FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Helene Gsänger

Dynamic Migration Decisions in Multicore Systems

Masterarbeit im Fach Informatik

30. November 2020

Please cite as:
Helene Gsänger, “Dynamic Migration Decisions in Multicore Systems”,
Master’s Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU),
Dept. of Computer Science, November 2020.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany

Dynamic Migration Decisions in Multicore Systems

Masterarbeit im Fach Informatik

vorgelegt von

Helene Gsänger

geb. am 22. November 1992
in Nürnberg

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Dr.-Ing. Peter Ulbrich**
	Phillip Raffeck, M.Sc.
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat**
Beginn der Arbeit:	**1. Juni 2020**
Abgabe der Arbeit:	**30. November 2020**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Helene Gsänger)
Erlangen, 30. November 2020

ABSTRACT

Since multicore platforms are becoming more common for real-time systems, multicore real-time scheduling algorithms become more relevant. For a more efficient use of the additional processing capacity of multiple cores, many of these algorithms allow tasks to migrate between cores, which improves the schedulability of task systems, but comes at the cost of additional migration overhead. One possible approach to reduce this overhead is the restriction of migration to predefined migration points that are known to be beneficial. For semipartitioned scheduling algorithms, an approach exists that identifies a set of potential migration points at which migrating tasks can be split statically, so that at run time, each job of the task migrates at this point. This has the disadvantage that it is impossible to avoid task migration, even if the actual run times of the current job would otherwise allow the job to finish on its current core.

This thesis presents a solution for this problem by introducing dynamic migration decisions. Instead of using a statically selected point, an algorithm for dynamic migration decisions tries to choose the latest possible migration point out of the predefined set, depending on the run-time behaviour of the current job. In order to use as much run-time information as possible, the presented algorithms use the concept of evaluation points, which are defined as the latest possible points at which migration decisions can be made. For each migrating job, an evaluation point is initially set and repeatedly recalculated, until migration decisions can no longer be delayed and a migration point is selected. From this approach, three algorithms are derived, which differ in the way evaluation points are defined. The algorithms presented in this thesis define evaluation points as points in code, points in execution time, and a combination of both, respectively.

All presented algorithms can be combined with any semipartitioned scheduling algorithm. It is shown that dynamic migration decisions do not cause any deadline misses in any task set, if the task set is schedulable with fixed migration points. An analysis of the additional required operations shows, that both the number of evaluation points, and the required effort for each evaluation point are at most logarithmic for all algorithms in the average case. Since the required effort decreases with increasing run times, the static assignment of split tasks to cores needs to consider only a constant overhead for each part of a split task.

All presented algorithms have been implemented on a Raspberry Pi v2 model B, based on a FreeRTOS port. Time measurements of this implementation indicate, that the overhead for dynamic migration decisions is unlikely to impact the schedulability of a given task system. While dynamic migration decisions make it possible to avoid migration if the current run times allow it, the measured response times show no significant adverse impact of the additional overhead of the presented algorithms.

KURZFASSUNG

Da Echtzeitsysteme immer häufiger Mehrkernplattformen verwenden, werden Echtzeit-Schedulingalgorithmen für mehrere Kerne immer relevanter. Um die zusätzliche Kapazität mehrerer Kerne effizienter zu nutzen, erlauben viele dieser Algorithmen Tasks, zwischen Kernen zu migrieren, was die Planbarkeit verbessert, aber zu zusäzlichen Migrationsoverheads führt. Eine möglicher Ansatz zur Verminderung dieses Overheads ist die Einschränkung von Migration auf als vorteilhaft bekannte, vordefinierte Migrationspunkte. Für semipartitionierte Schedulingalgorithmen existiert ein Ansatz, der eine Menge potentieller Migrationspunkte definiert, an denen migrierende Tasks statisch geteilt werden können, sodass jeder Job dieses Tasks zur Laufzeit an diesem Punkt migriert. Das hat den Nachteil, dass Taskmigration nicht vermieden werden kann, selbst wenn die tatsächliche Laufzeit des aktuellen Jobs es erlauben würde, den Job auf dem aktuellen Kern zu fertigzustellen.

Diese Arbeit präsentiert eine Lösung für dieses Problem, indem dynamische Migrationsentscheidungen eingeführt werden. Statt einen statisch festgelegten Punkt zu verwenden, versucht ein Algorithmus für dynamische Migrationsentscheidungen, den spätestmöglichen Migrationspunkt aus der vorgegebenen Menge auszuwählen, abhängig von verfügbaren Laufzeitinformationen über den aktuellen Job. Um möglichst viele Laufzeitinformationen zu verwenden, verwendet der vorgestellte Algorithmus das Konzept von Evaluationspunkten, die als spätestmögliche Punkte definiert sind, an denen Migrationsentscheidungen getroffen werden können. Für jeden migrierenden Job wird initial ein Evaluationspunkt gesetzt und wiederholt neu berechnet, bis Migrationsentscheidungen nicht mehr aufgeschoben werden können und ein Migrationspunkt ausgewählt wird. Ausgehend von diesem Ansatz werden drei Algorithmen abgeleitet, die sich in der Art der Definition von Evaluationspunkten unterscheiden, Die in dieser Arbeit präsentierten Algorithmen definieren Evaluationspunkte als Punkte im Code, Punkte in der Ausführungszeit bzw. als Kombination aus Beidem.

Alle präsentierten Algorithmen können mit beliebigen semipartitionierten Schedulingalgorithmen kombiniert werden. Es wird gezeigt, dass dynamische Migrationsentscheidungen keine Deadlineüberschreitungen verursachen, wenn das gegebene Task Set mit statisch festgelegten Migrationspunkten planbar ist. Eine Analyse der benötigten Operationen zeigt, dass im durchschnittlichen Fall sowohl die Anzahl an Evaluationspunkten, als auch der benötigte Aufwand für jeden Evaluationspunkt maximal logarithmisch zur Anzahl der Migrationspunkte ist. Da sich der erforderliche Aufwand mit steigenden Laufzeiten verringert, muss die statische Zuweisung von geteilten Tasks zu Kernen nur einen konstanten Overhead für jeden Teil eines aufgeteilten Tasks berücksichtigen.

Alle vorgestellten Algorithmen wurden auf einem RaspBerry Pi v1 Modell B implementiert, ausgehend von einem Port von FreeRTOS. Zeitmessungen dieser Implementierung deuten darauf hin, dass eine Beeinträchtigung der Planbarkeit eines gegebenen Task Sets durch dynamische Migrationsentscheidungen unwahrscheinlich ist. Während dynamische Migrationsentscheidungen bei ausreichend kurzen Laufzeiten die Vermeidung von Migration erlauben, zeigen die gemessenen Antwortzeiten keinen signifikanten nachteiligen Einfluss des Overheads der präsentierten Algorithmen.

CONTENTS

INTRODUCTION 1

The trend towards multicore systems also affects real-time scheduling, and poses new problems for scheduling algorithms in theory as well as in practical application, such as the tradeoff between schedulability and the reduction of overhead for task migration. One possible solution for this specific problem is the restriction of task migration to predefined migration points. This thesis presents an improvement of an already existing approach to restrict migration. In the existing approach, migrating tasks are determined to migrate at a statically defined migration point. For this approach, this thesis will introduce dynamic migration decisions, so that a migration point is chosen at run time, out of a set of statically defined potential migration points. This makes it possible to delay or even avoid migration, if the actual run times of the task allow this.

This chapter will first clarify the benefits of dynamic migration decisions, and later provide an overview over the remaining thesis.

1.1 Motivation

In the last decades, progress in hardware development has led to a continuing increase in processor speed. This increase in speed was first achieved by increasing the processor frequency, but due to energy consumption and heating problems, this approach was only viable up to a certain point [BBB15]. After a frequency ceiling was hit, instead of increasing the processor speed, hardware platforms with multiple processors were developed. This trend has not only affected general purpose platforms, but also embedded environments. With this trend, and due to increasing workload of real-time applications, real-time systems are increasingly often run on multicore platforms [BBB15].

Adapting real-time scheduling algorithms for multicore platforms is, however, not trivial, and multiple different approaches have been developed in order to solve this problem [DL78]. In order to use the processing capacity of additional cores more efficiently, many multicore scheduling algorithms allow tasks to migrate between different cores. Consider, for example, a task set consisting of three tasks, each of which has a period of three time units and a worst-case execution time (WCET) of two time units. Out of this task set, no two tasks can be fully fit on one core, which renders the task set unschedulable on two cores without task migration. With task migration, however, one task can be split in two parts, each of which will be executed on a different core, so that the task set is schedulable on two cores.

While it is beneficial for scheduling, tasks migration leads to additional overhead. A significant part of this overhead is caused by cache misses, when the migrated task is resumed. These cache effects can be caused by migration as well as preemption and are called cache-related preemption and migration delay (CPMD) [BBA10]. For migrations, this overhead is even higher than for preemptions, even though the difference decreases with increasing load [BBA10]. Estimations of

CPMD are difficult, with worst-case overhead significantly higher than the average case [BBA10] [HP09], which is especially disadvantageous for real-time scheduling, since in order to prevent deadline misses, the worst-case run times have to be considered.

Since CPMD is caused by cache misses, the extent of the overhead depends on the size of the active working set at the time of migration [BBA10]. This observation can be leveraged in the reduction of migration overhead by restricting task migration to a set of statically defined points in the code with a small working set. There are multiple approaches that use this strategy to reduce migration overhead, and this thesis provides an improvement of one of these approaches. The approach in question first defines multiple migration points for each task [Kla+19]. Depending on the requirements of the task set, one of these migration points is then chosen statically for each required task migration. The downside of this approach is that each migrating task will always migrate at the same point, regardless of actual run times. Even if faster run times would make it possible to avoid migration entirely, split tasks still migrate at their assigned migration points, which not only increases the response time of the task itself, but also unnecessarily impacts other tasks by increasing the traffic on the network on chip.

This thesis provides a solution for this problem by introducing dynamic migration decisions. Instead of always migrating at a fixed point, out of the initially defined set of migration points, one migration point will be chosen at run time out of a given set of potential migration points at run time, depending on the actual run-time behaviour of the task. If the task runs faster than expected, migration might be delayed or even avoided.

1.2 Organization of this Thesis

The remaining thesis is structured as follows. In order to provide some context for the algorithms presented later in this thesis, Chapter 2 will provide some context over some fundamental concepts of real-time scheduling in general, and real-time scheduling in multicore systems in particular, before explaining some existing approaches to reduce CPMD. Based on this, Chapter 3 will state the general requirements for dynamic migration decisions and present multiple algorithms for selecting migration points at run time. The efficiency of these algorithms and the additional overhead they need will be discussed in Chapter 4, before implementation details of the presented algorithms are discussed in Chapter 5. Chapter 6 will present measurements results of both overhead for migration decisions and response times of tasks with the given algorithms, while Chapter 8 will provide some concluding remarks and point out some potential improvements of the presented algorithms.

FUNDAMENTALS 2

This thesis presents an improvement to an existing approach to reduce migration overhead in semipartitioned scheduling, which is a category of multicore scheduling algorithms in real-time systems. Real-time systems are operating systems for applications that need to adhere to predefined timing constraints, because any violation of these constraints can lead to negative consequences. Examples are multimedia systems, embedded systems such as cell phones, but also safety-critical systems such as control systems for automobiles or power plants. Given the importance of timing requirements, any scheduling algorithm for real-time systems has to ensure that all deadlines can be met.

In order to ensure this, a model is needed to represent timing behaviour and constraints of the scheduled tasks. The task model that will be used in this thesis will be presented first in the remaining chapter. Based on this task model, some further fundamental concepts of real-time scheduling will be explained, before elaborating on the problems of multicore scheduling, as well as some approaches to solve these problems.

2.1 Task Model

In comparison to general-purpose systems, the tasks for real-time systems and their timing behaviour are known beforehand. The timing behaviour of a task can be represented in a task model, which is used by a real-time scheduling algorithm in order to make scheduling decisions or to verify that timing constraints can be met. While there are different ways to represent tasks, in this thesis, the three-parameter task model is used, which will be presented in this section. As the name indicates, this model represents each task by three parameters. Based on these parameters, various properties of tasks and task sets can be determined. Some of these properties will be explained later in this section, together with some general assumptions about tasks.

The model that will be used in this work, and which will be explained in the remaining section, is based on the description in [BBB15].

2.1.1 The Three-Parameter Task Model

The set of tasks to be scheduled is known before run time. It is comprised of n tasks and referred to as $\tau = \{\tau_1, \ldots, \tau_n\}$, with each task represented by some τ_i. At run time, each task releases an infinite number of jobs, which are instances of tasks and must be executed within the given time limits. These timing constraints, as well as other timing behaviour of a task are specified by three parameters. With these parameters, each task τ_i can be defined as follows:

$$\tau_i = (C_i, D_i, T_i)$$

C_i represents the worst-case execution time (WCET) of a task. For any task τ_i, each job is guaranteed to finish in no more than C_i time units of execution. Since scheduling decisions rely on the correctness of the specified C_i, a pessimistic estimation must be used, and any overhead of a task, such as overhead caused by interrupts, or cache effects following preemptions or migrations, must be included in the WCET.

Timing constraints of task τ_i are represented by its relative deadline D_i. D_i specifies, that every job released by τ_i must be completed until D_i time units after its release. If missed deadlines only lead to mild consequences, deadlines can be defined as soft of firm deadlines, which means that deadline misses are acceptable to a certain degree. In this thesis, however, all deadlines are assumed to be hard deadlines, which means that no job is allowed to miss its deadline.

The time between two job releases of the same task is specified by the parameter T_i of a task τ_i. For periodic tasks, this parameter refers to the exact time between two successive job releases, while for sporadic tasks it defines the minimal time.

2.1.2 Properties of Task Sets

Based on the specified parameters of a task set, the suitability of a scheduling algorithm for a given task set can be verified. However, in many cases, it is more useful to make more general statements about groups of task sets instead of some given task set in particular. For this purpose, different properties of tasks and task sets can be used. Examples for these properties are the utilization of tasks and task sets, as well as the classifciation of task sets based on the relation between deadline and period.

Based on the latter relation, different groups of task sets can be defined. Task sets that contain only tasks, for which the deadline equals the period are called implicit-deadline task sets. If the relative deadline of each task in a task set is lower than or equal to its period, this task set is referred to as constrained-deadline task set, whereas otherwise, the task set is called an arbitrary-deadline task set.

Another useful value is the utilization of tasks and task sets. The utilization of a task is the ratio of its WCET and its period, and is represented by $u_i = \frac{C_i}{T_i}$. The utilization of a task set is the sum of the utilization of all tasks, so that $U_{sum}(\tau) = \sum_{\tau_i \in \tau} u_i$. Another useful value is the maximal utilization $U_{max}(\tau) = \max\{u_i \mid \tau_i \in \tau\}$.

2.1.3 General Assumptions about Tasks

In order to make scheduling decisions, some assumptions must be made about the scheduled tasks. This thesis is based on the following assumptions.

- Each job is ready for execution immediately after its release, and can therefore be scheduled without delay.
- All tasks are fully preemptive, so that the job with the currently highest priority can be scheduled immediately after its release.
- No job relinquishes the processor on its own, before it has finished execution.
- All tasks are independent from each other, so that the timing behaviour of each task can be determined independent of the remaining task set.

- Any overhead is already included in the specified WCETs, so that no additional overhead needs to be considered.

While these assumption do not necessarily represent the realistic behaviour of real-time task systems, many results regarding the schedulability of task sets rely on these assumptions. Therefore, these constraints will also be assumed to be valid for the algorithms that will be presented in later chapters.

2.2 Scheduling in Real-Time Systems

The specification of the timing behaviour of a task set can be used by a real-time scheduling algorithm, in order to make scheduling decisions. Before the tasks are executed, however, the scheduling algorithm has to verify that the given task set can be scheduled without violating any timing constraints. If a given task set can be scheduled by some algorithm A without deadline misses, this task set is called A-schedulable.

Different scheduling algorithms can be compared to each other, with regards to the task sets that are schedulable by these algorithms. For this comparison, several metrics exist. One of this metrics is the utilization bound of a scheduling algorithm. If a scheduling algorithm has a utilization bound of u, then each task set with a utilization of u or less is schedulable by this algorithm. An algorithm A is called optimal, if any task set for which any schedule exists that meets all deadlines is also A-schedulable.

Scheduling decisions can be made statically or at run time. In this thesis, only scheduling algorithms that make decisions at run time are considered. These decisions are made according to priorities that are assigned to jobs, so that at each time instant, the job with the highest priority is running. The assignment of priorities can be more or less flexible, depending on the scheduling algorithm. Fixed task-priority algorithms assign priorities to tasks statically, so that each job has the priority that was statically assigned to the corresponding task. In fixed-job priority algorithms, jobs of the same task can have different priorities, but the priority assigned to a job does not change during its execution. Algorithms in which the priority of a job can change arbitrarily are called dynamic-priority algorithms.

Examples for real-time scheduling algorithms are Earliest Deadline First (EDF) [LL73] and Deadline Monotonic (DM) [Aud+00]. EDF assigns the highest priority to the job with the earliest absolute deadline. Since, with this policy, jobs of the same task can have different priorities, whereas the priority of a job, relative to other released jobs, does not change, EDF is a fixed job priority algorithm. EDF is known to be optimal for uniprocessor scheduling, and has a utilization bound of 1 for implicit-deadline systems [LL73]. DM is a fixed-task priority algorithm that assigns the highest priority to the task with the shortest relative deadline. For constrained-deadline task sets on unicore platforms, DM is the optimal fixed-priority scheduling algorithm [LW82].

2.3 Multicore-Scheduling in Real-Time Systems

On multicore platforms, more than one task can execute at each time instant. Which task can be scheduled on which core and at which speed depends on the hardware platform. In this thesis, only homogeneous multicore systems are considered. This means that each task can execute on each core at the same speed [DB11].

But even with this constraint, real-time scheduling is still more complicated on multicore platforms than it is on unicore platforms, and results of unicore scheduling are not necessarily valid for multicore scheduling [DL78]. In order to overcome these problems and to leverage the additional processing capacity of multicore platforms, various scheduling algorithms have been developed and analyzed. Depending on the degree to which tasks are allowed to migrate between cores, these algorithms can be classified into global, partitioned and semipartitioned scheduling algorithms.

2.3.1 Global Scheduling

Global scheduling does not restrict task migration, and tasks are assigned to cores dynamically, so that at each time instant, the tasks with the highest priority are executing on the available cores.

While unicore scheduling algorithms can still be applied, the results for these algorithms regarding the use of available processing capacity are not transferable. An example for this is EDF. In global scheduling, EDF is no longer optimal [DB11], and even with m cores, the utilization bound is still 1. This was shown by Dhall and Liu [DL78], based on a constructed implicit-deadline task set with utilization of $1 + m * \epsilon$ that is unschedulable by EDF on m cores. This task set consists of m tasks with period 1 and WCET $2 * \epsilon$, and another task with period $1 + \epsilon$ and WCET 1. When this task set is scheduled on m cores, using EDF, then all available processors will be used by the shorter tasks first, so that the longer task misses its deadline. With $\epsilon \to 0$, $U_{sum}(\tau) \to 1$, which shows, that the utilization bound of global EDF is no more than 1, regardless of the number of cores.

