# Boosting Job-Level Migration by Static Analysis

Tobias Klaus, Peter Ulbrich, Phillip Raffeck, Benjamin Frank,
Lisa Wernet, Maxim Ritter von Onciul, Wolfgang Schröder-Preikschat
Department of Computer Science, Distributed Systems and Operating Systems
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

***Abstract*—From an operating system's perspective, 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 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 this paper, we tackle the shortcomings of full migration in real-time systems by reducing the associated costs and increasing its predictability. Our approach is to analyze a task's source code to identify beneficial migration points considering the size of scheduling units and the associated migration costs. Consequently, we can do both: generate schedules that benefit from static migration as well as provide information about advantageous migration points to dynamic scheduling, making full migration more predictable. Our experiments show improved schedulability and a reduction in transfer size of up to 76 percent.**

## I. Introduction

To date, real-time scheduling on multi-core systems while fully utilizing the cores is a challenging task. Initially, static allocation of tasks to cores suffers from the well-known Dhall's effect [1]: adverse utilization characteristics of tasks may lead to poor overall utilization up to the point where the system becomes unschedulable. Consider for example, a task set comprising three tasks $\tau_a$, $\tau_b$, and $\tau_c$, where $\tau_a$ and $\tau_b$ have a processor utilization of $70\,$percent and $\tau_c$ of $40\,$percent. Allocating this task set on a system with two cores is infeasible. This problem of static allocation can be overcome by dynamic allocation and the possibility to migrate work between different cores, as it enables to spread work among cores. Such migration is conceptually possible at multiple levels of granularity: task, job, and instruction level [2]. While the former two are relatively easy to implement, they are incapable of solving the general issue. They still fail to find a feasible schedule for the previous example, as the infeasibility stems from the adversly large utilization on task and job level. Migration on instruction level, on the other hand, succeeds to utilize the system fully and thus to find a feasible schedule, as it allows split up a job and distribute its utilization across cores. An abundance of multi-core scheduling algorithms [3] rely on such fine-grained migration to exploit this potential.

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 different core-local memories. In contrast to core-local preemption, these costs are non-constant and highly dependent on the point of migration [4]. Thus, migration at the instruction level carries the risk to migrate at adverse points that are associated with high overheads. Choosing an inapt migration point may even jeopardize deadline tardiness and feasibility [5]. Even worse, worst-case execution time (WCET) analysis is forced to assume a pessimistic bound on the migration cost. To conserve predictability, migration is thus banned in many real-time operating systems, restricting the developer to a static allocation of tasks to cores with disadvantageous effects on the overall utilization and schedulability.

Without resorting to migration, however, the granularity of scheduling units is a crucial factor for the overall schedulability of a given system as it is typically easier to find a valid schedule for smaller scheduling units. For runtime migration, only the direct cost of the migration mechanism itself can be influenced by the operating systems. However, the fundamental issue is in the variability of the indirect cost, that is resident-set size. Consequently, we identified the following two major challenges to overcome the predictability issues and to boost migration in real-time systems.

### A. Challenge #1: Adverse Size of Scheduling Units

Considering our previous example, finding a schedule is infeasible only because of the adverse granularity of the three tasks although, in theory, the overall utilization is not exceeded. Splitting tasks into smaller scheduling units solves the problem, for example, by splitting task $\tau_c$ into two parts with a utilizaton of less than $30\,$percent each. However, cutting code to match a certain scheduling granularity 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.

*Our Approach:* We perform static analysis on the system at compile time to identify potential split points with the desired granularity based on execution cost estimations. Additionally, we ensure that program semantics are preserved across all control-flow branches. We leverage heuristic estimation of execution cost to keep the analysis overhead manageable. Subsequently, a target-specific WCET analysis can verify the granularity of the scheduling units.

### B. Challenge #2: Minimize Migration Overhead

Choosing scheduling units only by size still suffers from the same issues as instruction-level migration, namely potentially high migration cost. 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 cost may again jeopardize the schedulability gained by changing the granularity in the first place. The challenge is in the identification of split points that benefit both aspects.

*Our Approach:* 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 are able to choose split points with beneficial resident-set sizes.

