automatically. This results in up to 70 % more schedulable task sets and up to 76 % reduced migration cost overhead.

As we consider static schedules, each migration happens unconditionally at a fixed migration point, maybe needlessly. We thus propose to reuse the analysis part of our static approach to identify suitable migration points, but postpone migration decisions to runtime. Bringing our approach to dynamic systems allows us to overcome such inflexibilities, reducing the overhead even further.

IV. OS Support and Dynamic Scheduling

By splitting tasks and allocating the resulting scheduling units to different processor cores, we embed the information about advantageous migration points as fixed migrations in the generated schedule. As the scheduling algorithm has to work with the WCET of each scheduling unit, a migration deemed necessary in the worst case might actually be skipped in most cases. Consider, for instance, the scenario depicted in Figure 2. We have a set of three tasks with a deadline of 10 ms and a WCET of 6 ms (see Figure 2a). With the use of migration, this task set is schedulable on a two-processor system. For this migration, our analysis identified a beneficial migration point (`MP`) in τ_1 after 3 ms. Our static approach, therefore, generates the static schedule depicted in Figure 2b. However, as τ_2 has an average-case execution time (ACET) of 2 ms, an online scheduling algorithm can, in the average case, scheduled the jobs as depicted in Figure 2c, completely avoiding the costly migration.

As in the static schedule the migration is rigidly embedded, there is no possibility to skip the migration in cases where tasks execute for less than their WCET. We aim to overcome this inflexibility by exploiting knowledge about advantageous migration points at runtime. For this, we perform the same analysis as in our previous approach but perform no splitting of scheduling units. We rather pass information about advantageous migration points on to the operating system as *migration hints*. In the following, we propose three concepts how operating systems can make use of such migration hints.

A. Job-level Migration

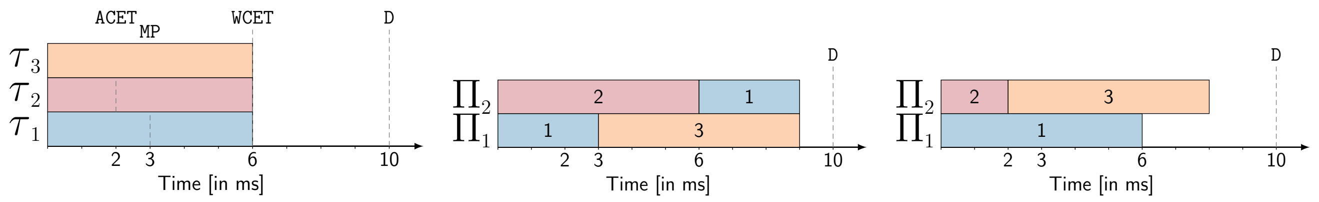
The obvious approach to dynamically exploit advantageous migration points is to keep the splitting mechanism of the static approach and create new jobs at exactly these boundaries where migration is cheap. But instead of generating a static schedule for this job set and pinning each job to a dedicated processor core, we defer allocation and scheduling to an online algorithm capable of job-level migration. As in the static case, we benefit from low migration cost. Additionally, we are able to save the migration-cost overhead in cases, where jobs execute for less than their WCET and thus increase the slack time available on a processor core. The online scheduling algorithm may then be able to schedule the jobs of a previously split task on the same core, thus skipping the migration. In contrast to the rigid schedule of the static approach, this splitting into jobs without pinning them to a core but rather performing the allocation on demand at runtime is able to induce migration overhead only when necessary.

As previous work [15] has shown, partitioned EDF scheduling benefits from splitting tasks into two jobs and assigning the first part a WCET-constrained deadline. That means, the first resulting job is assigned a deadline equal to its execution time. Our analysis can be instrumented to generate such job sets with an additional focus on minimal migration cost.

B. Migration as Exception

As an even further extension, we can try to leverage the ACET of a task. Placing an advantageous migration point after the ACET enables the operating system to check the need for migration in a straightforward fashion. If the job has executed for its ACET when reaching the migration point, it is on its worst-case path (or a similar one), meaning it will need more execution time and has to be migrated. If, on the other hand, the job has not yet depleted its ACET budget, it will finish execution in time, obviating the need for migration. This approach facilitates migration decisions and helps to avoid the migration most of the time, requiring it only in the exceptional case, where a job executes for longer than its ACET.

As it might be impractical to statically identify one migration point definitely after the ACET, or even the ACET itself, we can offload this task to the runtime system. For this, we instrument the static analysis to create migration points scattered across the whole task so that there are few migration points at the beginning and increasingly more towards the end



(a) Taskset with suitable migration point (b) Static Schedule (c) Dynamic Schedule

Figure 2: Scenario showcasing how the flexibility of a dynamic approach allows to save migration costs.

of a task. This ensures that scheduling units are more fine-grained after the ACET and, thus, migration decisions happen in shorter intervals. Combining this with runtime monitoring of execution times, to enable measuring the ACET, enables an operating system to make migration decisions based on the average-case budget of a job, as described in the previous paragraph. In this case, however, there is no requirement of computing the ACET at compile time.