There is, however, an optimal algorithm for global scheduling. This algorithm is called Pfair scheduling [BBB15]. A pfair scheduler breaks each job into subjobs of the length of one time unit. At run time, the scheduler ensures, that for each job, the number of finished subjobs at each time instant is proportional to the utilization of its task, with a deviation from this assigned share of no more than one time unit. An implicit-deadline task set τ is Pfair-schedulable on m cores, if $U_{sum}(\tau) \leq m$ and $U_{max}(\tau) \leq 1$, which makes the algorithm optimal for implicit-deadline task sets. Despite its optimality, this algorithm is rarely used because of practical considerations. Aside from a complicated implementation, scheduling decisions are needed at each time unit, which leads to a high overhead due to the number of required preeemptions and migrations.

Regardless of the algorithm, global scheduling has several drawbacks. Since tasks can run on each core, additional synchronization between cores is needed, in order to coordinate the access to tasks. With no restriction on task migration, migrations are difficult to predict, which leads to very pessimistic estimations of the required migration overhead [DB11].

2.3.2 Partitioned Scheduling

Partitioned scheduling does not allow any task migration, so that each task can only be executed on its assigned core. Since tasks are assigned to cores before run time, partitioned scheduling has two phases. In the assignment phase, tasks are assigned to cores statically, before they are run in the execution phase.

In the execution phase, the tasks are scheduled separately on each core. Since this allows the dynamic scheduling algorithm to ignore all other cores, results and algorithms of unicore scheduling can be applied. Furthermore, neither coordination between cores nor task migration are needed, which reduces the overhead that is needed for this approach.

The partitioning phase is, however, more complicated [DB11]. Finding an assignment of tasks to cores that makes optimal use of the available processing capacity is proven to be equivalent to the bin-packing problem [DL78], which is known to be NP-hard in the strong sense. Thus, partitioning algorithms have to use heuristics instead of exact algorithms.

But even with an optimal assignment, the granularity of tasks can impact the use of the available processing capacity. This can be shown by an example of a task set τ, consisting of $m + 1$ tasks τ_i with an utilization of $u_i = \frac{1}{2} + \epsilon$. With the given utilization, no more than one task can be scheduled on each core, so that no partitioned scheduling algorithm is able to schedule τ on a platform with m cores. With $\epsilon \to 0$, this leads to a utilization bound of no more than $\frac{m+1}{2}$ [DB11].

This means, that despite needing less overhead at run time, the static assignment of tasks to cores is complicated and the use of the available processing capacity can be heavily impacted by tasks with high utilization.

2.3.3 Semipartitioned Scheduling

Semipartitioned scheduling is a modified version of partitioned scheduling that allows limited migration, in order to reduce the unused processing capacity. As in partitioned scheduling, semipartitioned scheduling algorithms consist of an assignment and an execution phase.

The assignment phase starts as in partitioned scheduling, by assigning tasks to cores. The approaches differ, however, when the currently processed core has not enough processing capacity left for any entire remaining task. In this case, semipartitioned scheduling allows the splitting of one remaining task, so that the first part of this task can be assigned to the current core. The remaining part of this task is later assigned to one or multiple other cores. At run time, split tasks will migrate between cores. Migration can happen either between jobs, or during job execution. Both of these approaches have their own benefits and drawbacks.

2.3.3.1 Migration between jobs

For migration between jobs, each job of a migrating task is statically assigned to one core. At run time, a task migrates only after the current job has been finished. This approach is used in different semipartitioned scheduling algorithms, such as EDF-fm [ABD05], or the algorithm proposed by Dorin et al [Dor+10]. The advantage of this approach is the relatively low migration cost, since less data needed between two different jobs, which leads to fewer cache misses. Furthermore, the required coordination between cores is relatively simple. The drawback of this approach is the difficulty of load distribution. Since each job executes on its assigned core until it is finished, its load cannot be distributed evently on its assigned cores, which can lead to a short-term overload on the assigned core in some algorithms [ABD05]. This is especially disadvantageous for hard-deadline task sets. Anderson et al. even state that this approach cannot be optimal for sporadic task sets with hard deadlines [And+14].

2.3.3.2 Migration during job execution

In order to distribute the load of migrating tasks more evenly, many semipartitioned scheduling algorithms allow task migration during job execution, after a job has executed for a specified amount of time. Instead of assigning entire jobs to one core, the partitioning algorithm splits migrating tasks into multiple parts, which are then assigned to different cores. Examples for this approach are EDF-WM [KYI09], EDF-SS [ABB08], or EKG [AT06]. If task migration during job execution is allowed, additional coordination between cores is needed, in order to avoid the simultaneous execution of multiple parts of the same task. There are different approaches to solve this problem. One possible solution, used by EDF-SS [ABB08] is to define fixed time windows, in which each partial task is allowed to migrate. Alternatively, split tasks can be modeled as separate tasks, with their own WCETs and deadlines. In this model, jobs of migrated tasks are not released, until the preceeding job has finished execution. This approach is used by EDF-WM [KYI09] and another algorithm by Burns et al [Bur+12].

2.3.3.3 Summary of semipartitioned scheduling

In summary, semipartitioned scheduling has several advantages compared to global or partitioned scheduling. Compared to global scheduling, task migration is limited to some statically known tasks. Each job of a migrating task can migrate at most once per split, and for each migration, the target core is known in advance. Since this limits the number of migrations and increases the predictability of migrations, semipartitioned scheduling allows more optimistic estimations of migration overhead.

Compared with semipartitioned scheduling, more task sets are schedulable, if tasks can be split. However, task splitting requires additional synchronization between cores and causes migration overhead [BBA11].

2.4 Approaches to reduce Cache-Related Preemption and Migration Delays (CPMD)

As discussed in the previous section, algorithms that improve schedulability in theory, often increase the overhead needed for preemptions and migrations. In order to make these theoretically more efficient algorithms more viable in practical application, several approaches have been developed to reduce this overhead.

2.4.1 Approaches to reduce Preemption Overhead

There exist several strategies to reduce the overhead caused by preemptions. Some of these approaches are already included in semipartitioned scheduling algorithms. For example, EDF-SS and EKG both have parameters that allow to decrease the number of preemptions, at the cost of schedulability [ABB08] [AT06].

Other approaches exist independent of specific algorithms, such as the concepts of preemption thresholds and preemption points. A preemption threshold is an additional parameter that can be asigned to a task or job, in order to limit preemptions. A job with a given preemption threshold can only be preempted by another job with a priority that exceeds the specified preemption threshold. With a preemption threshold higher than the actual priority, the likelyhood of preemptions decreases [BBY13]. If preemption points are used, tasks can only be preempted at statically defined preemption points. This makes preemptions more predictable, and, if beneficial preemption points are chosen, the worst-case overhead for preemptions is reduced [BBY13].

2.4.2 Approaches to reduce Migration Overhead

Approaches to reduce migration overhead exist in both hardware and software. Sarkar et al. propose a hardware-based solution to reduce cache-related delays by transferring cache lines of migrating tasks between cores [Sar+09]. This transfer is initiated immediately before the task migrates, so that the transfer has ideally already been completed, before the migrating task is resumed. Since this requires the specification of the target core before a task migrates, this approach can only be applied, if the migration target is known in advance. In order to reduce the time for the cache-line transfer, an additional functionality is presented to limit the transfer to memory regions that can be specified by the application before a task migrates.

Software solutions to reduce migration overhead include the specification of migration points, analogous to the previously described use of preemption points. As with preemption points, the use of migration points increases predictability, and can reduce the worst-case migration overhead, if beneficial migration points are chosen. Migration points are used by different approaches.

Bertozzi et al. propose the use of migration points in global scheduling. Migration points can be chosen by the developer, who can also provide additional information about the data that needs to be transferred at this point. At run time, any migration request is delayed, until the next migration point is reached [Ber+06].

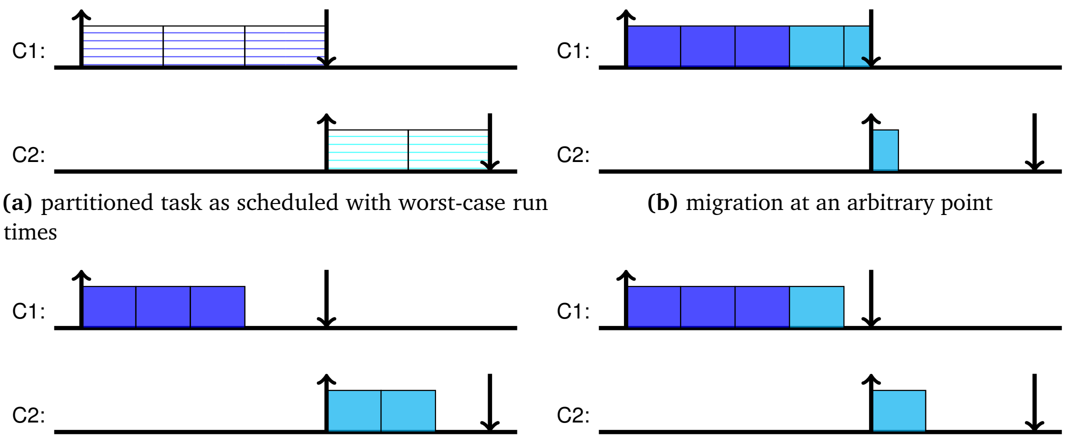
Migration points can also be used in semipartitioned scheduling [Kla+19]. Before a task set is partitioned, an algorithm chooses potential migration points for each task. Aside from the working set size, the definition of migration points also considers the distance to other migration points, in order to simplify task splitting. In the assignment phase, migrating tasks are split at one of their predefined migration points. At run time, each split task will migrate, when this migration point is reached.

2.5 Contributions of this thesis

This thesis is an improvement of the previously described approach of automated task splitting. One drawback of this approach is the static determination of designated migration points, regardless of actual run times. If task execution needs less time than the specified WCET, scheduling algorithms with unlimited migration might be able to finish a migrating task within the time assigned to the first core, so that migration can be avoided. This, however, is not possible, if the task is determined to migrate at a statically chosen migration point.

In order avoid migration if possible, this thesis introduces dynamic migration decisions to the algorithm presented in [Kla+19]. Dynamic migration decisions are applied at run time, so that the task is still partitioned as before. But instead of migrating at the migration point chosen by the partitioning algorithm, a migration point is chosen dynamically out of the already defined set of potential migration points, depending on the run-time behaviour of each job. If the job needs less time than expected, this approach can allow to delay, or even avoid migration. The algorithm presented in this thesis is independent of the underlying scheduling algorithm, and can applied to any scheduling algorithm that can work with fixed migration points.

To clarify the intent and the benefits of dynamic migration decisions, Figure 2.1 provides an example of different approaches to limit migration in semipartitioned scheduling. In each approach, the migrating task is split into two parts, as shown in Figure 2.1a. In this case, the splitting point is the same for all algorithms, but note that due to the granularity of sections between migration points, approaches using limited migration might have to migrate earlier. The run time behaviour with a run time of two thirds of the estimated WCETs is illustrated for each approach in the remaining subfigures. Figure 2.1b shows unrestricted migration. In this case, the full WCET assigned to the first core is used, before the task migrates. If the overhead is ignored, this leads to the best response time. The overhead for this approach can, however, be high, and impact both schedulability and response times. To mitigate this problem, migration can be limited to a statically determined migration point, as shown in Figure 2.1c. In this scenario, all sections are executed on their statically assigned core, so that the task migrates earlier than necessary, which increases the response time, and makes it impossible to avoid migration, even with faster run times. Figure 2.1d shows the behaviour with dynamic migration decisions. Since an additional section can be executed within the assigned WCET, the task migrates at a later migration point, and migration can be delayed. Note that with a fixed release time of the second partial task, the response time is improved, even without avoiding migration.



(a) partitioned task as scheduled with worst-case run times

(b) migration at an arbitrary point

(c) migration at a statically assigned point

(d) dynamically chosen migration point

Figure 2.1 – Comparison of different approaches to limit migration: in all approaches, the task is split as shown in figure 2.1a. The task is split according to the estimated WCETs, but in this instance only executes for two thirds of its WCET at run time. Figure 2.1b shows migration at an arbitrary point in the code, so that the task migrates after executing for its statically assigned run time. In figure 2.1c, the task migrates at its statically assigned migration point, regardless of actual run times. With dynamic migration decisions, as shown in figure 2.1d, migration can be delayed until the next migration point.

For the dynamic choice of migration points, different approaches will be presented. Each approach can be applied to any semipartitioned scheduling algorithm that works for fixed migration points. For each of these mechanisms, the required overhead will be estimated. Both overhead and response times will be measured for an implementation on a Raspberry Pi v2b, using a port of FreeRTOS as underlying operating system.

ARCHITECTURE 3

In this thesis, different algorithms for dynamic migration decisions will be presented, based on the existing approach using statically determined migration points. All algorithms are designed to execute as much code as possible on the current core, under the limitation, that no core is used longer than planned by the partitioning algorithm. For this, already available information about predefined migration points and split tasks is used.

In order to provide an overview over the already given information, the initial situation is summarized first in this chapter, before the notation for relevant concepts is defined. Before presenting any specific algorithm, the necessary requirements are defined, that must be fulfilled by dynamic migration decisions, in order to prevent deadline misses. After these requirements are defined, a general approach for dynamic migration decisions is outlined, and based on this approach, three different algorithms will be presented.

3.1 Initial Situation

The algorithm presented in this thesis is an extention of the previously described algorithm using fixed migration points. Thus, all information available for fixed-migration scheduling can still be used for dynamic migration decisions. This information includes all potential migration points and the WCETs between them, as well as information about split tasks, such as the assigned cores and the migration points at which the task is supposed to migrate with fixed-migration scheduling.

In the context of this thesis, dynamic migration decisions for any split task τ_i will only use information about the task itself. Information about other tasks, or otherwise available slack time, will not be considered. This allows dynamic migration decisions to work independent of the underlying scheduling algorithm. The presented algorithms can therefore be applied to any semiparitioned scheduling algorithm that works with fixed migration points.

All presented algorithms are designed for hard-deadline task systems. This means, that any schedulable task set will be guaranteed to meet all deadlines with fixed-migration scheduling, and is not allowed to miss any deadlines due to dynamic migration decisions.

Since dynamic migration decisions require additional operations, they will cause some additional overhead. This overhead will be discussed in Chapter 4. In the current chapter, any overhead, caused by dynamic migration decisions or other operations, is assumed to be already included in the given WCETs, and will not be considered further.

3.2 Notation

In this section, the notation for relevant concepts will be defined. This includes concepts for already available information, such as tasks, migration points and the static assignment of sections to split tasks. Furthermore, the notation of time instances and the available budget will be introduced.

3.2.1 Tasks

The scheduled task set is comprised of n tasks and is denoted as $\tau = \{\tau_1, \ldots, \tau_n\}$. For tasks, the 3-parameter task model is used, so that for each task, WCET, deadline and period are given. A task with these parameters is denoted as $\tau_i = (C_i, D_i, T_i)$

3.2.2 Migration Points

Since task migration is only allowed at predefined points, potential migration points are statically identified by the task splitting algorithm presented in [Kla+19]. This algorithm splits task τ_i in p sections, divided by migration points $x_{i,j}$, with $j \in \{0, \ldots, p\}$, where $x_{i,0}$ and $x_{i,p}$ represent the start and the end of τ_i, respectively. Section indices start at 1, so that section j with statically known WCET $c_{i,j}$, is located between migration points $x_{i,j-1}$ and $x_{i,j}$, as depicted in Figure 3.1.

For convenience, if there is no ambiguity about the task in question, the task index is omitted, so that $x_{i,j}$ and $c_{i,j}$ are denoted as x_j and c_j. Also for conveniance, an order is defined on all migration points of a task τ_i, so that

$$x_j \leq x_k \Leftrightarrow j \leq k$$

The WCET between two migration points x_j and x_k with $j \leq k$ is denoted as $WCET\left(x_j, x_k\right)$, with

$$WCET\left(x_j, x_k\right) = \sum_{l=j+1}^{k} c_l$$

and

$$WCET\left(x_j, x_l\right) = WCET\left(x_j, x_k\right) + WCET\left(x_k, x_l\right) \text{ for all } j \leq k \leq l$$

The minimal and maximal section WCETs of task τ_i are denoted as $cMin_i$ and $cMax_i$, respectively. Since, at run time, the largest section might have already been completed, so that its WCET is not relevant anymore, the maximal section length for any section following migration point x_m is defined as

$$cMax_i(m) = \max\left\{c_j \mid m < j\right\}$$

Since fewer sections have to be considered at a later point in the code, $cMax_i(m)$ canot increase with increasing m:

$$m \leq m' \Rightarrow cMax_i(m) \geq cMax_i(m')$$

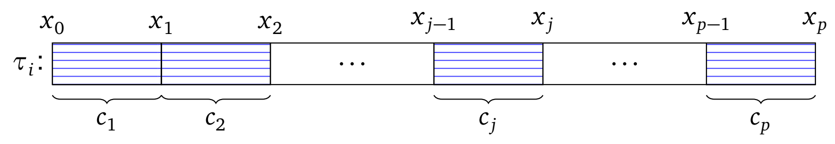


Figure 3.1 – sections of task τ_i: section indices start at 1, so that c_j refers to WCET between x_{j-1} and x_j

While the WCETs are known between migration points, there is no such information available for arbitrary points. At an arbitrary point pos in the code, the WCET until another migration point has to be estimated, using information about the surrounding migration points. For this purpose, the recently reached migration point is referred to as x_{curr}, while the next available migration point is referred to as x_{next}. If the current position coincides with some migration point x_j, then $x_{curr} = x_{next} = x_j$. Otherwise, $x_{next} = x_{curr+1}$. With this information, the WCET from pos until migration point x_j can be estimated by the following inequation:

$$WCET\left(x_{next}, x_j\right) \leq WCET\left(pos, x_j\right) \leq WCET\left(x_{curr}, x_j\right)$$

3.2.3 Split Tasks

If a task τ_i is chosen as migrating task by the partitioning algo, it is split in q partial tasks $\tau_i^l = (C_i^l, D_i^l, T_i)$, with $l \in \{1, \ldots, q\}$. These partial tasks are scheduled on their assigned cores as separate tasks, according to their task parameters.

When the partitioning algorithm splits a migrating task, each partial task τ_i^l is assigned a range of sections between two migration points. The start and end points of this range are referred to as $x_{start(l)}$ and $x_{end(l)}$ respectively, with

$$\begin{aligned} start(1) &= 0 \\ end(q) &= p \\ start(l+1) &= end(l) \text{ for all } l \in \{1, \ldots, q-1\} \end{aligned}$$

Since the specified WCET for each partial task must be sufficiently large for all assigned sections to fit in, it is known that:

$$C_i^l \geq WCET\left(x_{start(l)}, x_{end(l)}\right)$$

While C_i can be larger than required by the assigned sections, it is assumed in the remaining thesis, that with full section WCETs, no additional section fits in.