### C. Contribution and Paper Structure

In this paper, we present an approach to facilitate the use of migration in real-time systems by reducing the associated costs and increasing its predictability. Our toolchain allows for automated static analysis of existing source code to identify beneficial migration points by simultaneously considering both the size of scheduling units and the associated migration costs. We transform the control-flow graph into a split-point graph that models scheduling units and holds information on potential split points and their costs. Consequently, our analysis can be used to both generate static schedules with migration as well as provide hints on beneficial migration points to dynamic scheduling thus supporting online migration.

The remainder of this paper is structured as follows: In Section II, we present our approach to solve the aforementioned challenges. Section III gives an overview of the implementation of our prototype, which is evaluated in Section IV. Section V outlines related work in the context of job migration and Section VI concludes.

## II. Approach

In this section, we detail our approach to the identification of beneficial migration points by static code analysis. We, therefore, first introduce the system model and fundamental assumptions that we demand and give an overview of the general concept of our analysis. Finally, we detail the handling of loops and branches, which are in particular challenging and require specific cutting schemes.

### A. System Model and Assumptions

We consider a 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 $\tau$ of $n$ sporadic tasks with occurrence rate $T_i$ is scheduled on $m$ processors. Each task $\tau_i$ contains a set of $l$ *scheduling units* $J$ (a.k.a. jobs) and has a processor utilisation $U_i$. 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 $m$, i.e., if the following equation holds:

$$\sum_{i=1}^{n} U_i = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq m \text{ with } C_i = \sum_{k=1}^{l} C_i^{J_k} \tag{1}$$

Here, $C_i^{J_k}$ denotes the WCET of the scheduling unit *k* of $\tau_i$. We extend the inequality by the overhead $\alpha$ to express the (data transfer) costs associated with migration for each task:

$$\sum_{i=1}^{n} \frac{C_i + \alpha_i}{T_i} \leq m \tag{2}$$

As a non-functional requirement, we assume that upper bounds on the number of iterations for all loops are given. We further assume a RISC processor without out-of-order execution as the target and that cache-related overhead is already covered by the overapproximations required to incorporate preemption effects and delays.

With the notion of a *resident set*, we refer to the currently active (i.e., alive) part of a process's working set; the latter is often used interchangeably in the literature. That is, for example, local and global variables or the state of the stack. We restrict this definition to comprise only core-local data and assume that all other data is globally accessible.

### B. General Concept

We leverage static code analysis to identify interactions with the operating system's scheduler, that is system calls, from the tasks' source code. By incorporating knowledge about the semantics of the targeted operating system and its scheduling, we can thereof derive all truly existing scheduling units and their respective control-flow graphs; irrespective of the development model (e.g., process or run-to-completion) and style the developer pursued.

We further infer all additional information that is required to identify potential split points. First and foremost, this is the active resident set at all times. We, therefore, perform a liveness analysis of all local variables and compute the resident-set size for every instruction. Furthermore, we perform a heuristic timing analysis of all nodes to estimate their size in terms of execution time. Finally, the control-flow graphs are transformed into *split-point graphs* that hold all additional information. In this graph, edges correspond to the possible split points, and nodes represent all instructions between them.

The identification of split points consecutively transforms the split-point graph such that all nodes are at or below a predefined target size at minimal migration costs. To achieve this, we assess each possible split point according to two criteria: distance ($\delta$) to the intrinsic split point and the associated resident-set size ($\omega$), that is migration costs. Figure 1 illustrates the interplay of these two parameters. Recall that we assume global variables to be globally accessible from all cores.

We define the intrinsic split-point as the point, where the estimated worst-case execution time since the beginning of the scheduling unit, or, equivalently, the last split point, approximately equals the target scheduling-unit size. In our

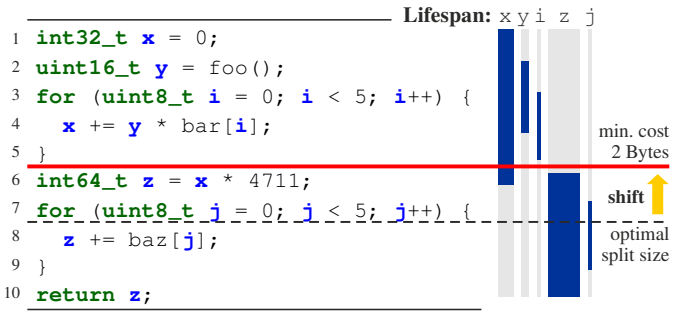


Figure 1: Working set and lifespan of automatic variables, showcasing the need to simultaneously consider both scheduling-unit size and resident-set size to obtain optimal split points with minimal overhead.

example, splitting between lines seven and eight would lead to optimal size. However, in this case, we suffer from significant migration costs, as `j` and `z` have to be transferred.