C. *On-demand Migration*

Lastly, we can use the size criterion to insert multiple beneficial migration points which are approximately equidistant. Instead of splitting tasks, we can inject traps into the program code. By combining this mechanism with a slack-aware scheduling algorithm, the operating system can at runtime at each migration point decide if the currently running task has to be migrated or if it can be dispatched again on the current processor until at least the next potential migration point. This concept is similar to partial preemption, in the sense that there are predefined points at which a task might be migrated.

D. *Thoughts on Efficiency*

The switch from offline to online scheduling naturally comes with overhead induced by making scheduling decisions at runtime. To keep this additional overhead small, we can, in the case of on-demand migration, instead of traps, we can inject special-tailored constructs for each migration point, triggering the migration of the resident set at this program point. Making these constructs toggleable by the operating system obviates the needs to activate the operating system at each migration point. Instead, the operating system can asynchronously set the flag accordingly, deactivating mirgration points known to be unnecessary in the current execution.

Additionally, we can instrument tasks to migrate themselves proactively at an advantageous migration point. This way, neither a scheduling decision nor other interaction with the operating system is necessary, greatly reducing the overhead.

V. Conclusion

In this paper, we presented our ongoing work on MIGROS, an approach to boost migration in hard real-time systems. An essential component of MIGROS is the automated analysis of tasks at the source-code level, which we presented in parts in [14]. Our analysis yields beneficial split points with minimal migration costs resulting in tighter WCET bounds. On the one hand, subdividing tasks into smaller scheduling units at compile-time improves the schedulability of static allocation schemes. On the other hand, knowledge about split points can be used by the RTOS as *migration hints* that serve as an enabler for migration thresholds; analogous to limited preemptive scheduling [8]. We further presented our ongoing efforts on runtime support that leverages the migration hints acquired by static analysis: *job-level migration*, *migration as exception*, and *on-demand migration*.

Acknowledgment

This work is supported by the German Research Foundation (DFG) under grants no. SCHR 603/13-1, SCHR 603/9-2, the CRC/TRR 89 Project C1.

References

[1] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. H. Anderson, and S. K. Baruah, "A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms," in *Handbook of Scheduling*, 2004.

[2] R. I. Davis and A. Burns, "A Survey of Hard Real-time Scheduling for Multiprocessor Systems," *ACM Computing Surveys*, vol. 43, no. 4, p. 35, 2011.

[3] E. W. Briao, D. Barcelos, F. Wronski, and F. R. Wagner, "Impact of Task Migration in Noc-based Mpsocs for Soft Real-time Applications," in *Int'l Conf. on Very Large Scale Integration*, Oct. 2007, pp. 296–299.

[4] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "LITMUS^RT : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers," in *Proc. of RTSS '06*, Dec. 2006, pp. 111–126.

[5] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan, "Push-assisted Migration of Real-time Tasks in Multi-core Processors," *Sigplan Notes*, vol. 44, no. 7, pp. 80–89, 2009.

[6] A. Bastoni, B. Brandenburg, and J. Anderson, "Cache-related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability," in *Proc. of OSPERT '10*, 2010, pp. 17–22.

[7] B. Peng, N. Fisher, and M. Bertogna, "Explicit Preemption Placement for Real-Time Conditional Code," in *Proc. of ECRTS '14*, Jul. 2014, pp. 177–188.

[8] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited Preemptive Scheduling for Real-time Systems. A Survey," *IEEE Trans. on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2013.

[9] J. H. Anderson, V. Bud, and U. C. Devi, "An EDF-based Restricted-migration Scheduling Algorithm for Multiprocessor Soft Real-time Systems," *Real-time Systems*, vol. 38, no. 2, pp. 85–131, Feb. 2008.

[10] T. Klaus, F. Franzmann, M. Becker, and P. Ulbrich, "Data Propagation Delay Constraints in Multi-rate Systems: Deadlines Vs. Job-level Dependencies," in *Proc. of RTNS '18*, ser. RTNS '18. New York, NY, USA: ACM, 2018, pp. 93–103.

[11] F. P. Franzmann, T. Klaus, P. Ulbrich, P. Deinhardt, B. Steffes, F. Scheler, and W. Schröder-Preikschat, "From Intent to Effect: Tool-based Generation of Time-Triggered Real-Time Systems on Multi-Core Processors," in *Proc. of ISORC '16*, 2016.

[12] F. Nemati, M. Behnam, and T. Nolte, "Efficiently migrating real-time systems to multi-cores," in *Proc. of ETFA -09*. IEEE, 2009, pp. 1–8.

[13] J. Jahn, M. A. A. Faruque, and J. Henkel, "Carat: Context-aware runtime-adaptive task migration for multi-core architectures," in *DATE*, March 2011, pp. 1–6.

[14] T. Klaus, P. Ulbrich, P. Raffeck, B. Frank, L. Wernet, M. R. von Onciul, and W. Schröder-Preikschat, "Boosting Job-Level Migration by Static Analysis (Best Paper)," in *Proc. of OSPERT '19*, 2019, pp. 33–44.

[15] A. Burns, R. I. Davis, P. Wang, and F. Zhang, "Partitioned EDF Scheduling for Multiprocessors Using a C=D Task Splitting Scheme," *Real-Time Systems*, vol. 48, no. 1, pp. 3–33, Jan. 2012.