Note that with dynamic migration decisions, sections do not necessarily run on their assigned core. In this case, τ_i^l refers to all sections that run on the lth assigned core of τ_i, regardless of static section assignment.

3.2.4 Execution Time

When a section is run on the current core, dynamic migration decisions must ensure that the execution time of the current partial job does not exceed the WCET of the current partial task. For

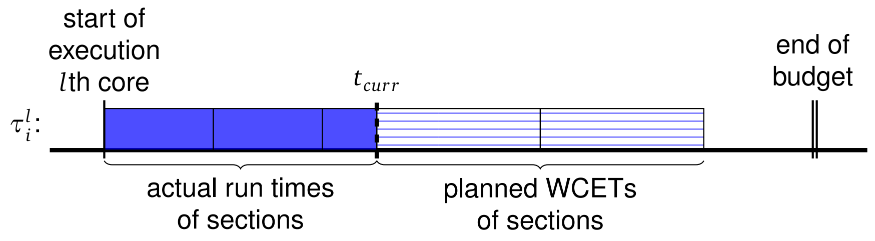

start of execution lth core
t_{curr}
end of budget
τ_i^l:
actual run times of sections
planned WCETs of sections

Figure 3.2 – depiction of timeline for partial task τ_i^l: time coresponds to execution time of partial task, with transparent preemptions. For all finished sections the actual execution time can be used, while the time for unfinished sections has to be estimated by the given WCETs.

compliance with this condition, only the execution time of the current partial job is relevant, as opposed to the time that has passed since task release. Thus, any time value considered in dynamic migration decisions corresponds to the execution time of the current partial job. Interruptions of the task, for example by preemptions, are transparent for this view.

According to this view on time, when a time instant t is considered, t refers to the time instant at which the current partial job has been executing for t time units. The position in the code at time t is referred to as $pos(t)$, while the time instant at which migration point x_j is reached is denoted as $t(x_j)$. When a migration decision is made, t_{curr} refers to the current time of the decision. If $pos(t_{curr})$ does not match any migration point, then $pos(t_{curr})$ can be estimated by $x_{curr} < pos(t_{curr}) < x_{next}$, so that the remaining WCET until any given migration point can be estimated accordingly.

3.2.5 Budgets and Reachability

In order to ensure that no task runs longer than allowed, each partial task τ_i^l is assigned a budget corresponding to its assigned C_i^l. The remaining budget of τ_i^l at time instant t is referred to as $B_i^l(t)$, with

$$\begin{aligned} B_i^l(0) &= C_i^l \\ B_i^l(t) &= B_i^l(0) - t \\ &= C_i^l - t \end{aligned}$$

If at some instant t, a migration point x_j can be reached within the remaining budget, even if all remaining sections run with worst-case run times, x_j is called *reachable at t*. Reachability can be expressed as follows:

$$x_j \text{ is reachable at } t \Leftrightarrow WCET\left(pos(t), x_j\right) \leq B_i^l(t)$$

For reachability, some properties can be derived:

- if x_j is reachable at t, then any $x_k \leq x_j$ is also reachable at t
- if x_j is reachable at t, then x_j is still reachable at any later time instant $t' \geq t$
- if x_j is unreachable at t, it might become reachable at some later time instant $t' \geq t$

While the first statement is obvious, the latter two statement can be shown by comparing the changes of the remaining budget with the changes of the remaining WCET. Between $pos(t)$ and $pos(t')$, the remaining budget decreases by the actual execution time between these points, while the remaining WCET decreases by $WCET\,(pos(t), pos(t'))$. Since the execution time never exceeds the WCET, the remaining WCET decreases at least as fast as the remaining budget, so that all reachable migration points remain reachable later in time, and previously unreachable migration points might become reachable.

Further statements about reachability can be derived from the assignment of sections to partial tasks. Since all sections assigned to τ_i^l are known to fit into C_i^l, but no additional sections fit in with worst-case run times, the following statements hold:

- if τ_i^l starts at any $x_j \geq x_{start(l)}$, then $x_{end(l)}$ is reachable at $t = 0$.
- if τ_i^l starts at any $x_j < x_{start(l)}$, then $x_{end(l)}$ is not reachable at $t = 0$.

Whenever the execution of a task is depicted in the remaining thesis, timing information is illustrated as shown in Figure 3.2. As in this figure, start and end point of the assigned budget, as well as the current time will be marked. Before the current time, the actual run times are depcited, while after the current time instant, only WCETs are available.

3.3 Requirements for dynamic Scheduling

As discussed in previous sections, the algorithm for dynamic migration decisions can be added to any semipart fixed-migration scheduling algo, in order to delay migrations. This requires, that, regardless of the underlying scheduling algorithm, dynamic migration decisions do not lead to any deadline misses, for any schedulable task set. The compliance with this requirement will be proved for each scheduling algorithm presented later in this chapter. In order to simplify these proofs, some sufficient conditions for this will be defined in this section.

In order to derive these conditions, the desired property is specified first.

Definition 1. *Let M be an algorithm for dynamic migration decisions. M is* schedulability preserving*, iff for any semipartitioned scheduling algorithm A that uses fixed migration points, and any A-schedulable task set, M can be applied without causing any deadline misses.*

From this definition, conditions can be derived that can be used to verify that a given algorithm preserves schedulability.

Theorem 1. *Let M be an algorithm for dynamic migration decsions. M is schedulability preserving, if for any semipartitioned fixed-migration scheduling algorithm A, any A-schedulable task set τ, and any migrating task $\tau_i \in \tau$ with p migration points, the following conditions hold when M is applied:*

1. *all jobs of all τ_i^l execute within their budget*
2. *for each job of τ_i, some partial job reaches x_p*

If the migration decisions of M for τ_i are only based on information about τ_i and ignore other information, such as run times of other tasks or otherwise available slack, then the above conditions are not only sufficient, but also necessary.

Proof. First, the sufficiency of the defined condition is shown, by assuming that the specified conditions hold. Condition 1 implies that all jobs of each τ_i^l execute within their given budget, as do all other tasks. Hence, the A-schedulability of τ guarantees that every job of each $\tau_j \in \tau$ can execute for C_j time units before its deadline, which is sufficient for all non-migrating tasks to meet their deadlines. Condition 2 ensures, that, for all jobs of all migrating tasks τ_i, some partial job reaches the end of τ_i. According to condition 1, this partial job is completed within its budget, and is therefore reached before the partial deadline. Since the deadline of each partial job is not later than the deadline of the entire job, all jobs of all tasks meet their deadlines.

In order to show the necessity of the above conditions, it suffices to show that there exists a case, in which deadlines will be missed without one of the above conditions. Without Condition 1, a job of τ_i^l will need more time than planned. If all other tasks on this core execute with worst-case run times, and there is no otherwise available slack, then some deadline will be missed on this core. Without the second condition, some job of τ_i has not been completed, before the last partial job ends. Thus, this job cannot be finished, before the next job of τ_i is released, which will lead to a deadline miss, if the deadline of τ_i is not longer than its period. □

From this, minimal required end points for all partial tasks can be derived:

Theorem 2. *Let M be an algorithm for dynamic migration decisions that fulfills the above conditions. Let τ be a task set that is schedulable with the given algorithm. Then, for each migrating task $\tau_i \in \tau$, each partial task τ_i^l reaches $x_{end(l)}$.*

Proof. Let τ_i^{l+1} be any partial task that is required to reach $x_{end(l+1)}$. According to the properties of reachability, this can only be guaranteed, if $x_{end(l+1)}$ is reachable at $t = 0$, which is requires τ_i^{l+1} to start at some $x_j \geq x_{start(l+1)}$. Since $start(l+1) = end(l)$, each partial task τ_i^{l+1} is only guaranteed to reach $x_{end(l+1)}$, if the previous partial task τ_i^l reaches $x_{end(l)}$.

Since Condition 2 requires τ_i^q to reach $x_p = x_{end(q)}$, the rest follows by induction. □

Since the above condition implies Condition 2, two conditions can be defined to simplify the verification of dynamic migration algorithms. An algorithm for dynamic migration decisions that only considers information about the currently processed split task is schedulability preserving, if the following conditions are fulfilled by each split task τ_i:

(S1) all partial tasks τ_i^l end or migrate within their budget

(S2) all partial tasks τ_i^l reach $x_{end(l)}$

Aside from proving that a given algorithm preserves schedulability, Condition (S1) can also be used in the construction of a dynamic migration algorithm itself. If both conditions can be verified later, all dynamic migration decisions regarding partial task τ_i^l can assume $x_{end(l)}$ to be reachable without further considerations.

3.4 Dynamic Migration Decisions

As stated before, the goal of dynamic migration decisions is to delay migration as far as possible. In order to do so, information about reachability is used. Since more optimistic estimations of reachability are possible when more sections have been completed, migration decisions are also delayed as far as possible. In the remaining section, different approaches to delay migration decisions will be presented. Each presented algorithm will be verified using the conditions defined in the previous section.

3.4.1 A Simple Approach

A relatively simple algorithm can be defined, that decides at each migration point whether to migrate. At each migration point, the WCET of the next section is compared to the remaining budget, and the decision is made accordingly.

It can be easily shown that this approach preserves schedulability. Since the task migrates only, if the next migration point is not reachable, each partial task τ_i^l that starts at some migration point $x_j \geq x_{start(l)}$ will reach $x_{end(l)}$. With τ_i^1 starting at x_0, Condition (S1) is fulfilled. Condition (S2) also holds, since at each migration point, the task is only continued on the current core, if the next migration point is identified as reachable.

The drawback of this approach is its potentially high overhead. At each migration point, the WCET of the next section is compared with the remaining budget, which is kept updated at each timer tick. In a system that separates address spaces of operating system and application, the remaining budget is part of the operating system, so that at each migration point, access to kernel data is necessary. This can be expensive, especially for tasks with a high number of migration points.

3.4.2 Skip Reachability Checks by using Evaluation Points

In order to reduce the overhead, all further approaches will try to reduce the the number of reachability checks. Instead of inserting reachability checks at each migration point, reachability will be evaluated only at a few selected points, which do not necessarily coincide with migration points. These points will be referred to as *evaluation points*. Each migration point that has been neither identified as evaluation point nor chosen for task migration, will be skipped.

Note that the skipping of migration points still requires additional overhead at each migration point, in order to ensure that no evaluation point is skipped. This requires a comparison of the current migration point to the next migration point with a special function. If this comparison is done by the application, the indices of these migration points are only accessed by the operating system, when an evaluation point is reached. This means, that these indices can be kept in the address space of the application without causing a significant overhead, so that migration points can be skipped without accessing kernel data.

While this approach can reduce the effort for most migration points, evaluation points can still require a higher overhead. Thus, all algorithms presented in the following are designed to keep the number of evaluation points as low as possible. In order to use as much run-time information as possible at each evaluation point, evaluation points are set as late as possible. From this, a general approach of algorithms using evaluation points can be derived. Initially, an evaluation point is defined as the latest possible point from which migration decisions can be made without exceeding the budget. When this evaluation point is reached, more run-time information is available, and a further delay of migration decisions might be possible. If migration decisions can be delayed, another evaluation point is defined, and the task is resumed. Otherwise, a migration point is chosen, at which the task will migrate.

There are different ways to define evaluation points. In the remaining chapter, an algorithm will be presented for each of the following definitions:

(A1) Evaluation points are defined by a position in the code

(A2) Evaluation points are defined by a point in execution time

(A3) Evaluation points are defined by either a position in the code or a point in execution time

Each presented algorithm will be illustrated by applying it to example task τ_i, which is depicted in 3.3. This task is divided into 12 sections. Each section has a WCET of 6, except for section 9 with $c_9 = 10$, and section 10 with WCET $c_{10} = 8$. At run time, each section will need exactly one half of its assigned WCET. Dynamic migration decisions will be made for the first partial task τ_i^1, which has an assigned budget of $B_i^1(0) = 40$. The partitioning algorithm has split the task at $x_{end(1)} = x_6$.

3.4.3 Evaluation Points as Position in Code

This algorithm is an improvement of the previously presented simple approach. As in the simple approach, all reachability checks are done at migration points, but in this case, only a few migration points are selected as evaluation points, so that most checks will be skipped.

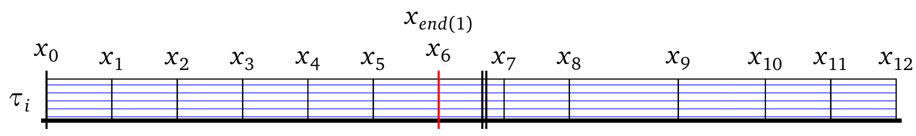


Figure 3.3 – Used example task for the remaining chapter. Aside from sections 9 and 10 with a WCET of 10 and 8 timer ticks, respectively, all sections have a WCET of 6. The task is split at x_6, with a budget of 40 assigned to the first partial task.

An evaluation point x_{eval} is defined as the last migration point, at which migration decisions can be made without exceeding the budget. Since immediate migration is possible at each evaluation point, the last reachable migration point is chosen. This leads to the following definition of evaluation points:

$$x_{eval} := \max\left\{x_j \mid WCET\left(x_{curr}, x_j\right) \leq B_i^l(t_{curr})\right\}$$

After an evalation point is chosen, the actual task is resumed. When the chosen evaluation point is reached, the sections in the mean time might have been executed faster than expected, so that further migration points might be reachable within the remaining budget. In this case, another evaluation point is defined. Otherwise, the task migrates immediately.

3.4.3.1 Example

In Figure 3.4, this algorithm is applied to the previously defined example task. Initially, $x_{end(1)} = x_6$ is known to be reachable, while x_7 is estimated to be reached at $t = 42$, which is not within the given budget. Thus, x_6 is chosen as the first evaluation point. With run times of one half of the estimated WCETs, x_6 is reached at $t = 18$. At this time, x_9 is estimated to be reached at $t = 40$, while x_{10} is unreachable, so that x_{eval} is set to x_9. When x_9 is reached at $t = 29$, x_{10} is identified as the last reachable migration point, and therefore chosen as x_{eval}, while at x_{10}, migration decisions can still be delayed until x_{11}. At $t = 36$, x_{11} is reached. At this point, 4 time units of the budget remain, while the next section has a WCET of $c_{12} = 6$, which makes x_{12} unreachable. This means, that migration decisions cannot be delayed until the next migration point, and the current job migrates immediately.

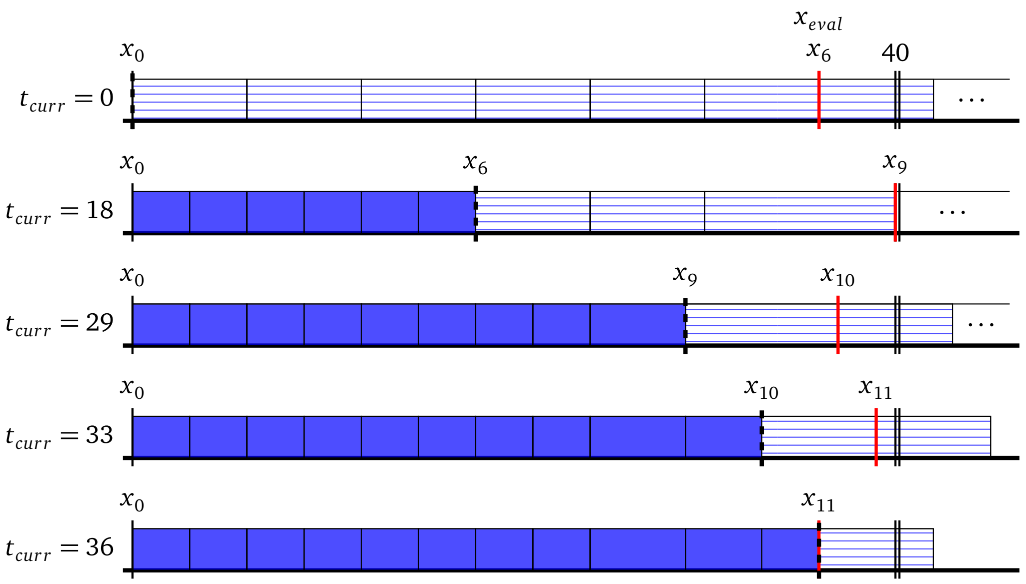


Figure 3.4 – Given example task, executed with Algorithm (A1). x_{eval} is initially set to x_6. With actual runtimes, x_6 is reached earlier than expected, so that migration decisions can be delayed until x_9. After reaching x_9, further evaluation points are set at x_{10} and x_{11}, until at x_{11}, migration decisions cannot be delayed further and the task migrates at x_{11}.

3.4.3.2 Schedulability

In order to be schedulability preserving, the algorithm presented above has to fulfill the conditions defined in the previous section. This is shown in the following two theorems.

Theorem 3. *Algorithm (A1) fulfills (S1), i.e. no budget is exceeded.*

Proof. A migrating task can only migrate or end at a migration point that has been chosen as x_{eval}. x_{eval} is reachable by definition. □

Theorem 4. *Algorithm (A1) fulfills (S2), i.e. each τ_i^l reaches $x_{end(l)}$.*

Proof. Let τ_i^l be a partial task that starts at some $x_j \geq x_{start(l)}$. According to the properties of reachability, this means that $x_{end(l)}$ is always reachable by τ_i^l. Since x_{eval} is defined as the maximal reachable migration point, no evaluation point of τ_i^l can be smaller than $x_{end(l)}$. Thus, τ_i^l cannot migrate, before $x_{end(l)}$ is reached, and τ_i^{l+1} starts at some $x_j \geq x_{start(l+1)}$.

With τ_i^1 starting at $x_0 = x_{start(1)}$, it follows by induction that each partial task τ_i^l starts at some $x_j \geq x_{end(l)}$ and reaches $x_{end(l)}$. □

3.4.3.3 Calculation of x_{eval}

The above algorithm requires a search algorithm to identify the next evaluation point. Since x_{eval} is defined as the maximal reachable migration point, and both x_{curr} and $x_{end(l)}$ are known to be reachable, any search algorithm can start to search at $\max(x_{curr}, x_{end(l)})$, without considering previous migration points. In order to select the next evaluation point out of the remaining candidates, various algorithms can be used, three of which will be discussed here.

A relatively simple solution is to use linear search. Linear search tests all migration points from the defined starting point until the first unreachable migration point, and has an effort linear to all reachable migration points. While this is beneficial in scenarios with a small remaining budget, it can lead to a relatively high effort, if many additional migration points are reachable.

Since all migration points preceeding x_{eval} are reachable, and all succeeding migration points are unreachable, search algorithms for sorted lists can be applied, such as binary seach. The effort needed for binary seach is logarithmic to the number of all migration points following the defined starting point, regardless of reachability. This can lead to a higher overhead with many remaining unreachable migration points. The worst-case effort is, however, still logarithmic.