We, therefore, employ the size of the resident set at a given program point to further consider the migration cost. We acquire $\omega$ by a liveness analysis to identify all local variables that are referencable from a given program point. In the example in Figure 1, the state is minimal between lines five and six, where just the intermediate result stored in `x` has to be transferred.

By simultaneously optimizing both criteria, we are, in general, able to obtain best-suited points for cutting the scheduling unit locally. However, we have to assess possible split points also from a global point of view. Only by that, we can guarantee that the resulting cut is both correct and suitable in cases where different possibilities of program flow exist, for example in branches and loops. Therefore, we search for a minimal cut on the split-point graph to find split points in all branches that result in global split points associated with minimal migration costs.

In the following sections, we give further insights on how our approach handles specific program constructs.

### C. Splitting Branches

Assessing a sequential control flow according to our criteria only requires a straightforward assessment of all split-point candidates and selection of the optimal one. In contrast, branches are harder to split as we need to maintain global relations between split points in all branches belonging to the same conditional construct to preserve program semantics. A further challenge is to avoid an increase in the overall WCET by an unbalanced subdivision of branches. Figure 2 illustrates the underlying problem: For the original, uncut branch (left), we have a WCET of $C_{uncut} = 205$. In this example, the liveness analysis reveals minimal migration costs at the beginning of the `true` and the end of the `false` branch. Consequently, the cut scheduling units $SU_A$ and $SU_B$ contain the greater part of the `true` and `false` branches respectively. Ultimately, the WCET analysis suffers from a pessimistic overapproximation in both branches yielding an overall WCET of $C_{cut} = 350$.

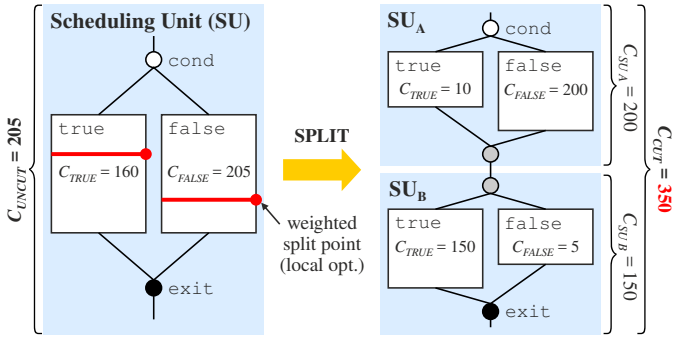


Figure 2: Bloated WCET estimate due to an unbalanced cut, caused by a branch-local optimization of split points.

This unfavorable behavior is rooted in the branch-local optimization of the migration cost. We address this issue by considering both criteria (i.e., size and migration cost) simultaneously across all branches to find globally suitable split points that lead to a balanced cut even for nested branches. Throughout this paper, we call this a *horizontal cut*.

### D. Splitting Loops

Further measures are required for the splitting of loops as splitting inside the loop body is entirely ineffective as the related costs outweigh the benefits. Additionally, an individual loop iteration typically contributes only a small fraction to the loop's overall WCET. Therefore, we subdivide loops at a granularity of whole iterations using index set splitting [6] to separate the index range of a loop into smaller subranges until the individual execution time fits the targeted size. As the resident-set size is usually the same for all loop iterations, we consider all possible split points as equally suited[1].

In general, we require an upper bound on the number of loop iterations $iter_{max}$ to estimate a loop's overall execution time, which is commonly available knowledge in (safety-critical) real-time systems. Considering the WCET $C_{loop}$ of a single loop iteration, we can derive the number of iterations $iter_{fit}$ required to fit the target size ($C_{target}$) of the scheduling units:

$$iter_{fit} = \lceil C_{target} / C_{loop} \rceil \tag{3}$$

The number of required cuts $n_{cut}$ results from the total number of iterations $iter_{max}$ and the fitting size $iter_{fit}$:

$$n_{cut} = \lfloor iter_{max} / iter_{fit} \rfloor \tag{4}$$

By splitting loops by their index range, we obtain suitable scheduling-unit size while avoiding the potentially disadvantageous effects of splitting the loop body.