Both linear and binary search only consider the reachability of migration points, but do not use additional available information, such as the upper bound of section lengths, given by $cMax_i$. The upper bound of section lengths can be used, in order to estimate the number of additional reachable migration points. With a given budget B, and a known maximal section length $cMax$, at least $\left\lfloor \frac{B}{cMax} \right\rfloor$ further migration points are reachable. This estimation is, however, not exact and can leave some reachable migration points undetected. In order to improve the result, the estimation can be applied again. After a migration point x_j has been identified as reachable, the remaining budget after x_j can be estimated by $B - WCET\left(x_{curr}, x_j\right)$. With the remaining budget, additional reachable migration points can be identified.

Based on this, a third search algorithm is presented. This algorithm identifies reachable migration points $x_{r(n)}$, using the following recurrent definition:

$$\begin{aligned} r(0) &= \max(curr, end(l)) \\ r(n+1) &= r(n) + step(n+1) \\ step(n+1) &= \left\lfloor \frac{B_i^l(t_{curr}) - WCET\left(x_{curr}, x_{r(n)}\right)}{cMax_i\left(r(n)\right)} \right\rfloor \end{aligned}$$

Since the number of reachable migration points is limited and $r(n) \in \mathbb{N}$, this recurrence reaches its fixpoint r in a finite number of steps. The number of steps is logarithmic, depending on the ratio of average and maximal section lengths and the number of reachable migration points, and will be further discussed in Chapter 4.

Even after a fixpoint is reached, there might sill be some unidentified reachable migration points. The upper bound of this number can be derived from the $step$-function. When $step(n+1) = 0$, then the estimated remaining budget at $x_{r(n)}$ is less than $cMax_i$. Within this budget, less than $\frac{cMax_i}{cMin_i}$ can be reachable, so that from x_j, at most $\left\lceil \frac{cMax_i}{cMin_i} \right\rceil - 1$ additional migration points are reachable, which are identified via linear search.

With similar section lengths, the estimations used by this algorithm are more accurate, so that less loop iterations are needed. Compared to binary search, this algorithm considers only reachable migration points, which is beneficial in scenarios with longer run times. The disadvantage of this algorithm is the more complicated implementation. Additional comparisons and expensive operations such as integer division will lead to a higher run time for each loop iteration compared to linear or binary search. Furthermore, the estimation gets less precise with more uneven section lengths, so that with a large ratio of maximal and average section length, even more loop iterations than for binary search might be needed.

Since binary search and the previously described algorithm both have advantages in specific situations, both will be considered as possible search algorithms and evaluated in further chapters. Due to the linear effort, linear search will not be further considered for this approach. In order to reduce run time, all presented search algorithms need to calculate the WCET between arbitrary migration points in constant time. This can be done by calculating the difference of cumulative WCETs.

3.4.3.4 Discussion of Approach (A1)

The described approach defines the last reachable migration point as evaluation point and migrates the task only, if no further section can be fit in the remaining budget. Since migration is avoided, if any further migration point is reachable, the budget is used optimally, with regards to the given constraints and the given information.

The disadvantage of this approach is the potentially high number of recalculations, as shown in the previous example. With faster run times, more recalculations are needed, which leads to a higher overhead.

3.4.4 Evaluation Points as Point in Execution Time

This approach tries to reduce the number of recalculations by defining evaluation points as points in execution time, rather than as points in the code. In the previous approach, recalculations are needed, if the remaining budget at an evaluation point is larger than expected. With an evaluation point as point in time, the remaining budget is the same as expected, regardless of actual run times.

In this approach, an evaluation point t_{eval} is defined as the latest possible point in time, at which a migration decision can be made without exceeding the remaining budget. When the evaluation point is reached, it can be either delayed, or a migration point x_{migr} is chosen, at which the task will migrate without further calculations.

3.4.4.1 Calculation of t_{eval}

As specified, t_{eval} is defined as the latest point in execution time, at which migration decisions can be made. This means, that at t_{eval}, at least one further migration point must be reachable, as illustrated in Figure 3.5. For the next migration point to be reachable, the remaining budget must be at least as large as the WCET of the remaining current section. The budget required for this is not known exactly, since at the defintion time of t_{eval}, neither the current section at t_{eval}, nor the exact position within this section are known. Thus, the minimal remaining budget is estimated by the full section WCET of the largest relevant section. A section is considered relevant in this context, if it ends at a potentially unreachable migration point. Given that at each point, both x_{next} and $x_{end(l)}$ are known to be reachable, t_{eval} for partial task τ_i^l can be calculated as follows:

$$\begin{aligned} t_{eval} &:= B_i^l(0) - cMax_i(m) \\ m &:= \max(next, end(l)) \end{aligned}$$

Note that instead of the above estimation, m could also be calculated as the exact last reachable migration point by using any search algorithm described for the previous approach. But assuming that the differences between section lengths are relatively small, and considering that t_{eval} can still be recalculated at a later point, the above heuristic is used instead, since it requires only a constant effort.

When t_{eval} is calculated initially for a partial task, $cMax_i(m)$ might be larger than the available budget. In this case, x_{migr} is set immediately. Otherwise, after t_{eval} is set, the task is executed until the defined evaluation point is reached. At t_{eval}, the largest section might have been completed in the meantime, so that $cMax_i(m)$ has decreased. In this case, a new t_{eval} is set, and the migration decision is delayed further. If no such delay is possible, a migration point x_{migr} is chosen, at which the task will migrate.

3.4.4.2 Calculation of x_{migr}

When no further delay of t_{eval} is possible, a migration point x_{migr} is chosen, at which the task will migrate. Ideally, x_{migr} is the last reachable migration point at t_{eval}. In most cases, however, $pos(x_{eval})$ is somewhere between two migration points, so that reachability can only be estimated.

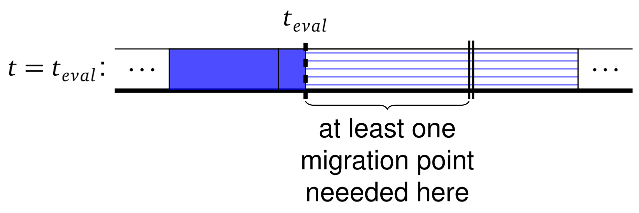


Figure 3.5 – At any t_{eval}, at least one migration point must be reachable. This means, that, regardless of the position in the code, the rest of the current section must fit in the remaining budget.

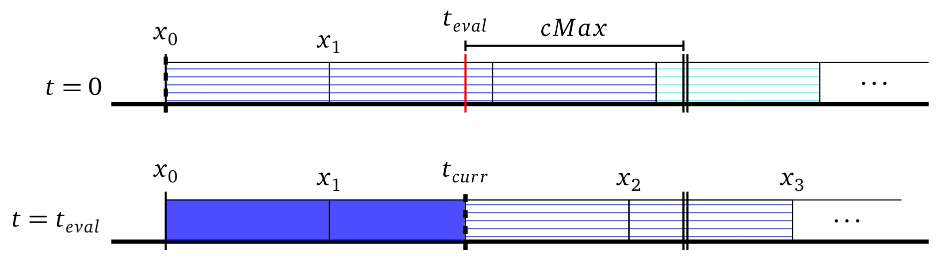


Figure 3.6 – Problems of reachability calculations at t_{eval}: Since, at t_{eval}, the exact position in section 2 is unknown, the time until x_2 has to be estimated by using the full section WCET. This leads to a pessimistic estimation that falsely identifies x_3 as unreachable.

As shown in Figure 3.6, this can lead to the false identification of migration points, including $x_{end(l)}$, as unreachable. Thus, instead of calculating the reachability of available migration points from the current position, a heuristic is used, based on the knowledge about reachability. With this heuristic, x_{migr} is defined as:

$$x_{migr} := \max\left(x_{next}, x_{end(l)}\right)$$

When x_{migr} is reached, the current job migrates without further calculations.

3.4.4.3 Example

This algorithm is applied to the example task, as shown in Figure 3.7. The maximal section length following $x_{end(1)} = x_6$ is given by $c_9 = 10$. With a budget of 40, t_{eval} is set to $40 - 10 = 30$. Since this is more than the current time, migration decisions can be delayed, and the task is executed until $t = 30$. At $t = 30$, the evaluation point is reached. Section 9 has been completed at this point, so that the largest remaining section is section 10, with $c_{10} = 8$. This allows migration decisions to be delayed until $t = 32$. At $t = 32$, section 10 is still running. Since its remaining run time is estimated by its full WCET, migration decisions cannot be delayed further. The algorithm chooses a migration point $x_{migr} = \max(x_{next}, x_{end(1)}) = \max(x_{10}, x_6) = x_{10}$ and returns to the task. At x_{10}, the task migrates.

Note that at the time of migration, the next migration point could be identified as reachable, and by ignoring this, the algorithm leads to a non-optimal use of the given budget in this case.

3.4.4.4 Schedulability

If migration decisions are made according to this algorithm, schedulability is preserved, which will be shown here. In order to prove that no budget is exceeded, the following lemma will be shown first.

Lemma 1. *At each calculation of t_{eval}, x_{next} is reachable.*

Proof. Let $t_{eval(n)}$ be the nth evaluation point of the partial task τ_i^l, so that $t_{eval(0)}$ represents the start of τ_i^l. The lemma can be shown by induction over n:

At $t_{eval(0)}$, the current partial task has just migrated to the current core. Since migration is only possible at migration points, the current position is at some migration point x_j, so that $x_{curr(0)} = x_{next(0)} = x_j$, which means that $x_{next(0)}$ is reachable.

For evaluation point $t_{eval(n+1)}$, different cases can be considered. If $x_{end(l)}$ has not been reached yet, then $x_{next(n+1)}$ not larger then $x_{end(l)}$, and therefore reachable. If no section has been completed

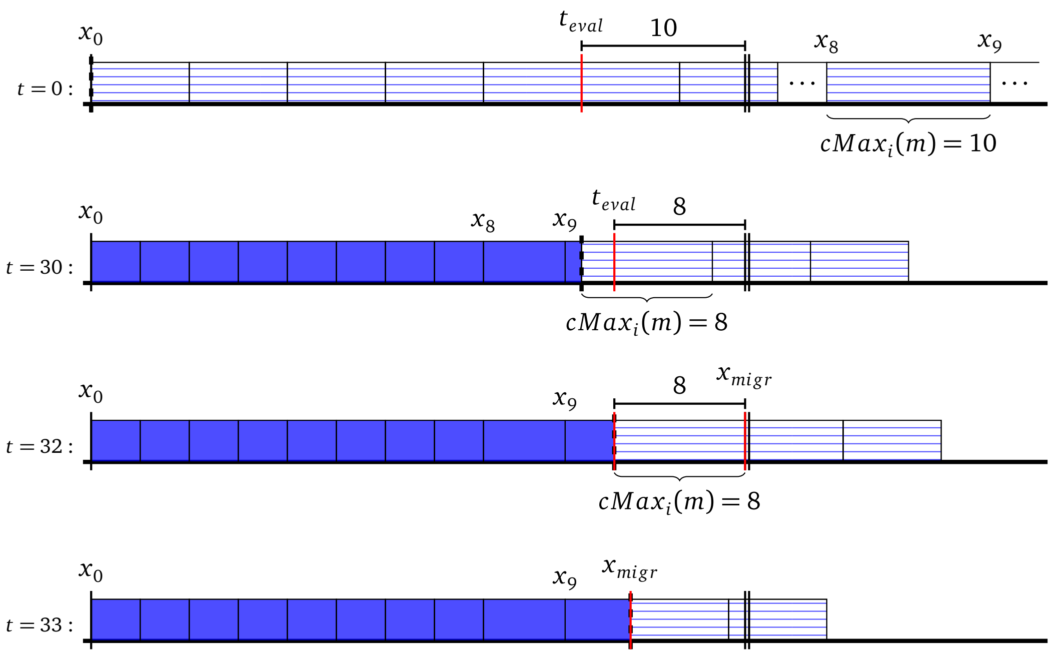


Figure 3.7 – Example task, executed with Algorith (A3). With a budget of 40 and a maximal section length $c_9 = 10$, t_{eval} is initially set to 30. When t_{eval} is reached, the largest section has been completed, so that t_{eval} can be delayed until $t = 32$. At $t = 32$, no further delay of migration decisions is possible, and x_{migr} is set to $x_{next} = x_{10}$, where the task migrates.

since the last evaluation point, then $x_{next(n+1)}$ is equal to $x_{next(n)}$, which is reachable by induction hypothesis. This leaves the case, in which $x_{next(n+1)}$ is larger than both $x_{next(n)}$ and $x_{end(l)}$. In this case, the reachability of $x_{next(n+1)}$ can be shown by calculating the the remaining budget at $t_{eval(n+1)}$:

$$\begin{aligned} B_i^l(t_{eval(n+1)}) &= C_i^l - t_{eval(n+1)} \\ &= C_i^l - \left(C_i^l - cMax_i\left(m(n)\right)\right) \\ &= cMax_i\left(m(n)\right) \\ m(n) &= \max\left(next(n), end(l)\right) \\ &\leq next(n+1) - 1 \end{aligned}$$

With this estimation of $m(n)$, the remaining budget can be estimated:

$$\begin{aligned} B_i^l(t_{eval(n+1)}) &= cMax_i\left(m(n)\right) \\ &\geq cMax_i\left(next(n+1) - 1\right) \\ &\geq c_{next(n+1)} = WCET\left(x_{curr(n+1)}, x_{next(n+1)}\right) \end{aligned}$$

With a sufficiently large budget for the full section WCET between $x_{curr(n+1)}$ and $x_{next(n+1)}$, $x_{next(n+1)}$ is reachable at $t_{eval(n+1)}$. $\square$

From this, Condition (S1) can be derived.

Theorem 5. *Algorithm (A2) fulfills (S1), i.e. no budget is exceeded.*

Proof. No partial task can execute longer than until x_{migr} is reached. x_{migr} is defined at some evaluation point, with $x_{migr} = \max(x_{next}, x_{end(l)})$. Since $x_{end(l)}$ is always reachable, and, according to the previous lemma, x_{next} is reachable at t_{eval}, x_{migr} is reached within the given budget. □

Theorem 6. *Algorithm (A2) fulfills (S2), i.e. each τ_i^l reaches $x_{end(l)}$.*

Proof. The task cannot migrate, before x_{migr} is reached, which is at least $x_{end(l)}$ by definition. □

3.4.4.5 Discussion

This algorithm defines evaluation points as points in execution time. Since the remaining budget at evaluation points does not depend on run-time behaviour, this approach is beneficial, if the sections of the given task run faster than expected. Evaluation points are recalculated, if the section with the maximal length has completed in the meantime, but compared with the first approach, recalculations are less likely.

The disadvantage of this approach is that not all reachable migration points are identified as such, so that in some cases, the task migrates earlier than necessary, as shown in the example task. This problem is most likely to occur with uneven section WCETs, and large sections at the end of the task. With the already given mechanism for identifying potential migration points, large differences between section lengths are, however, unlikely, since the algorithm used for this aims for a relatively even distribution of migration points, in order to simplify task partitioning [Kla+19].

3.4.5 Combination of both approaches

When comparing the previous algorithms, Algorithm (A2) needs less evaluation points at the start of the task, while Algorithm (A1) is more exact at the end of the task. In order to combine the advantages of both approaches, the following algorithm starts as in Approach (A2), and continues as in Approach (A1).

The first evaluation point t_{eval} of a partial task is defined as a point in time, as in Approach (A2). When t_{eval} is reached, x_{eval} is defined, using the definition of x_{migr}. At x_{eval}, the algorithm proceeds as Approach (A1). Note that x_{eval} is set at the first t_{eval}, instead of trying to delay t_{eval}. While a recalculation of t_{eval} is possible, this would allow a delay in the size of the difference of the maximal section lengths, so that in most cases, more time can be gained by recalculating x_{eval}.

3.4.5.1 Example

The application of Algorithm (A3) to the example task is shown in Figure 3.8. As in the previous approach, the first evaluation point t_{eval} is set to 30. At $t = 30$, x_{eval} is set, without trying to delay t_{eval}. When x_{eval} is calculated, the same definition as for x_{migr} is used, so that x_{eval} is set to $\max(x_{next}, x_{end(l)}) = \max(x_6, x_{10}) = x_{10}$. When x_{10} is reached, the next migration point is identified as reachable, and the next evaluation point is set to x_{11}. At x_{11}, no further migration point is reachable, and the task migrates.

In this example, less evaluation points are needed than for Algorithm (A1), and compared to (A2), migration can be delayed until x_{11}, instead of migrating at x_{10}.

3.4.5.2 Schedulability

Like the previous approaches, this algorithm is schedulability preserving. This can be shown by using results of the previous proofs.

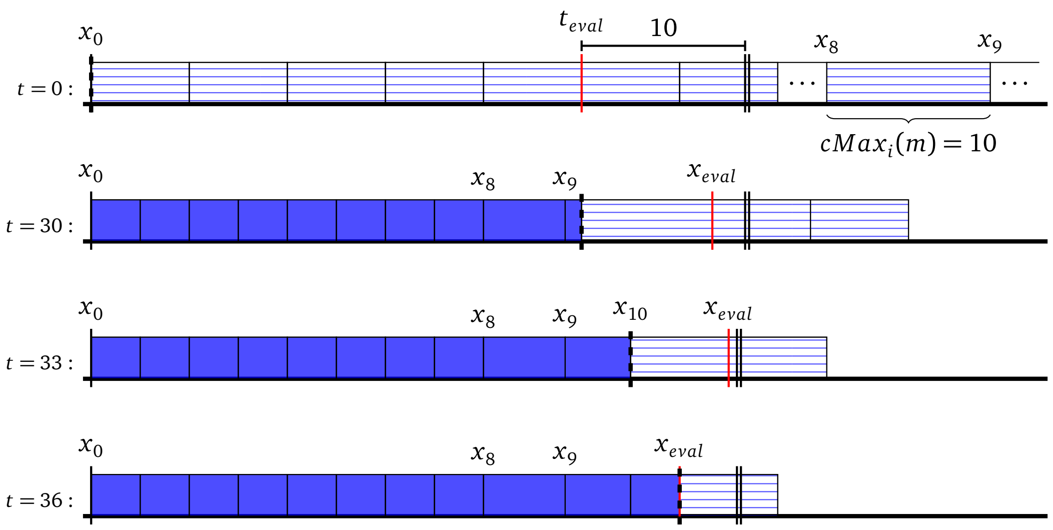


Figure 3.8 – Example task, executed with Algorithm (A3). With a given budget of 40 time units, and a maximal section length of 10 time units, t_{eval} is initialized with 30. When t_{eval} is reached, x_{eval} is set to $x_{next} = x_{10}$. At x_{10}, migration point x_{11} is identified as reachable and is chosen as next evaluation point. At x_{11}, no further migration points are reachable, and the task migrates.

Theorem 7. *Algorithm (A3) fulfills (S1), i.e. no budget is exceeded.*

Proof. The first x_{eval} is defined as $\max(x_{next}, x_{end(l)})$. Since $x_{end(l)}$ is always reachable, and x_{next} is reachable at t_{eval}, as shown in the previous lemma, the first x_{eval} is always reached within the given budget. Further recalculations of x_{eval} ensure, that each new x_{eval} is also reachable, as discussed for Algorithm (A1). □

Theorem 8. *Algorithm (A2) fulfills (S2), i.e. each τ_i^l reaches $x_{end(l)}$.*

Proof. As shown for approach (A2), the initial x_{eval} is at least $x_{end(l)}$. Since migration is not possible before x_{eval}, Condition (S2) is fulfilled. □

3.4.5.3 Search Algorithms for x_{eval}

When x_{eval} is reached, the next evaluation point is calculated as in Algorithm (A1), and the same search algorithms can be used. In this case, however, a different situation in regards to reachability has to be considered, so that different search algorithms are suitable for this approach. Due to its definition, the remaining budget at t_{eval} is at most $cMax_i$. As shown in Figure 3.9, this bound

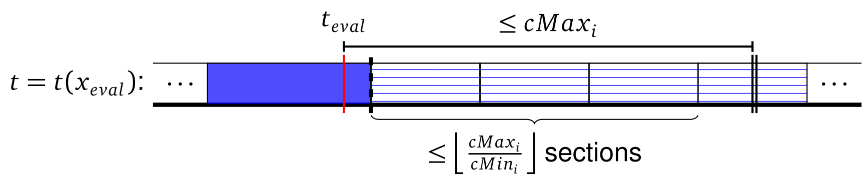

t_{eval}
$\leq cMax_i$
$t = t(x_{eval})$:
$\leq \left\lfloor \frac{cMax_i}{cMin_i} \right\rfloor$ sections

Figure 3.9 – Additionally reachable sections at the first x_{eval} in the combined approach. The number of sections is bounded by the ratio of maximal and minimal section length.

limits the number of reachable migration points to $\left\lfloor \frac{cMax_i}{cMin_i} \right\rfloor$. This limit impacts the suitability of the presented search algorithms.

Linear search tests the reachability of migration points only, until the first unreachable migration point is identified. This limits the number of required loop iterations to $\left\lfloor \frac{cMax_i}{cMin_i} \right\rfloor + 1$, which is assumed to be a relatively small number.

The effort needed for binary search is logarithmic to the number of all remaining migration points, regardless of reachability. With a low variance in section lengths, and a sufficiently high number of unreached migration points, binary search will need more loop iterations than linear search.

With a remaining budget of at most $cMax_i$, the estimation used in the third search algorithm is unlikely to identify any further reachable migration points, and is therefore unsuitable for the given situation.

Based on these considerations, linear search is used as search algorithm for Algorithm (A3).

3.4.5.4 Discussion

The combined approach starts with an evaluation point defined by a point in execution time, and proceeds with evaluation points defined as last reachable migration points. While the first evaluation point makes it possible to reduce the number of recalculations with short section run times, the remaining evaluation points ensure the identification of all reachable migration points. Since in this approach, the number of reachable migration points at each x_{eval} is limited and relatively small, evaluation points will likely cause less overhead than in Algorithm (A1).

On the downside, elements of both previous algorithms need to be implemented for this approach, and overhead for both kinds of evaluation points needs to be considered.

4 ANALYSIS

The goal of dynamic migration decisions is to execute as much code as possible in each partial task, without impacting the schedulability of the task set. In this chapter, the presented algorithms will be evaluated with regards to this goal. The first section will discuss, how much code can be executed on each core, before the task migrates.

The schedulability has already been discussed in the previous chapter, under the assumption that all overhead is already included in the section WCETs. When tasks are scheduled using dynamic migration decisions, the overhead has to be considered when the budget is allocated for each partial task, as well as when dynamic migration decisions are made, and can therefore impact both schedulability and budget usage. The additional overhead caused by dynamic migration decisions will be analyzed in the second section of this chapter.

4.1 Usage of Budget

In order to avoid migration and reduce response times, each algorithm tries to use the budget assigned to the current partial task as efficiently as possible. Compared to semipartitioned scheduling with unrestricted migration, in which the full budget is always usable, the granularity of the given sections will lead to some unused budget in most cases. In the remaining section, the maximal amount of unused budget before migration will be discussed for each presented algorithm.

When the next migration point is unreachable, even an optimal algorithm needs to migrate immediately, and will thus leave some budget unused in most cases. The amount of unused budget is, however limited. In an optimal algorithm, a split task will only migrate at migration point x_j, if x_{j+1} is unreachable, i.e. if the remaining budget at x_j is smaller than the next section WCET c_{j+1}. Since the simple approach, as well as Algorithms (A1) and (A3) always test the reachability of the next migration point immediately before migrating, these algorithms are optimal in respect of budget use.

Algorithm (A2), however, allows migration at non-optimal migration points, as already shown in the example illustrated in Figure 3.7. While in this example, the task migrates earlier than needed, Algorithm (A2) can also lead to unnecessary migrations. The maximal unused budget can be derived from the definition of t_{eval}. When t_{eval} is reached, and no delay of migration decisions is possible, $cMax_i(m)$ time units of execution remain at t_{eval}, with $m = \max(x_{curr}, x_{end(l)})$ In the worst case, the next migration point $x_{next} > x_{end(l)}$ is reached immediately after t_{eval}, so that for a migration at x_j, $cMax_i(j-1)$ time units of the budget remain unused. This is at least as much as for all other approaches, with a difference increasing with the difference between section WCETs.

In conclusion, the relatively low overhead of Approach (A2) comes at a cost of a less efficient usage of the given budget. For all presented algorithms, the maximal remaining budget depends on the length of the current of the maximal remaining section, so that regardless of the used algorithm, the budget can be used more efficiently with decreasing section lengths.

4.2 Overhead

In the previous chapter, the algorithms for dynamic migration decisions were presented under the assumption that no overhead needs to be considered. In practical application, however, different aspects of both scheduling algorithms and dynamic migration decisions have to consider the overhead caused by various operations, some of which will be analyzed in this section. The analysis will be limited to additional operations for dynamic scheduling decisions. Other overhead sources, such as operations for task management, or cache-effects caused by task migration, have already been there before the introduction of dynamic migration decisions, and will therefore not be discussed further.

Since all approaches require comparisons of the remaining budget, the remaining budget has to be updated at each timer tick. Additionally, Algorithms (A2) and (A3) require the identification of t_{eval}, which can also be done by a comparison in the tick handler, if no separate interrupt mechanism is used for this purpose. This overhead is constant and very small, but has to be considered for each timer tick.

The remaining overhead for the simple approach is relatively easy to analyze. At each migration point between two sections, the reachability of the next migration point is tested. The overhead required for this can be included in the WCETs of each section. While this overhead is constant, it can be high in a system that uses different address spaces for kernel and application data.

For all other approaches, the overhead at each migration point includes the logging of the current section, and a comparison with the currently defined evaluation point. The overhead for this is constant and can be inluded in the WCET for each section. Since this can be implemented without access to kernel data, the required overhead is relatively small, even if separate address spaces are used.

Access to kernel data is, however, needed at each evaluation point, additionally to the calculations for migration decisions. If evaluation points are triggered by application code, a context switch might be needed, depending on the operating system. For time-triggered evaluation points, overhead needs to be included for either polling in the tick handler, or an interrupt mechanism. The time required for this depends highly on operating system and hardware platform, so that generalized statements about this overhead are difficult to make. While in practical application, this overhead must be considered for each evaluation point, in the remaining section, it is assumed to be already implicitly included.

The remaining discussion will focus on the effort needed for the calculations at each evaluation point. The time needed for these calculations is highly dependent on both task parameters and run-time behaviour. In order to estimate this overhead, it is useful to consider the purpose of the analysis, and limit the estimation to a set of scenarios that are relevant for this purpose. In this section, the overhead of dynamic migration decisions will be disussed for the following purposes:

- Partitioning of the task set: when a task set is partitioned, the assigned budgets must include time for dynamic migration decisions. In order to preserve schedulability, enough additional budget must be included for each τ_i^l, so that $x_{end(l)}$ is still always reachable.

- Dynamic migration decisions: the overhead must be considered in reachability checks. In order to preserve schedulability, any migration point x_j is only allowed to be identified as reachable, if the additional overhead still allows the task to migrate at x_j within the remaining budget. Additionally, for each partial task τ_i^l, $x_{end(l)}$ must still be always identified as reachable.

- Evaluation of the average-case run-time overhead: even if the task set remains schedulable, split tasks should not be significantly slowed down by dynamic migration decisions. To evaluate this, the average-case run-time overhead is estimated.

The overhead required for these purposes will be discussed for each algorithm in the remaining section.

4.2.1 Overhead for Partitioning

When a task is split by the partitioning algorithm, the assigned budget must include overhead for dynamic migration decisions. As stated above, the budget of a partial task τ_i^l is sufficiently large, if $x_{end(l)}$ is reachable, i.e. if the budget is not exceeded, if τ_i^l migrates at $x_{end(l)}$. Thus, the set of relevant scenarios can be limited to cases, in which the current partial task migrates at $x_{end(l)}$. These cases are analyzed in the following subsection for all algorithms using evaluation points.

4.2.1.1 Overhead for Algorithm (A1)

The overhead for relevant scenarios in this case is illustrated in Figure 4.1. This figure depicts the overhead needed for partial task τ_i^l, in case $x_{end(l)}$ is chosen for migration. Note that while the figure depicts a partial task in the middle of τ_i, the following discussion can also be applied to the first and last partial tasks τ_i^1 and τ_i^q.

At the start and end of each partial task, time is required for task release and task migration, respectively. Since this overhead was already needed for statically assigned migration points, it will not be discussed further. Between evaluation points, other migration points have to be skipped. As already mentioned at the beginning of this section, the overhead needed for the logging of the current section and for skipping migration points is relatively small and constant, and can be included in the section WCETs.

The interesting part is the overhead caused by evaluation points. The number of evaluation points can be deduced by considering constraints on reachability in the relevant scenario. Since in all relevant cases, the partial task τ_i^l task will migrate at $x_{end(l)}$, $x_{end(l)+1}$ is always unreachable until the task migrates. This means that at the first evaluation point, $x_{end(l)}$ is the last reachable migration point and therefore chosen as the next x_{eval}. When $x_{end(l)}$ is reached, $x_{end(l)+1}$ is still unreachable, so that the task migrates immediately. Thus, in all relevant cases, each partial task needs exactly two evaluation points.

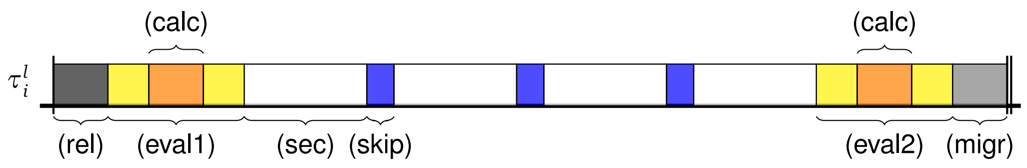


Figure 4.1 – Operations needed by Algorithm (A1) for partial task τ_i^l, if τ_i^l migrates at $x_{end(l)}$. At the start and end of the current partial task, overhead is caused by its release (rel), and the preparations for its migration (migr). Dynamic migration decisions are made at the evaluation point immediately after migration (eval1), and when $x_{end(l)}$ is reached (eval2), where the decision is made to migrate immediately. Both evaluation point require the calculation (calc) of the next evaluation point. Between these evaluation points, sections of the task are executed (sec). Between each consecutive section, a migration point is skipped (skip).

With a fixed number of evaluation points, the overhead for each evaluation point is needed in order to identify the additional overhead for each budget. At each evaluation point, the next evaluation point is calculated. This value is compared with the current migration point, and, depending on the result, the task either migrates or resumes execution. The only potentially non-constant overhead is the search for the next x_{eval}. In the general case, the time required by the search depends on the used search algorithm. In all relevant cases, however, $x_{end(l)}$ is identified as the next evaluation point, which makes it possible to reduce the required overhead. As discussed in the previous chapter, all search algorithms can start under the assumption, that $\max(x_{curr}, x_{end(l)})$ is reachable. Any search algorithm that starts with a reachability test of $\max(x_{curr+1}, x_{end(l)+1})$ can therefore provide the result in constant effort. While linear search does this implicitly, other search algorithms can be easily extended by this test, so that $\max(x_{curr+1}, x_{end(l)+1})$ is tested, before the actual search algorithm is started. With this modification, the overhead that each budget of a partial task needs to include is constant and independent of task parameters.

4.2.1.2 Overhead for Algorithm (A2)

The operations needed by approach (A2) in the relevant case are shown in Figure 4.2. As for the previous approach, the overhead caused for task release, task migration and the skipping of migration points will not be discussed further, and only the time needed for evaluation points is analysed.

As in Algorithm (A1), the number of evaluation points is fixed, since t_{eval} cannot be recalculated, if the current partial task τ_i^l migrates at $x_{end(l)}$. Any delay of t_{eval} is only possible, if $cMax_i(m)$ has changed since the current t_{eval} has been defined. This requires a change of $m = \max(x_{curr}, x_{end(l)})$. If the task will migrate at $x_{end(l)}$, then $x_{end(l)}$ has not been passed yet at t_{eval}, so that $x_{curr} \leq x_{end(l)}$. This means that both at the time of the definition of t_{eval} , as well at t_{eval} itself, $m = x_{end(l)}$. Since m stays the same, no recalculation of t_{eval} is possible.

With the limited number of evaluation points, the additional overhead to include in the budget is defined by the initial calculation of t_{eval}, the effort needed at t_{eval}, and the additional effort at x_{migr}.

Initially, the available budget is tested to ensure that migration decisions can be delayed. If this is the case, t_{eval} is calculated, and the task is resumed. Otherwise x_{migr} is calculated as the maximum of two available values. In both cases, only constant effort is needed. When t_{eval} is reached, x_{migr} is calculated after a failed attempt to delay t_{eval}. This also requires only a constant effort. At x_{migr}, migration is started in a constant amount of time.

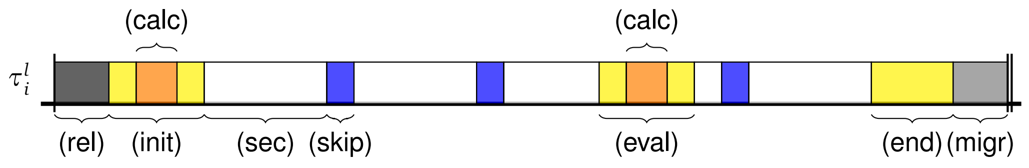


Figure 4.2 – Operations needed by Algorithm (A2) for the current partial task τ_i^l, if τ_i^l ends at $x_{end(l)}$. As for Algorithm (A1), overhead is caused by task release (rel) and migration (migr), as well as for skipping migration points (skip) between sections (sec). For dynamic migration decisions, overhead is needed at the start of τ_i^l, when t_{eval} is initialized (init), when t_{eval} is reached (eval), and when the chosen migration point has been reached (end). At the initial migration point and at t_{eval}, overhead is caused by the calculation of the next evaluation point (calc).

With a fixed number of evaluation points and a relatively small, constant effort for the calculations at each of them, only a constant overhead has to be included by the partitioning algorithm.

4.2.1.3 Overhead for Algorithm (A3)

The overhead needed for Algorithm (A3) is depicted in Figure 4.3. As in the previous algorithms, only overhead for evaluation points is considered, and overhead for task release, migration, and the skipping of migration points will be ignored. The number and effort of evaluation points can be derived from the results of the previous algorithms.

Initially, when t_{eval} is calculated, two possible cases need to be considered. If the budget is too small to define t_{eval}, the partial task proceeds as in Algorithm (A1), with the respective overhead. Otherwise, t_{eval} is calculated with constant effort. When t_{eval} is reached, x_{eval} is calculated. Since at this point $x_{next} \leq x_{end(l)}$, the x_{eval} will be set to $x_{end(l)}$, so that at $x_{end(l)}$ the task migrates after a calculation with constant effort, as shown for Algorithm (A1).

While this approach needs to include more operations than the previous approaches, the overhead for these operations is still constant.

4.2.2 Overhead for Dynamic Migration Decisions

Not only the partitioning algorithm, but also dynamic migration decisions have to consider the additional overhead, in order to ensure that all partial tasks migrate within their budget. This must be ensured for all migration points that are identified as reachable. Since this includes migration points after the statically assigned end points, the relevant cases cannot be limited to scenarios, in which the current partial task τ_i^l migrates at $x_{end(l)}$. Without this limitation, the overhead calculated in the previous subsection cannot be applied for this purpose.

The goal of this overhead calculation is to choose evaluation points so that at each evaluation point, migration within the given budget is still possible. This means, regardless of the algorithm, that the definition of any evaluation point e needs to include overhead for the following operations:

(E1) the remaining calculations until e is determined

(E2) all calculations at e, in case e cannot be delayed further

(E3) the initiation of migration at the selected migration point

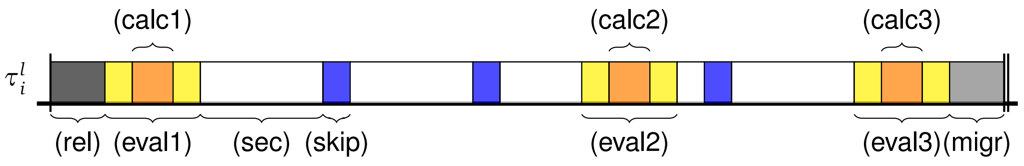


Figure 4.3 – Operations needed by Algorithm (A3) for the current partial task τ_i^l, if τ_i^l ends at $x_{end(l)}$. As for the previous algorithms, overhead is caused by task release (rel) and migration (migr), as well as for skipping migration points (skip) between sections (sec). Dynamic migration decisions need time at the start of τ_i^l, when t_{eval} is initialized (init), when t_{eval} is reached (eval1), and when the next chosen evaluation point x_{eval} is reached (eval2). At each evaluation point, overhead is caused by some calculations. Initially, t_{eval} is calculated (calc1), while at t_{eval}, x_{eval} is set (calc2). When x_{eval} is reached, the next evaluation point has to be calculated (calc3), before the decision is made to migrate immediately.

Note that only the case without further delay of e is considered. The other case is ignored, since the possibility to delay e implies the existence of a later point that still fulfills the above conditions.

Since the overhead needed for the above operations depends on the algorithm, this overhead will be discussed separately for each approach.

4.2.2.1 Overhead for Algorithm (A1)

Algorithm (A1) includes the overhead in reachability tests, by adding the additional overhead to the required WCET. The overhead needed for this can be estimated by analyzing the previously listed operations.

Operations (E2) and (E3) are needed when x_{eval} is reached, and include the calculation of the next evaluation point, and the subsequent decision to migrate. Since the task migrates immediately in all relevant cases, the calculation ends, when x_{curr+1} is identified as unreachable. As discussed in the previous subsection, this can be done in a constant amount of time, if the reachability of $\max(x_{curr+1}, x_{end(l)+1})$ is tested first.