In summary, by employing our concept of split-point graphs, we can successfully subdivide tasks into scheduling units of smaller size. We can efficiently cut both composite branches and loops. Liveness analysis allows us to identify split points with minimal migration costs. Thereby, we improve schedulability in multi-core settings and reduce the otherwise inevitable overapproximation of indirect migration overheads.

[1]Theoretically, loop constructs exist that violate this assumption. In that case, we overapproximate the resident-set size by the overall maximum.

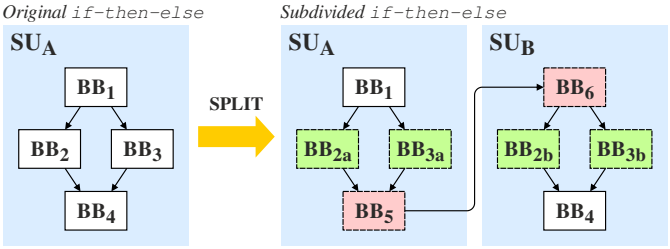


Figure 3: Schematic depiction of the split procedure for `if-then-else` branches.

## III. Implementation

We based the implementation of our approach on the *Real-Time Systems Compiler* (RTSC) [7]. The RTSC is an LLVM-based [8] toolchain that's characteristic feature is the automated manipulation of non-functional properties of real-time systems. For this purpose, the RTSC employs static code analysis to transform a given source system into an OS-agnostic intermediate representation, which in turn is based on the LLVM intermediate representation (LLVM-IR). As a result of this, the application's program-flow is represented as single-entry single-exit regions[2] between system calls, that is scheduling units that do not affect the internal state of the OS. Based on this intermediate representation, the RTSC facilitates the systematic manipulation of non-functional properties. For example, a conversion from event to time-triggered execution. Currently, the RTSC supports various real-time operating systems with its front and backends respectively [10], [11].

### A. Split-Point Graph Generation

Currently, only the application code is part of the intermediate system representation. Thus, with our prototype, we focused on the identification of suitable split point within the application, which is the main subject of migration. Conceptually, however, our approach can be extended to the operating-system code, which we, however, consider as future work. To identify optimal split points, we need to derive the split-point graph from the intermediate representation and enrich it with execution time estimates and resident-set sizes.

In a first step, we perform a liveness analysis of all variables on the intermediate representation of LLVM. The results of this analysis provide information about the size of the resident-set size $\omega$, which we then utilize as one optimization criterion.

In a second step, we estimate the execution time per instruction in the LLVM representation[3] to determine the distance $\delta$ from the intrinsic split points for each instruction. For this estimation, we rely on a simple model of the execution platform and assign execution costs to each instruction, giving more weight to classes of instruction, which are likely to be more expensive, for example, load and store instructions. This heuristic oversimplifies the process of modeling the WCET but is a reasonable approximation at the abstraction level of the intermediate representation. Deriving better estimates for the execution time is beyond the scope of this paper.

[2]A concept introduced as *Atomic Basic Blocks* (ABB) in [9].

[3]We inline all function calls to ease execution time estimation.

### B. Split Point Optimization

With the foundations laid, we can assess the local aptness of potential split points by a cost function that combines our two criteria with appropriate weights as shown in Equation 5. $\delta$ thereby denotes the distance to the intrinsic scheduling granularity, $\omega$ the migration costs in bits, and $w_\delta$ and $w_\omega$ the respective weights. Using complex weight functions, the interaction of weights and their effect on the assessment of split points can be influenced, for example, to exponentially punish an increasing distance $\delta$. For simplicity, we resort to a constant factor of one for both weights in our prototype:

$$w_\delta \cdot \delta + w_\omega \cdot \omega \tag{5}$$

In the case of sequential code, we can directly choose the point that is most suitable according to our cost function. However, as outlined in Section II, in the presence of loops and branches, we have to perform the global assessment of all branches to ensure a horizontal cut. For the search for the minimal horizontal cut among the split-point candidates in different branches, we apply the Ford-Fulkersson algorithm [12] on the split-point graph. After having identified a globally suitable split point across all branches, we then perform the actual splitting in `if-then-else` branches as depicted in Figure 3. At the split point in each branch, we cut the existing basic block (e.g., $BB_2$) in two parts by inserting instructions around the split point. In the first part (e.g. $BB_{2a}$), we set a flag indicating we executed that branch and jump to a newly created basic block terminating the first scheduling unit ($BB_5$). In the second part (e.g., $BB_{2b}$), we introduce a label which we can use as a jump target from the newly created entry basic block ($BB_6$) of the second scheduling unit.