Operations at (E1) include the remaining calculation of the last evaluation point, the subsequent decision against migration and the return to the task. Out of these operations, the remaining calculation is the only potentially non-constant part. Since the calculated overhead will be used for reachability tests, in order to determine the suitability of some migration point x_j as evaluation point, the overhead can be estimated under the assumption that x_j will be chosen and x_{j+1} is unreachable. Thus, for the remaining calculation, time needs to be included from the test of x_j until the test of x_{j+1}. The required effort for this depends on the used search algorithm.

Since linear search processes all migration points consecutively, only two loop iterations are needed, and only a constant effort is required.

Binary search, however needs logarithmic effort. When x_j has been identified as reachable, and x_p has not yet been tested, the reachability of $p - j$ migration points is still unknown. If binary search is implemented so that the lower element is chosen in intervalls of uneven length, then $\lfloor log_2(p+1-j)\rfloor$ additional loop iterations are needed, until x_{j+1} is tested. In order to prevent overhead calculations at run time, this number can be estimated by $\lfloor log_2(p-end(l))\rfloor$, before scheduling is started. Alternatively, the number of loop iterations can be estimated by a sufficiently high constant value. If, for example, overhead for 32 loop iterations is included, this overhead is sufficient for all tasks with less than 2^{32} sections, which is probably a reasonable assumption.

The search algorithm using the estimation by $cMax$ only needs a constant number of loop iterations. After the first loop has identified x_j as reachable, no additional reachable migration points can be identifed in the next loop iteration, and the first loop ends. The second loop tries to identify additional reachable migration points by linear search. With x_{j+1} unreachable, this loop will end after the first iteration. Thus, this algorithm needs to include the constant overhead needed by two iterations of the first loop, and one iteration of linear search.

In summary, the overhead to include in reachability tests for dynamic migration decisions depends on the search algorithm, and is either constant, or can be estimated by a relatively low, statically known value.

4.2.2.2 Overhead for Algorithm (A2)

In this approach, the overhead is subtracted from the calculated t_{eval}, so that the remaining budget at t_{eval} includes additional time for migration decisions. This overhead can be determined by considering the required actions for the previously listed operations.

At (E1), t_{eval} is recalculated after a test if migration decisions can be delayed. Operations for (E2) include the failed attempt to delay t_{eval}, and a subsequent calculation of x_{eval}, while at (E3), the current migration point is recognized as chosen migration point and migration is started.

These are the same operations that need to be included by the partitioning algorithm. As already discussed, only a constant overhead is required.

4.2.2.3 Overhead for Algorithm (A3)

Since different kinds of evaluation points are used in this approach, the overhead that need to be considered by migration decisions needs to be considered separately.

First, t_{eval} is initialized at the start of the task. The definition of t_{eval} must include the overhead for the remaining current calculation, the calculations at t_{eval}, and for migration at x_{eval}. While the former two operations have already been discussed for Algorithm (A2), the overhead for the last operation is the same as for Algorithm (A1). As in the previous algorithms, the required overhead is constant.

At t_{eval}, the next evaluation point x_{eval} is chosen as the maximum of x_{next} and $x_{end(l)}$. Since t_{eval} is not recalculated, no overhead is considered in this decision, and the correctness of the resulting x_{eval} has to be ensured by the previous calculation of t_{eval}.

When x_{eval} is reached, the task proceeds as in Algorithm (A1). Since, in this case, linear search is used, the required overhead is also constant.

4.2.3 Overhead at Run Time

The previous section has established the overhead to include, in order to prevent deadline misses. But even if no deadline is missed, split tasks should ideally not be slowed down significantly by dynamic migration decisions. To estimate, how much a migrating task will be slowed down, the general run-time overhead for dynamic migration decisions will be analyzed in this subsection.

Since schedulability has already been established, the average-case overhead is more interesting than the worst-case overhead. Therefore, the run time overhead will be discussed using a simplified example task τ_i.

Task τ_i has p sections. In order to simplify estimations, each section has a WCET of c time units, except for the last section, which has a WCET of $c_p \geq c$, which allows to show the effects of uneven section lengths. Thus, $cMax_i(j) = cMax_i = c_p$ for all migration points x_j. At run time, each section executes for $a * c$ time units, with $0 < a \leq 1$. With the given run times, the time at which a migration point $x_j < x_p$ is reached can be calculated:

$$t(x_j) = a * c * j$$

Dynamic migration decisions are made for the first partial task τ_i^l. If the available budget is used optimally, the example task will migrate at migration point $x_m < x_p$. For migration at x_m, at least c time units of the budget remain, when x_{m-1} is reached. This allows an estimation of the available budget:

$$\begin{aligned} & B_i^1(t(x_{m-1})) \geq c \\ \Rightarrow \quad & B_i^1(0) \geq t(x_{m-1}) + c \\ & \qquad = a * c * (m-1) + c \end{aligned}$$

4.2.3.1 Overhead for Algorithm (A1)

The relevant factors for run-time overhead are the number of evaluation points, and the time needed for each evaluation point. Both will be discussed for the example task.

In order to estimate the number of evaluation points for the example task, a function $f(n)$ will be defined. This function determines the number of remaining sections between the nth evaluation point and x_m. With this function, the nth evaluation point can be defined as $x_{m-f(n)}$. Since the nth evaluation point is calculated, when the $(n-1)$th evaluation point is reached, the current position at the time of calculation is $x_{curr} = x_{m-f(n-1)}$. With this information, the remaining budget at this time can be estimated:

$$\begin{aligned}
B_i^1(t_{curr}) &= B_i^1(0) - t_{curr} \\
&= B_i^1(0) - t\left(x_{m-f(n-1)}\right) \\
&\geq (a*c*(m-1)+c) - (a*c*(m-f(n-1))) \\
&\geq a*c*(f(n-1)-1)+c
\end{aligned}$$

Since each section has length c, the number of additional reachable migration points $r(n)$ at the nth calculation of x_{eval} can be estimated with the given budget:

$$\begin{aligned}
r(n) &= \left\lfloor \frac{B_i^1(t_{curr})}{c} \right\rfloor \\
&\geq \left\lfloor \frac{a*c*(f(n-1)-1)+c}{c} \right\rfloor \\
&= \lfloor a*(f(n-1)-1)+1 \rfloor \\
&\geq a*(f(n-1)-1)
\end{aligned}$$

From this, a recursive definition of $f(n)$ can be derived:

$$\begin{aligned}
f(0) &= m \\
f(n+1) &= f(n) - r(n+1) \\
&\leq f(n) - a*(f(n)-1) \\
&= (1-a)*f(n)+a
\end{aligned}$$

A non-recursive function can be defined as follows:

$$\begin{aligned}
f(n) &\leq (1-a)^n * m + a * \left(\sum_{i=0}^{n-1}(1-a)^i\right) \\
&= (1-a)^n * m + a * \left(1 + \sum_{i=1}^{n-1}(1-a)^i\right) \\
&< (1-a)^n * m + a * \left(1 + \frac{1}{1-(1-a)}\right) \\
&= (1-a)^n * m + a + 1
\end{aligned}$$

Since $f(n) \in \mathbb{N}$, $f(n)$ can be rounded down, so that:

$$f(n) \leq \lfloor (1-a)^n * m + 1 \rfloor$$

From this, the number of calculations of evaluation points can be approximated. If x_{m-1} is an evaluation point, then at most two further recalculations are needed when x_{m-1} is reached: the first calculation sets x_m as next evaluation point, while the second calculation at x_m identifies no further reachable migration points, so that the task migrates at x_m. If x_{m-1} is no evaluation point, only the seccond calculation is needed. This means, that when x_{m-1} is reached, at most two further calculations are needed. The number of all previous calculations can be estimated using $f(n)$. If x_{m-1} is the result of the nth calculation, then n can be derived by calculating the value of n for which $f(n) = 1$. Since the given definition of $f(n)$ is an approximation rather than an exact value, the minimal value of n for $f(n) \leq 1$ is estimated.

$$\begin{aligned}
& & \lfloor (1-a)^n * m + a + 1 \rfloor &\leq 1 \\
&\Longleftrightarrow & (1-a)^n * m + a + 1 &< 2 \\
&\Longleftrightarrow & (1-a)^n * m &< 1 - a \\
&\Longleftrightarrow & (1-a)^{n-1} &< \frac{1}{m} \\
&\Longleftrightarrow & \left(\frac{1}{1-a}\right)^{n-1} &> m \\
&\Longleftrightarrow & n - 1 &> log_{\frac{1}{1-a}} m \\
&\Longleftrightarrow & n &> log_{\frac{1}{1-a}} m + 1 \\
&\Longleftrightarrow & n &\geq \left\lfloor log_{\frac{1}{1-a}} m \right\rfloor + 2
\end{aligned}$$

Since, at x_{m-1}, two additional calculations are needed until the task migrates, at most $\left\lfloor log_{\frac{1}{1-a}} m \right\rfloor + 4$ calculations are needed. In summary, this means that the number of calculations is logarithmic, and increases with faster run times.

The time that is needed for each recalculation depends on the reachability of the remaining sections, and the used search algorithm. Linear search needs one loop iteration for each reachable migration point. From the start of partial task τ_i^l until its migration at x_m, all migration points from $x_{end(l)+1}$ until x_{m+1} are tested at least once, and no more than twice. As already discussed, the effort needed for binary search is logarithmic to the number of all remaining migration points.

The number of loop iterations for the third search algorithm can be estimated analogous to the number of evaluation points needed by Algorithm (A1). In this analogy, the estimated remaining budget after the nth loop iteration corresponds to the remaining budget at the nth evaluation point. In each loop iteration, the search algorithm estimates a reachable migration point by dividing the estimated budget by c_p, the migration algorithm calculates the next x_{eval} by effectively dividing the remaining budget by c. Both algorithms end, when the remaining budget is too small for c, or c_p, respectively. The number of loop iterations needed by this search algorithm can therefore also be estimated by a similar equation, in which m is substituted by the number of reachable sections, and a represents $\frac{c}{c_p}$.

With these estimations, each partial task can require a logarithmic number of evaluation points. At each evaluation point, logarithmic effort is needed for calculations, in addition to the overhead that is needed to invoke the system call representing evaluation points.

4.2.3.2 Overhead for Algorithm (A2)

As already discussed, each calculation of t_{eval} or x_{migr} needs only a relatively small, constant effort. Thus, the interesting part is the number of recalculations of t_{eval}. In order to estimate the magnitude of the number of evaluation points, first, the necessary conditions fur multiple recalculations are defined. An evaluation point can only be delayed, if the largest remaining section has been completed between the previous and the current evaluation point. For n recalculations, a subsequence of n sections with descending WCETs is needed. All sections in this subsequence must be larger than all following sections outside of this subsequence. At run time, each of these sections must complete its execution, until the next evaluation point is reached. This means, that for a larger n, the average run time of these sections must be shorter than the difference between their WCETs.

In theory, it is possible to construct scenarios in which the number of evaluation points is linear to the number of migration points. Realistically, considering the above conditions for a large number of recalculations, the number of additional evaluation points is likely to be very low. In the given example task, no recalculation is possible, since $cMax_i(j)$ is equal for all migration points $x_j < x_p$.

4.2.3.3 Overhead for Algorithm (A3)

The run-time overhead for this approach can be derived from previous results. As in Algorithm (A2), a small constant effort is needed for the initial calculation of t_{eval}, as well as the calculation of the first x_{eval}, when t_{eval} is reached. As opposed to the previous algorithm, no recalculations of t_{eval} are possible, so that the remaining run-time overhead is caused only by recalculations of x_{eval}.

Compared to Algorithm (A1), the remaining budget at each x_{eval} is bounded by $cMax_i$. This limits both the number of additional evaluation points, and the number of reachable migration points at each evaluation point. The number of additional evaluation points is limited by the number of migration points that can be reached within the remaining budget. In the example task, at least c time units of budget remain when x_{m-1} is reached. Since each section until x_{m-1} needs $a * c$ time units of execution, at most $\left\lfloor \frac{c_p - c}{a * c} \right\rfloor + 1$ sections can be executed until the task migrates at x_m.

The number of reachable migration points at each x_{eval} is at most $\left\lfloor \frac{cMax_i}{cMin_i} \right\rfloor$, which limits the number of loop iterations, when the next evaluation point is searched by linear search.

Both number and effort of evaluation points decrease with similar section lengths. As in Algorithm (A1), the number of evaluation points increases with the ratio of WCETs to actual run times, although with faster run times, this approach is likely to need less evaluation points. The bound on the number of reachable migration points at each x_{eval} allows the use of a simpler search algorithm, and limits the number of loop iterations to a relatively low value.

Compared to Algorithm (A2), the average-case overhead for this algorithm is higher in most cases.

IMPLEMENTATION 5

The previously described algorithms are implemented on a Raspberry Pi v2 model B, which features a quadcore processor based on ARM Cortex-A7 [Ras]. As operating system, the FreeRTOS port piRTOS is used. Starting with the already existing partitioned EDF scheduling algorithm, task migration is introduced, in order to provide a semipartitioned scheduler. Based on the infrastructure for semipartitioned scheduling, the three algorithms using evaluation points are implemented.

In order to provide some background for the implementation, first, the initial situation is outlined, which is given by the partitioned scheduling algorithm provided by piRTOS. Based on this, the additionally introduced functionality for task migration is described, and the actual implementation of dynamic migration decisions is presented.

5.1 Initial Situation

The implementation of dynamic migration decisions is based on the partitioned scheduling algorithm provided by piRTOS [Jar19]. Since piRTOS is a port of FreeRTOS for the architecture of a Raspberry Pi with multiple cores, this section will first provide some general information about FreeRTOS, before the implementation of partitioned scheduling in piRTOS is described.

5.1.1 FreeRTOS

FreeRTOS [Bar16] is an open-source operating system for real-time applications. It is designed mainly for embedded systems and small microprocessors, and provides a real-time scheduler, memory management and other functionalities, such as synchronization primitives or mechanisms for communication between tasks. These functionalities can be used by a user-defined application, which consists of a set of tasks [Bar16].

When tasks are scheduled dynamically, each task has a state, which is represented by a task queue. Tasks that are ready for execution are contained by the readylist, while tasks that cannot be executed, before some event occurs, are located in a queue for blocked tasks. Multiple queues for blocked tasks exist for different kinds of events. One of these queues is the list for delayed tasks, which contains tasks that wait for the release time of their next job. Additionally, event lists can be defined, which contain tasks that wait for other events. Tasks that are not currently available for scheduling are located in a list for suspended tasks.

Due to the assumptions that were made about tasks in Chapter 2, only some of these task states are relevant for the remaining chapter. While the lists for ready and delayed tasks will be used by the implementation of dynamic migration decisions, the list for suspended tasks and all other lists for blocked tasks will not be discussed further, since all jobs are assumed to be always ready for execution from release until completion.

5.1.2 Partitioned Scheduling in piRTOS

In order to use FreeRTOS on a Raspberry Pi with four cores, the FreeRTOS port piRTOS [Jar19] is used. This port provides multicore scheduling algorithms for global and partitioned scheduling, but no implementation of semipartitioned scheduling. Thus, a semipartitioned scheduling algorithm is added, before dynamic migration decisions are introduced. Semipartitioned scheduling is implemented by adding a function to migrate a task to its next assigned core. In order to provide some context for this implementation, the relevant existing infrastructure provided by piRTOS is outlined first.

As FreeRTOS, piRTOS organizes tasks by using different queues representing task states. When partitioned scheduling is used, each core has its own set of lists. At run time, each task is passed between the lists of its assigned core, according to its current state.

The states that are relevant for the implementation of this thesis, as well as the functions that are used to change the state of a task, are illustrated in Figure 5.1. A task changes into the ready state, when its next job is released. Job releases are implemented in the function `xTaskIncrementTick()`, which is used to handle timer interrupts. This function identifies all tasks that need to be released at the current time instant, and passes these tasks from the list for delayed tasks to the readylist. If the priority of one of the currently released tasks is higher than the priority of the currently running tasks, a switch of the currently running task is requested. When a job of a periodic task ends, `vEndTaskPeriod()` is called, which removes the current task from the readylist, calculates its next release time, and inserts it into the list for delayed tasks. Before `vEndTaskPeriod()` returns, the currently running task is switched.

Switches of the currently running task are implemented before the return from an interrupt, if a designated flag is set. In this case, the function `vTaskSwitchContext()` is called. This function chooses the new highest-priority task from the readylist and sets it as the currently running task. After this task is chosen, its context is restored before returning from the interrupt. Since `vTaskIncrementTick()` is already executed as interrupt handler, it suffices to set the specified flag in order to request a task switch. In order to update the currently running task outside of interrupts, a software interrupt and subsequent task switch can be triggered by the function `taskYIELD()`.

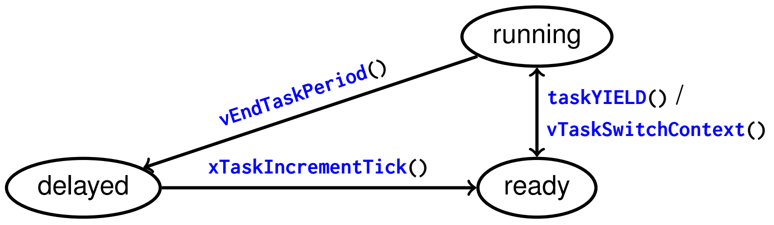


Figure 5.1 – Existing data structures for partitioned scheduling: tasks are passed between running, ready and delayed state by different functions of the operating system.

5.2 Migration in piRTOS

Based on the existing infrastructure for partitioned scheduling, semipartitioned scheduling is implemented by adding a function to migrate tasks between cores.

Compared to global scheduling, the possibilities for task migration in semipartitioned scheduling are limited. Based on this limitations, some assumptions about task migration are made, in order to reduce the complexity of the required infrastructure. In the following implementation, task migration is only possible for statically selected tasks with statically assigned target cores. Each core is assigned to at most two migrating tasks. Out of these tasks, at most one task starts at this core, and at most one task starts on another core. Before a task migrates, migration has to be requested explicitely by the original core.

Based on these assumption, a function for task migration is added. The implementation of this function is described in the remaining section, starting with the additional data structures, and continuing with the required synchronization between original and target core.

5.2.1 Transfer Between Cores

When a task migrates, it has to be passed between task lists of different cores. Since parallel access to already existings task lists would require additional synchronization at multiple points, migrating tasks are not directly inserted into a task list on the target core. Instead, migrating tasks are passed between lists indirectly, using an additional buffer for newly arrived tasks on each core.

The interaction between this buffer and the existing task queues is depicted in Figure 5.2. When a task requests migration, the original core inserts this task into the buffer for arrived tasks on the target core. Later, the target core removes this task from the buffer, and inserts in one of its task lists, depending on the given task parameters.

With the above limitations of task migration, race conditions regarding the access to the buffer for arrived tasks can be avoided without further synchronization. Since at most two tasks can arrive at each core, it is possible to assign a separate buffer slot to each task, by allocating two slots on each core. Each buffer slot is only written, when its assigned task is either inserted or removed. Since these operations cannot happen simultaneously, no further synchronization of this buffer is required.

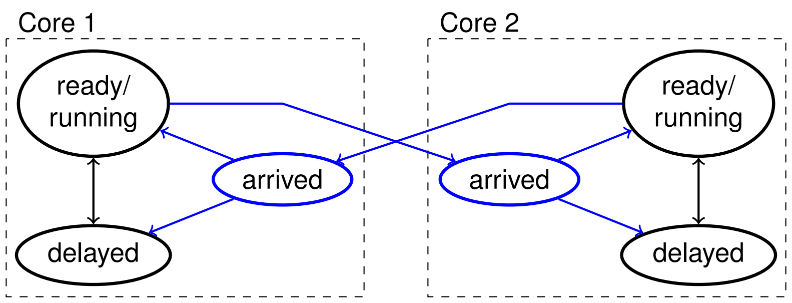


Figure 5.2 – Data structures to allow task migration between two cores: the existing lists are extended by a buffer for migrated tasks. In order to pass tasks between cores, the migrating task has to be inserted in the designated buffer on the target core. From there, the target core is responsible for inserting the task into the appropriate queue.

5.2.2 Synchronization for Task Migration

Even though race conditions regarding the access to the buffer for arrived tasks can be avoided, some synchronization is still required for the coordination between original and target core. After a migrating task has been inserted into the buffer for arrived tasks, the target core must be notified. This notification can be implemented in different ways, such as per interprocessor interrupts or mailbox mechanisms. For simplicity, in this implementation, the arrival of a migrating task is detected via polling in the tick handler. At each timer tick, the target core tests for arrived migrating tasks, and inserts these tasks in the appropriate task queue.

Without further synchronization, this approach leads to a race condition, as depicted in Figure 5.3. Since migration is initiated by the original core, a migrating task is written to the buffer for arrived tasks while it is still running on the original core. If the target core detects and schedules this task before the task switch on the original core, the migrating task executes on two cores at the same time.

In order to prevent this scenario, a new flag is added for each task, in order to signal that this task is still used by the original core. This flag is set by the original core before the insertion in the buffer, and reset by the original core after the switch to the next task. When the target core finds a migrating task in its buffer, and this flag is still set, the task is ignored until the next timer tick. This can lead to a delay between the partial tasks, as depicted in Figure 5.4. In the worst case, the tick handler of the target core tests the flag immediately before it is reset. In this case, the delay consists of the time interval from this unsuccessful test until the migrating task can be scheduled, i.e. until the return from the next timer tick, in which the flag was reset. The size of this interval is less than two timer ticks, but needs to be considered, when the task set is partitioned. If a different communication mechanism between original and target core is used, a reduction of this delay might be possible.

5.2.3 Implementation of Task Migration

The previously described mechanism for task migration is implemented at different points in the code. The initiation of task migration on the original core is implemented in the new function

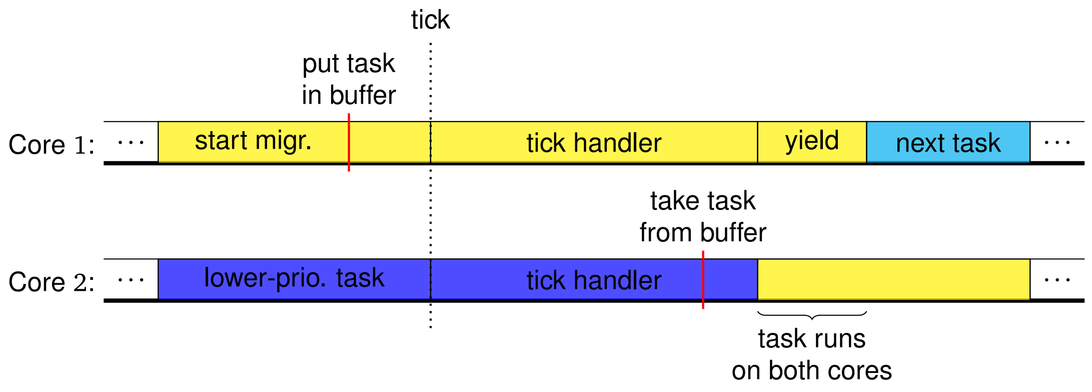


Figure 5.3 – Potential race condition without further synchronization: the migrating task is taken from the buffer by Core 2, before Core 1 has scheduled the next task. As a result, the task is scheduled on multiple cores at the same time.

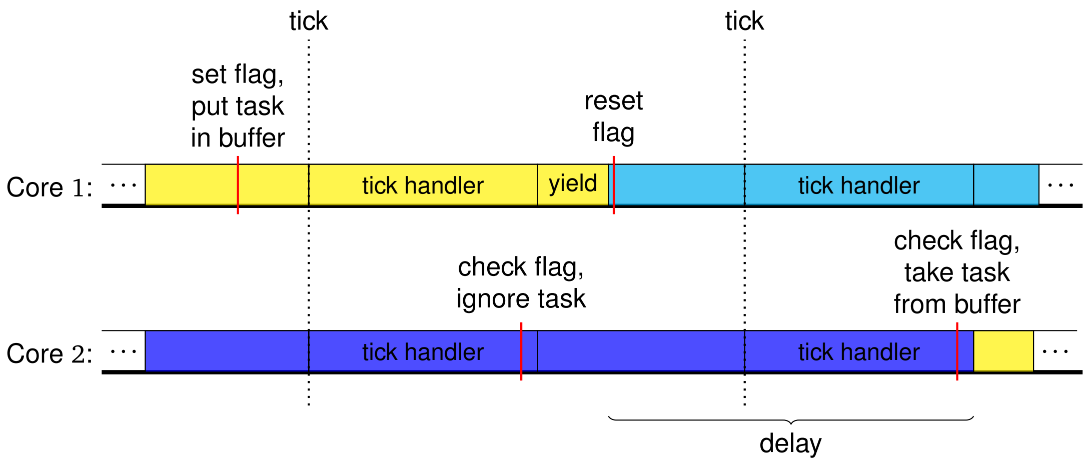


Figure 5.4 – Synchronization of task switches: Core 2 cannot take the migrating task from the buffer, until Core 1 has reset the in-use flag after the switch to the next task.

`vTaskMigrate()`, while instructions to reset the additional flag are added to the assembler code after each task switch. The polling for arrived tasks on the target core is added to the tick handler.

For these functions, additional data is needed for each task struct. A new member variable `l` represents the index of the current partial task. For each partial task, additional information is stored, such as identifiers of assigned cores, which are contained in the array `assignedCores[]`. Aside from scheduling information, the flag `inUse` is added to each task, in order to indicate whether the task is still in use by its original core. To simplify the assembler code that is needed to reset this flag, each core gets an additional pointer `inUsePtr`, which can be set to the address of the in-use flag of the current task before the call to `taskYIELD()`.

With this information, the function `vTaskMigrate()` can be implemented, as shown in Listing 5.1. In this function, the index `l` of the current partial task is updated, and the target core is identified. This information is used in order to write a pointer to the current task in the buffer on the target core, after the in-use flag is set. When the task has been written to the buffer, it is removed from the readylist, and `inUsePtr` is set to a pointer to the element `inUse` of the current task. The latter two operation require the suspension of the scheduler, so that no task can change its state in the meantime. This is needed to prevent race conditions in regards to the access of the readylist, and in order to prevent unrelated task switches from resetting the current in-use flag, before the current task is removed from the readylist. After the scheduler has been resumed again, the current task is switched out by a call to `taskYIELD()`. When `taskYIELD()` has switched to the next task, the `inUse` flag is reset, so that the migrating task can be passed to the appropriate task list on the target core.

Note that in this implementation, buffer slots are assigned depending on whether the task migrates during the execution of its current job or migrates back to its first assigned core after completing a job. According to the previously defined assumptions about migrating tasks, no assignment conflicts are possible.

5.3 Dynamic Migration Decisions

If the previously described function `vTaskMigrate()` is called at fixed points in the application code, semipartitioned scheduling with statically selected migration points can be implemented. Based

```
vTaskMigrate(isEndOfTask) {
    task = currentTask[cpuid];

    task->l = isEndOfTask ? 0 : task->l + 1;
    targetCore = task->assignedCores[task->l];

    task->inUse = 1;
    taskArrived[targetCore][isEndOfTask ? 0 : 1] = task;

    // suspend scheduler on current core

    inUsePtr[cpuid] = &task->inUse
    // remove task from readylist

    // resume scheduler on current core

    taskYIELD();
}
```

Listing 5.1 – Function for Task Migration

on this, dynamic migration decisions are introduced. Instead of calling `vTaskMigrate()` at statically defined points, a new function `vTaskEndSection()` is called at each migration point, in order to decide at run time, whether to call `vTaskMigrate()`. This decision is based on one of the previously defined algorithms. Since the implementation of the simple approach is not very complex, only the implementation of the three algorithms using evaluation points will be presented in the remaining section.

Even though piRTOS does not implement separate address spaces for operating system and application, the implementation of dynamic migration decisions will keep the values of x_{curr} and x_{eval} in the application code, in order to clarify the concept. The value of t_{eval}, which is needed by Algorithms (A2) and (A3), however, does not need to be accessed by the application and will therefore be stored by the operating system.

All algorithms for dynamic migration decisions need some additional information about both partial tasks and migration points, which is stored in the task struct. These additional variables will be discussed first, before the implementation of dynamic migration decisions is presented. The presentation of these algorithms will start with a function to skip migration points, which is identical for all algorithms using evaluation points. In order to decide which migration points will be skipped, the evaluation points are implemented, which will be discussed for each algorithm separately.

5.3.1 Data Structures for Dynamic Migration Decisions

Since dynamic migration decisions are based on information about migration points and partial tasks, this information is added to each task struct. An overview of this additional data is provided in Listing 5.2.

In order to decide about reachability, the currently available budget is stored in the member variable `budgetLeft`. The remaining budget is initialized at the release of each partial task, using the assigned budgets stored for each partial task in the array `budgets[]`. It is decremented at each timer tick that interrupts the execution of this task. The statically assigned end points for each partial task are contained in the array `plannedEnds[]`.

```
struct TCB {
    ...

    // already defined:
    inUse;

    numCores;         // q
    coreIDs[];        // ids of assigned cores
    relDeadlines[];   // partial deadlines
    l;                // index of current partial task

    // additional information about core allocations
    budgets[];        // budgets allocated on assigned cores
    plannedEnds[];    // end(l) for each l
    budgetLeft;       // remaining budget
    budgetEval;       // budget - t_eval, needed for (A2), (A3)

    // Information about sections of task
    numSections;          // p
    WCETsCumulative[];    // wcet from x_0 until x_(j+1)
    WCETsMax[];           // cMax_i(m)
    xCurrPtr;             // pointer to x_curr
    xEvalPtr;             // pointer to x_eval or x_migr

    ...
}
```

Listing 5.2 – Additional Member Variables for Dynamic Migration Decisions

Aside from partial tasks, information about migration points is needed. The number of sections is given by `numSections`, while information about section WCETs is stored in the array `WCETsCumulative[]`. Instead of the WCET of each section, this array contains the WCET from the start of the task until the end of each section. This makes it possible to calculate the WCET between two arbitrary migration points by subtracting the cumulative WCETs from each other. The maximal section WCET following each migration point is contained in the array `WCETsMax[]`.

Since x_{curr}, x_{eval} and x_{migr} are each stored by the application, the task struct contains only pointers to each of these values. These pointers are represented by the variables `xCurrPtr` and `xEvalPtr`. When an evaluation point is defined as a point in execution time, the variable `budgetEval` contains the remaining budget at the next evaluation point. In order to simplify comparisons with the remaining budget, this variable contains the value of $B_i^l(t_{eval})$ instead of t_{eval}. Since this value is never accessed by the application, it can be stored in the task struct.

At run time, this information can be accessed by retrieving the appropriate array elements. In order to simplify the access to these elements, and in order to avoid the translation between zero-based array indices and one-based indices for sections and partial tasks, some macros are defined in order to access array elements. An overview over these macros and their definitions is given in Listing 5.3. For the current partial task τ_i^l, the statically assigned end point $x_{end(l)}$ can be retrieved by the macro `CURR_PLANNED_END()`. WCETs between two arbitrary migration points $x_{i,j}$ and $x_{i,k}$ of the current task τ_i can be calculated in constant time by the macro `WCET_BETWEEN(j, k)`. When WCETs are used by dynamic migration decisions, the relevant WCET starts usually at x_{curr} and includes additional overhead. For this purpose, the macro `WCET_UNTIL(j, ovrh)` can be used. The maximal section WCET of the current task τ_i following a specified migration point $x_{i,m}$ can be retrieved by using the macro `WCET_MAX(m)`.

```
// WCET from the start of a given task until migration point x_j
#define WCET(task, j) \
    ( j ? 0 : task->WCETsCumulative[j - 1] )

// WCET between two migration points of the currently running task
#define WCET_BETWEEN(j, k) \
    ( WCET(current_task[cpuid], k) - WCET(current_task[cpuid], j) )

// WCET until the currently running task reaches migration point x_j, if a ↘
    specified overhead is included
#define WCET_UNTIL(j, ovrh) \
    ( WCET_BETWEEN(*current_task[cpuid]->xCurrPtr, j) + ovrh )

// maximal section WCET following migration point x_m of the currently ↘
    running task
#define WCET_MAX(m) \
    ( current_task[cpuid]->WCETsMax[m] )

// statically assigned end point of the currently running partial task
#define PLANNED_END(task) \
    ( current_task[cpuid]->plannedEnds[current_task[cpuid]->l] )
```

Listing 5.3 – Macros for Accessing Array Data

5.3.2 Skipping Migration Points

With given values for x_{curr} and x_{eval}, a decision can be made whether to skip the current migration point. This decision is made at each migration point, and is the same for all algorithms using evaluation points. It is implemented by the function `vTaskEndSection()`, which is shown in Listing 5.4. This function increments the value of the current migration point, compares it to the value of the next evaluation point, and decides accordingly whether to skip the current migration point. If the current migration point cannot be skipped, `vTaskEvaluate()` is called, which implements evaluation points and depends on the used algorithm. Since the value of x_{curr} is written in `vTaskEndSection()`, a pointer to its value is passed as parameter. Since, in Algorithms (A2) and (A3), the value of x_{eval} can be changed asynchronously when t_{eval} is reached, this parameter is also passed as a pointer. In order to reduce run-time overhead, this function can be inlined in the application code, but for clarity, it is presented as a separate function.

Since evaluation points in Algorithm (A1) are always reached synchronous to the task execution, no race conditions can occur between skipped migration points and evaluation points. Algorithms (A2) and (A3), however, define some evaluation points as time instances, which can be reached

```
vTaskEndSection(xCurrPtr, xEvalPtr) {
    *xCurrPtr++;
    if (*xCurrPtr == *xEvalPtr) {
        vTaskEvaluate();
    }
}
```

Listing 5.4 – Function representing Migration points

at any point, including during the execution of `vTaskEndSection()`. Since both x_{eval} and x_{curr} are accessed when t_{eval} is reached, race conditions must be ruled out.

A possible race condition can occur, if, due to compiler optimizations, x_{curr} is compared to x_{eval}, before its updated value is written back to its memory location. If, in this case, t_{eval} is reached immediately after x_{eval} has been read for the comparison, and if x_{curr} has not been written back yet, the next x_{eval} is calculated using an outdated value of x_{curr}. If x_{curr} has just been set from x_{j-1} to x_j, then x_{eval} can might be set to $x_{next} \leq x_{curr+1} = x_j$. Since the new value of x_{eval} will not be used until the next migration point, this evaluation point will be missed.

In order to prevent evaluation points from being missed, it suffices to ensure that x_{curr} is always written back to its memory location before it is compared to x_{eval}, which can be done by declaring x_{curr} as volatile.

5.3.3 Dynamic Migration Decisions

If a migration point cannot be skipped, `vTaskEvaluate()` is called. The behaviour of this function is different for all approaches, so that the implementation of this function and dynamic migration decisions in general are discussed separately for each algorithm in the remaining section.

Note that additional to the implementation of `vTaskEvaluate()`, the first evaluation point of a task is set before the return from `vEndTaskPeriod()`, after the current job has been released. Since this initialization is similar to the operations at the start of each other partial tasks, a discussion of changes in `vEndTaskPeriod()` will be omitted.

5.3.3.1 Dynamic Migration Decisions of Algorithm (A1)

In Algorithm (A1), all evaluation points are defined as selected migration points, so that all migration decisions are made in `vTaskEvaluate()`, which is shown in Listing 5.5.

When an evaluation point is reached, the algorithm tries to delay migration by recalculating x_{eval}. If further migration points are reachable, x_{eval} is set to a new value, and `vTaskEvaluate()` returns to the task. Otherwise, migration cannot be delayed further, and the task migrates, using the previously defined function `vTaskMigrate()`. When the migrated task is resumed on the next core, the next x_{eval} is calculated. On the new core, it is possible that the next section is larger than the budget that is assigned to this core, so that the task has to migrate again immediately. For this reason, migration

```
void vTaskEvaluate() {
    task = current_task[cpuid];
    x_curr = *task->xCurrPtr;

    x_eval = calculateNextEvalPoint();

    while (x_curr == x_eval) {
        vTaskMigrate(false);
        x_eval = calculateNextEval();
    }

    *task->xEvalPtr = x_eval;
}
```

Listing 5.5 – Implementation of Dynamic Migration Decisions in Algorithm (A1)

and recalculation are executed in a loop. Note that this is only possible, if the next section was assigned to a later partial task, so that schedulability condition (S2) still holds.

In order to identify the next evaluation point, `calculateNextEval()` is called, which implements the used search algorithm. As already discussed, different search algorithms can be used. Since both linear and binary search are widely known search algorithms, only the search algorithm that identifies migration points by estimating the section lengths will be presented. This algorithm is shown in Listing 5.6.

This search algorithm consists of two loops. In the first loop, reachable migration points are identified by estimating the WCET of each section by the maximal remaining section WCET. Each loop iteration starts with some migration point $x_{minEval}$ that is known to be reachable. The remaining

```
#define OVRH_NEXT   WCET_LOOP_1 + WCET_LOOP_2

#define OVRH_THIS_1 2 * WCET_LOOP_1 + WCET_LOOP_2
#define OVRH_THIS_1 2 * WCET_LOOP_2

#define OVRH1       OVRH_SYS + OVRH_NEXT + OVRH_THIS_1
#define OVRH2       OVRH_SYS + OVRH_NEXT + OVRH_THIS_2

calculateNextEvalPoint() {
    task = current_task[cpuid];
    x_curr = *task->xCurrPtr;

    minEval = MAX( x_curr, CURR_PLANNED_END());

    // loop 1: identify reachable migration points using cMax_i(curr)
    while (True) {
        // estimate remaining budget after minEval is reached
        estBudgetLeft = task->budgetLeft - WCET_UNTIL(minEval, OVRH1);

        // estimate number of additional reachable migration points
        step = estBudgetLeft / WCET_MAX(minEval);
        if (step < 1) {
            break;
        }

        // check if end of task is reachable
        if (minEval + step >= task->numSections) {
            return task->numSections;
        }

        minEval += step;
    }

    // loop 2: identify further reachable migration points via linear search
    for (i = minEval; i < task->numSections; i++) {
        if (WCET_UNTIL(i + 1, OVRH2) > task->budgetLeft) {
            // i + 1 unreachable
            return i;
        }
    }
    return task->numSections;
}
```

Listing 5.6 – Search Algorithm to Identify the Next Evaluation Point

budget when $x_{minEval}$ is reached is estimated by subtracting the WCET until $x_{minEval}$ from the currently remaining budget. The remaining budget at $x_{minEval}$ is then used to identify further reachable migration points by dividing its value by the maximal section WCET after $x_{minEval}$. If no further migration point can be identified as reachable, the loop ends. If this number exceeds the number of available migration points, the end of the task is known to be reachable, and x_p is returned as next evaluation point. Otherwise, this number is used to update $x_{minEval}$ before the next loop iteration. Since some reachable migration points can be missed by this estimation, the final result is obtained by a second loop via linear search.