For loops, we use the procedure shown at the example in Listings 1 and 2. Using the method outlined in Section II-D, we identify the number of loop iterations $iter_{fit}$ that should constitute one scheduling unit (5 in the example). We then split the loop by duplicating the body and adjusting the loop condition to preserve the program semantics. For this, we introduce a counter for the split loops (`sCo`) that has to be less than $iter_{fit}$ for the loop condition to be evaluated to `true`.

## IV. Evaluation

To assess our approach's ability to cope with the initial two challenges, its real-world usability ,and runtime, we conducted two experiments and one theoretical consideration.

```
1 LOOP_Bound(x:10);
2 for(int i = 0;
3  i < x; ++i)
4 {
5  ....
6 }
```

Listing 1: Loop in the original state.

```
1  int i = 0, C = 5;
2  for(; i < x && C; ++i) {
3   --C;
4   ....
5  }
6   ....
7   C = 5;
8  for(; i < x && C; ++i) {
9   --C;
10  ....
11 }
```

Listing 2: Loop after the splitting procedure.

### A. Runtime Overheads of Splitting

Since splitting of scheduling units is realised by the insertion of specific instructions, e.g., jump statements, at the designated cut point, it comes with an additional overhead. This overhead differs depending on the type of control flow. In the case of sequential control flow, we insert only one instruction per cut. The number of additional instructions for the splitting of branches $i_{if}^{+}$ depends on the number of cuts $n_{cut}$ and the number of branches $n_{branch}$. The first term of the sum shown in Equation 6 refers to instructions for setting a flag marking the active branch. The second term represents the jump instruction that terminates the first scheduling unit and the third term contains instructions for checking the stored flag and proceeding with the correct branch.

$$i_{if}^{+} = n_{cut} * n_{branch} * 2 + n_{cut} * 1 + n_{cut} * 3 \quad (6)$$

Splitting loops adds $i_{loop}^{+}$ instructions to the instructions of the original loops. The first term of Equation 7 refers to instructions necessary to maintain an additional counter for the iterations planned in each scheduling unit of the loop. The second term corresponds to the end of each scheduling unit at a split point and comprises instructions for exiting the scheduling unit and resetting the additional iteration counter. The third term contains instructions to decide whether to execute the following part of the loop after each split point.

$$i_{loop}^{+} = (n_{cut} + 1) * 5 + n_{cut} * 2 + n_{cut} * 3 \quad (7)$$

The runtime overhead introduced by splitting is therefore minor compared to the execution time of most scheduling units. Exceptions are the extreme cases of conditional constructs with a large number of branches and loops with small bodies. However, we can detect and avoid these cases in the search for suitable split points, which we consider future work.

### B. Schedulability of Synthetic Benchmarks

We assessed the validity of our approach concerning schedulability by generating 12000 synthetic benchmark systems with a utilization between 3.5 and 4.0 comprising few OSEK-compliant tasks. We tried to find a feasible allocation and schedule for each task set on a system with four processor cores by employing specialized versions [13] of the branch-and-bound allocation algorithm and the minimax scheduling algorithm, both by Peng et al. [14]. Figure 4 shows the achieved relative schedulability of the generated task sets both with (left bars) and without (right bars) automated splitting of scheduling units. The results show that the relative schedulability increases through splitting, leading to 70 percent more schedulable task sets for the highest utilization. This confirms the general capability of our approach to employ migration automatically to achieve better utilization.

### C. Minimize Migration Costs

To evaluate our approach regarding Challenge #2, we analyzed possible splitting points in real-world benchmarks taken from the TACLeBench suite [15]. For this, we converted the benchmarks to OSEK tasks and created OSEK systems

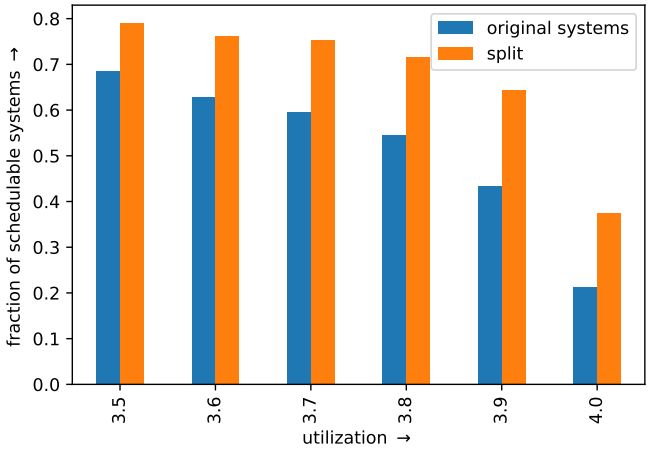


Figure 4: Relative schedulability of generated task sets achieved with (left bars) and without (right bars) splitting of scheduling units, showing an increased schedulability reached through automated splitting.

comprising one benchmark task and two load tasks. We adjusted the amount of load to achieve a system which is unschedulable on two cores without migration to force our RTSC extension to split the benchmark task. We recorded the worst-case dynamic migration costs for every instruction as well as the statically composed migration costs of split points that comprise all branches of the horizontal cut. Table I shows an overview of both the worst-case migration cost observed in all possible split-point candidates as well as the migration cost of the split point chosen by our approach for several TACLeBench benchmarks we analyzed. The costs represent the size of the resident set in bits based on LLVM IR types, which allow for a variable type width from 1 to $2^{23} - 1$ bits. The results indicate that our approach is capable of providing worst-case migration costs for a whole horizontal split that are beneath the dynamic worst-case of single branches, with a reduction of up to 76 percent. These improvements in worst-case migration overhead ultimately allow reducing the pessimism in the response-time analysis.

## V. Related Work

Studies on migration in real-time systems agree that migration costs vary significantly with the resident-set size [4], [16],

| Benchmark | Worst-case Resident-set Size [bits] | Split-point Resident-set Size [bits] |
|---|---|---|
| binarysearch | 225 | 224 |
| bitonic | 65 | 64 |
| complex_update | 480 | 288 |
| countnegative | 2176 | 1568 |
| filterbank | 60 736 | 60 704 |
| iir | 432 | 400 |
| insertsort | 544 | 128 |
| minver | 17 568 | 16 800 |
| petrinet | 5057 | 5056 |

Table I: Comparison of the worst-case dynamic resident-set size and the resident-set size at the split point chosen by our approach in bits on the basis of LLVM IR types.

[17]. There exists a large body of related work on scheduling algorithms [2], [18] that assume a constant upper bound on migration overheads. To increase predictability and reduce costs, various approaches focused on restricting preemption and finding thresholds or placements for preemption points [17], [19]–[23]. Anderson et. al [24] extended this concept to restricted migration, but lack a practical implementation. Automatic analysis and generation of multi-core systems [13], [25], [26] for non-preemptive scheduling has been studied. All these approaches either ignore migration or are accompanied by substantial pessimism, yet they provide a good starting point for the practical application of our migration hits at runtime.

Orthogonal to our approach of (horizontal) splitting is (vertical) slicing of real-time applications. Here, the time-sensitive code is separated from time-insensitive code to enhance schedulability [27]. In contrast to our approach, it is difficult to control and optimize the output of code slicing timing-wise. In a safety-critical systems, checkpointing [28], [29] is considered to partition tasks, wherein the state to be saved is minimized. However, checkpointing is specific to the application and not universally applicable. Sarkar et al. [5] proposed hardware-assisted migration, from which our approach would also benefit but on which it does not depend.

## VI. Conclusion & Outlook

In this paper, we presented an approach to boost migration in real-time systems by an automated analysis of tasks at the source-code level. Our analysis reveals beneficial split points with minimal migration costs. On the one hand, this knowledge can be exploited by the RTOS at runtime and thus is an enabler for migration thresholds; analogous to preemptive thresholds or limited preemptive scheduling [20]. Consequently, static WCET analysis can infer tighter bounds on migration overheads. On the other hand, our toolchain supports the subdivision of tasks into smaller scheduling units already at compile time, thereby improving overall schedulability of static allocation schemes.

We continue to make migration in real-time systems more accessible and predictable and are currently working on the following topics: (1) Improving the WCET heuristics used for splitting, for example, by adding more precise hardware models that allow for calculating migration costs based on cache lines instead of data bits. (2) Adapting an existing RTOS to support migration thresholds. (3) Extending the analysis to the OS implementation and the system calls respectively.

*Source code is available:*
`www4.cs.fau.de/Research/RTSC/experiments/abbslicing/`


## Acknowledgment

This work is supported by the German Research Foundation (DFG) under grants no. SCHR 603/14-2, SCHR 603/13-1, SCHR 603/9-2, the CRC/TRR 89 Project C1, and the Bavarian Ministry of State for Economics under grant no. 0704/883 25.


## References


[1] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[2] 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.

[3] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comp. Sur.*, vol. 43, no. 4, p. 35, 2011.

[4] A. Bastoni, B. Brandenburg, and J. Anderson, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability," *Proceedings of OSPERT '10*, pp. 33–44, 2010.

[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] M. Griebl, P. Feautrier, and C. Lengauer, "Index set splitting," *Int'l Journal of Parallel Prog'ing*, vol. 28, no. 6, pp. 607–631, Dec. 2000.

[7] F. Scheler and W. Schröder-Preikschat, "The real-time systems compiler: Migrating event-triggered systems to time-triggered systems," *Software: Practice and Experience*, vol. 41, no. 12, pp. 1491–1515, 2011.

[8] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the Int'l Symp. on Code Generation and Optimization*, Washington, DC, USA, 2004.

[9] F. Scheler and W. Schröder-Preikschat, "Synthesising real-time systems from atomic basic blocks," in *Proc. of RTAS '06 WIP*, 2006.

[10] M. Stilkerich, J. Schedel, P. Ulbrich, W. Schroder-Preikschat, and D. Lohmann, "Escaping the bonds of the legacy: Step-wise migration to a type-safe language in safety-critical embedded systems," in *Proc. of ISORC '11*, March 2011, pp. 163–170.

[11] S. Vaas, P. Ulbrich, M. Reichenbach, and D. Fey, "Application-Specific Tailoring of Multi-Core SoCs for Real-Time Systems with Diverse Predictability Demands," *Signal Processing Systems*, Jul. 2018.

[12] L. R. Ford and D. R. Fulkerson, "A simple algorithm for finding maximal network flows and an application to the hitchcock problem," *Canadian Journal of Mathematics*, vol. 9, pp. 210–218, 1957.

[13] 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*, 2018, pp. 93–103.

[14] D.-T. Peng, K. G. Shin, and T. F. Abdelzaher, "Assignment and scheduling communicating periodic tasks in distributed real-time systems," *IEEE Trans. on Software Eng'ing*, vol. 23, no. 12, pp. 745–758, 1997.

[15] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research," in *Proc. of WCET '16*, vol. 55, 2016, pp. 2:1–2:10.

[16] C. Burguière, J. Reineke, and S. Altmeyer, "Cache-related preemption delay computation for set-associative caches–pitfalls and solutions," in *OASIcs-OpenAccess Series in Informatics*, vol. 10, 2009.

[17] 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.

[18] A. Burns and R. Davis, "Mixed criticality systems – a review," Tech. Rep. 9. Edition, 2016.

[19] B. Peng, N. Fisher, and M. Bertogna, "Explicit preemption placement for real-time conditional code," in *Proc. of ECRTS*, 2014, pp. 177–188.

[20] 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.

[21] M. Bertogna, G. Buttazzo, and G. Yao, "Improving feasibility of fixed priority tasks using non-preemptive regions," in *Proc. of RTSS '11*. IEEE, 2011, pp. 251–260.

[22] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo, "Optimal selection of preemption points to minimize preemption overhead," in *Proc. of ECRTS '11*, Jul. 2011, pp. 217–227.

[23] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Proc. of RTCSA '99*. IEEE, 1999, pp. 328–335.

[24] 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.

[25] 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.

[26] F. Nemati, J. Kraft, and T. Nolte, "A framework for real-time systems migration to multi-cores," Tech. Rep., 2009.

[27] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *Softw. Eng. Notes*, vol. 30, no. 2, pp. 1–36, 2005.

[28] K. G. Shin, T.-H. Lin, and Y.-H. Lee, "Optimal checkpointing of real-time tasks," *IEEE T. on Comp.*, vol. 100, no. 11, pp. 1328–1341, 1987.

[29] S. W. Kwak, B. J. Choi, and B. K. Kim, "An optimal checkpointing-strategy for real-time control systems under transient faults," *IEEE Trans. on Reliability*, vol. 50, no. 3, pp. 293–301, 2001.