In each WCET calculation, overhead is included as discussed in Chapter 4. As discussed, this overhead includes the remaining time for the current search, the time for the search at the next evaluation point, and a constant overhead for all other operations. In Listing 5.6, the required time is calculated statically, with given WCETs for each loop iteration, and for the operations outside of the search algorithm. For the current search, different values are included for each loop. If the last reachable migration point has been identified in the first loop, only one further iteration of each loop is needed, so that it suffices to include two iterations of the first, and one iteration of the second loop. The second loop only requires two iterations from the identification of the last reachable migration point until the end of the loop. The overhead for the search at the next evaluation point is calculated for the case that at this point, no further migration points are reachable. Since both loops end, when the last reachable migration point has been identified, only one iteration of each loop has to be considered.

5.3.3.2 Dynamic Migration Decisions of Algorithm (A2)

In Algorithm (A2), migration decisions are made in both `vTaskEvaluate()` and in the function that handles t_{eval}. Since t_{eval} is defined as a point in time, a mechanism is needed to initiate migration decisions at the specified time. This can be done by different mechanisms, such as interrupts that are triggered after a specified amount of execution time. For simplicity, this implementation polls for t_{eval} in the tick handler, as shown in Listing 5.7. At each timer tick, the remaining budget is compared to `budgetEval`, and when both values are equal, migration decisions are made. In order to try to delay the evaluation point, t_{eval} is recalculated with current run-time information. If no such delay is possible, x_{migr} is defined. The calculated value is then written to the memory location pointed to by `xEvalPtr`, so that the designated migration point can be identified by the application.

The initial value of `budgetEval` is set after each task migration in `vTaskEvaluate()`, as shown in Listing 5.8. As opposed to the previous algorithm, the current task always migrates, when the selected migration point has been reached. After the migrated task has been resumed on the target core, `budgetMin` is initialized. If this value is smaller than the remaining budget, the task can continue, until the t_{eval} is reached. Else, migration decisions cannot be delayed and a migration point is chosen immediately. If $x_{end(l)}$ has not yet been reached, x_{migr} is set to $x_{end(l)}$. Otherwise, the task migrates again instead of returning to the task. As in the previous algorithm, this possibility requires task migration to be implemented in a loop.

As in the previous algorithm, overhead needs to be included when the next evaluation point is calculated. This overhead is always constant and can be determined according to the discussion in Chapter 4. When a value for the required time is given, this value is added to `budgetEval`.

5.3.3.3 Dynamic Migration Decisions of Algorithm (A3)

Since this approach is a combination of the previous algorithms, its implementation will not be discussed in detail. Time-triggered evaluation points can be implemented similar to Approach

```c
xTaskIncrementTick() {
    ...
    task = current_task[cpuid];
    if (task->budgetLeft == task->budgetEval) {
        // t_eval is reached
        x_curr = *task->xCurrPtr;

        m = MAX(x_curr, CURR_PLANNED_END());
        task->budgetEval = WCET_MAX(m) + OVRH;

        if (task->budgetEval == task->budgetLeft) {
            m = MAX(x_curr + 1, CURR_PLANNED_END());
            *task->xEvalPtr = m;
        }
    }
    ...
}
```

Listing 5.7 – Evaluation Point in Algorithm (A2)

```c
void vTaskEvaluate() {
    task = current_task[cpuid];
    x_curr = *task->xCurrPtr;

    for (;;) {
        vTaskMigrate();

        // initialize budgetMin for new core
        m = MAX(x_curr, PLANNED_END());
        task->budgetMin = WCET_MAX(m) + OVRH;
        if (task->budgetMin > task->budgetLeft) {
            return;
        }

        // not enough time to delay decision, decide here
        if (x_curr < CURR_PLANNED_END(task)) {
            *task->xEvalPtr = CURR_PLANNED_END();
            return;
        }

        // end(l) is already reached, try again on next core
    }
}
```

Listing 5.8 – Implementation of Selected Migration Points in Algorithm (A2)

(A2), except that in this case, no recalculation of `budgetEval` is needed. The implementation of `vTaskEvaluate()` is similar to Approach (A1), with the difference, that `budgetEval` is calculated after a migrated task is resumed. If `budgetEval` is too large for the assigned budget, this approach continues as in Algorithm (A1).

MEASUREMENTS 6

In Chapter 4, both benefits and overhead of dynamic migration decisions have been discussed from a theoretical view. In order to approximate the additional overhead, the overhead for all purposes discussed in Chapter 4 is measured. Additionally, the effect of dynamic migration decisions on split tasks is evaluated by comparing the chosen migration points and measuring response times. Before measurement results are presented, the applied methods are clarified.

6.1 Methods of Measurements

All measurements in this chapter are done on a Raspberry Pi v2 model B, using the implementation described in Chapter 5. As in Chapter 4, the focus of this section are the additional operations that are needed for dynamic migration decisions, as opposed to the overhead for task migration itself, which has already been needed before dynamic migration decisions were introduced. Thus, the overhead for `vTaskMigrate()` is excluded from measurements.

If not specified otherwise, time intervals are measured using the Processor Cycle Counter of the performance measuring unit, which provides a 32-bit register the value of which is incremented for each CPU cycle [Arm]. For measurements, the CPU frequency is set to 600 MHz.

Measurements are made for a task that is split into a specified number of partial tasks, each of which has a specified number of assigned sections. Since all migration points before $x_{end(1)}$ will be skipped by all algorithms, the number of sections in the first partial task is not relevant for most measurements and is set to one, if not specified otherwise. When a of sections for other partial tasks is specified, the WCETs of these sections are assigned randomly, using a uniform distibution between the minimal and maximal section WCETs, which are calculated from the assigned budget and a specified ratio of maximal to minimal section WCET.

Each section is implemented by a loop that waits actively, until a calculated number of timer ticks has passed. One timer tick refers to the time between two tick interrupts, which are generated by the Generic Timer. While in the original implementation of piRTOS, the interval between two tick interrupts has a length of 1.2 miliseconds, this time is set to 0.15 miliseconds instead, in order to increase the speed of measurements. Since dynamic migration decisions do not depend on absolute WCETs, when the relative size of the WCETs stays the same, this should not impact measurements results.

Note that the general significance of these measurements is limited. Since run times depend on the underlying hardware and software, these results cannot be generalized for all systems. For example, an operating system that separates the address spaces of system and application will need a significantly higher overhead at each evaluation point. Furthermore, the example task is run without further tasks in the task set, so that possible interference from other tasks is not included. Aside from this, measurements in general only represent a lower bound of required WCETs.

6.2 Overhead for Dynamic Migration Decisions

In Chapter 4, the additional overhead that is caused by dynamic migration decisions was discussed theoretically. In order to estimate the actual required time, each kind of overhead that has been analyzed in Chapter 4 will be measured in this section. This includes measurements of the overhead to include by the partitioning algorithm, the overhead to include by dynamic migration decisions, and the average-case run-time overhead. Furthermore, the overhead for skipping migration points and for polling for t_{eval} in the tick handler will be measured. Measurements results for all of these overheads will be presented in the remaining section.

6.2.1 Overhead for Skipped Migration Points

The WCET of each section needs to include the overhead for skipping migration points, which is implemented in `vTaskEndSection()`. Since this function is the same for all algorithms using evaluation points, this operation is measured only for Algorithm (A1) and the simple approach.

The time is measured from the call of `vTaskEndSection()` until the return from this function, for all function calls in which migration points are skipped. Measurements are made for a task that is split in two partial tasks, each of which has 101 statically assigned sections of 10 timer ticks each. Each section runs with its full WCET. Of this task, 50 jobs are measured, so that 10000 migration points are skipped. In order to avoid interference of tick interrupts, timer interrupts are disabled for the measured operations.

The results can be seen in Table 6.1. In average, the simple approach needs more time for each migration point, which can be explained by the additional accesses to the array that contains the section WCETs, which are needed for the reachability check. The maximal measured time is slightly above 1 μs for the simple approach and below 0.8 μs for Algorithm (A1). In relation to one timer tick, both values are relatively small. Even though the simple approach requires more time, the difference between both algorithms is relatively small. This is due to the lack of separation of address spaces in the given operating system.

Additionally, the time for polling of t_{eval} in the tick handler was measured. For this, the previously defined task was used, and the first 10000 unsucessfull checks of t_{eval} were measured. The average required time was 81.69 ns, with a standard deviation of 2.00 and a maximal value of 281.67 ns. While this is a relatively short time, this overhead is required for each timer tick.

Note that all measured operations in this subsection can be optimized. Different optimizations, such as function inlining, can be applied to reduce the run times of skipped migration points, while polling for t_{eval} can be avoided entirely by using an interrupt mechanism instead.

Simple Approach			Algorithm 1		
max	avg	stdev	max	avg	stdev
780.00	134.73	8.76	1033.33	220.86	9.58

Table 6.1 – Measured time in nanoseconds for each skipped migration point

6.2.2 Overhead to Include in Budgets

When the partitioning algorithm assigns tasks to cores, it has to include time for dynamic migration decisions in the budgets of each split task. The operations that need to be considered in the calculation of this time have already been analyzed in Chapter 4 and will be measured in this subsection.

At the start of each partial task, the initialization of the first evaluation point needs to be considered, under the assumption that the task will migrate at its statically assigned end. For this, the time is measured from the return from `vTaskMigrate()` until the return from `vTaskEvaluate()` for all algorithms. Additionally, the last evaluation point is measured, at which the task migrates, by measuring the time from the call of `vTaskEvaluate()` until the call of `vTaskMigrate()`. For all algorithms that use time-triggred evaluation points, the time for one such evaluation point at which no recalculation of t_{eval} is possible has to be included. These operations in the tick handler are also measured for Algorithms (A2) and (A3). The overhead that needs to be included in the budget of each split task is calculated as the sum of these measured overheads.

Measurements are made for a task that is split into four parts, each of which has two assigned sections of 10 timer ticks. Since, for this purpose, only cases with migration at the statically assigned migration points are relevant, all sections run with their full WCET. Of this task, 1000 jobs are measured, so that data is available for 3000 evaluation points of each type. As in the previous measurements, timer interrupts are disabled for the measured time intervals.

The results are shown in Table 6.2. For all algorithms, the maximal measured overhead lies between 2.2 μs and 3.3 μs , with Algorithm (A3) requiring the most, and Algorithm (A2) requiring the least time. This result is expected, since Algorithm (A2) migrates at x_{migr} without further reachability checks and does not need complex calculations for migration decisions, while Algorithm (A3) needs to include overhead for all types of evaluation points, while still requiring a call to `calculateNextEvalPoint()` before migrating.

With less that 4 μs , the overhead that needs to be added to each budget is relatively small and is unlikely to impact the schedulability of a given task system.

6.2.3 Overhead to Include in the Calculation of Evaluation Points

When the next evaluation point is calculated, the overhead for dynamic migration decisions must be considered by this calculation. In this subsection, this overhead is calculated from the measured overhead of all relevant operations, according to the analysis in Chapter 4,

As discussed in Chapter 4, the definition of time-triggered evaluation points needs to consider the same overhead that is added to the budgets. Since this overhead has already been measured in the previous subsection, it will not be discussed further.

	Algorithm 1			Algorithm 2			Algorithm 3		
	max	avg	stdev	max	avg	stdev	max	avg	stdev
Start	2015.00	283.92	56.23	470.00	103.74	11.60	2103.33	314.20	56.66
End	706.67	312.43	14.77	785.00	180.61	19.12	613.33	181.64	13.73
t_{eval}				953.33	229.26	22.93	488.33	145.33	10.87
Sum	2721.67	596.35		2208.33	513.61		3204.99	641.17	

Table 6.2 – Measured time in ns to include in the budgets of split tasks: For each algorithm, the relevant operations are measured and added. The overhead that needs to be considered by the partitioning algorithm can be estimated by the sum of the maxima of all partial overheads.

This leaves the overhead that needs to be included in reachability checks of Algorithms (A1) and (A3). This overhead is caused by the remaining calculation of x_{eval}, the return to the actual task, and the calculations when x_{eval} is reached, under the assumption that no further migration point is reachable.

The time for the remaining calculation can be determined from the WCETs of loop iterations in the used search algorithm. Thus, the time required for loop iterations is measured. Loop iterations are measured for a task that is split in two partial tasks with a budget of each 10000 timer ticks with 1000 sections assigned to the second partial task, each of which has a WCET of 10 timer ticks. In order to increase the number of loop iterations for the search algorithm that uses the estimation, an additional section is added to the end with a WCET of 100 timer ticks, and the second budget is increased accordingly. At run time, all sections run with 62,5% of their specified WCET. Loop iterations of multiple jobs are measured, until data exists for at least 2000 loop iterations. Timer interrupts are disabled during the search.

As shown in Table 6.3, the maximal measured time for loop iterations ranges between 0.8 μs and 1.6 μs . From these values, the required time for the remaining calculation can be determined, according to the analysis in Chapter 4. The result of these calculations is also shown in Table 6.3. For binary search, the number of required loop iterations is estimated by 32, which leads to a comparatively high cost of about 28.3 μs . Alternatively, a lower value could be calculated depending on the given task parameters. Since the other search algorithms need to include only a constant number of loop iterations, significantly less time has to be included for these algorithms with 2.1 μs for linear search and 4.2 μs for the search algorithm using the estimation.

Additional to the time for loop iterations, the time from the end of the search until the return from `vTaskEvaluate()` was measured for Algorithms (A1) and (A3), using the same task as before.

Since the overhead at x_{eval} with no further reachable migration points has already been measured for all algorithms in the previous subsection, the overhead to include in reachability checks can now be calculated by combining the given values for the used migration algorithm with the given value for the used search algorithm. Since linear search has already been chosen for Algorithm (A3), and ruled out for Algorithm (A1), the overhead is only calculated for the combinations of Algorithm (A3) with linear search, and Algorithm (A1) with the other two search algorithms.

The results for these combinations are shown in Table 6.4. With the already high estimated overhead for the remaining calculation of binary search, the combination of Algorithm (A1) and binary search needs to include by far the most time with more than 31.1 μs . In combination with the other search algorithm, only 7.1 μs are required. The least overhead needs to be included by Algorithm (A3), due to the relatively low overhead of the remaining calculation if linear search is used.

	Search using Estimation			Binary Search			Linear Search		
	max	avg	stdev	max	avg	stdev	max	avg	stdev
iter	1583.33	285.37	50.43	883.33	154.97	31.44	1028.33	157.85	23.10
calculation	2 * est + lin			32 * bin			2 * lin		
loop ovrh	4204.99			28266.56			2056.66		

Table 6.3 – Overhead for the remaining calculation, when x_{eval} has been identified. From the maximal time for one loop iteration, the overhead for the remaining calculation can be determined.

Considering that dynamic migration decisions currently calculate with time units of timer ticks, which have a significantly higher time interval than the calculated overheads, the current implementation estimates this overhead for all algorithms with one timer tick, which is unlikely to have a significant impact on the behaviour of dynamic migration decisions.

6.2.4 Run-Time Overhead

Aside from the overhead that needs to be considered by scheduling and migration functions, the average-case run-time overhead is measured for all algorithms using evaluation points. Since Algorithm (A1) can be combined with different search algorithms, the overhead caused by these algorithms will be compared.

6.2.4.1 Comparison of Search Algorithms for Algorithm (A1)

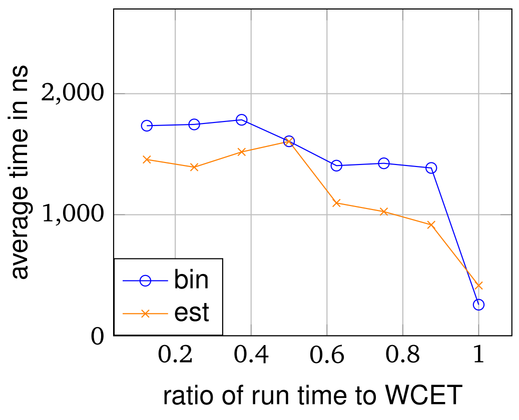
As discussed in Chapter 3, different search algorithms can be used to identify the next migration point. Since linear search has already been dismissed as potential search algorithm for Algorithm (A1), only the other two search algorithms are measured.

In order to measure the overhead for the search, measurements are made from the call of `calculateNextEvalPoint()` until the return from this function. Timer interrupts are disabled during measurements. The measured task is split into two partial tasks. In order to get run times for large amounts of input data, the second partial task is assigned 10000 sections. Both partial tasks have a budget of 100000 timer ticks. This budget is distributed among the sections according to a specified ratio of maximal to minimal section length. Since the accuracy of the estimation using $cMax_i(m)$ depends on the variance in section WCETs, and the possible range of ratios produced by the task splitting algorithm presented in [Kla+19] is unspecified, measurements are made for ratios of 1.5, 2, 4, and 9. Aside from representing a wide range of plausible ratios, these values result in integer values for maximal and minimal section WCET with a symmetric distance from the average section WCET of 10 ticks. In order to measure the behaviour for different numbers of available and reachable migration points, measurements are made with actual run times of $\frac{i}{8} * WCET$, with $i \in \{1, \ldots, 8\}$. Multiple jobs are measured, until at least 25 measurements have been made for each configuration.

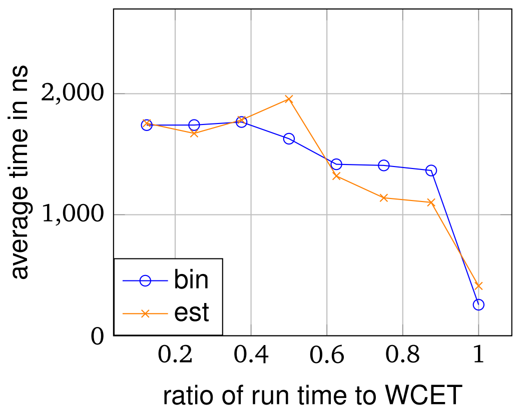
The results are shown in Figure 6.1. As expected, the overhead of binary search stays constant with differing variances of section lengths, while the search algorithm using the estimation needs more time for higher variances. Contrary to expectations, the results of this search algorithm improve with for $\frac{cMin_i}{cMax_i} = 4$. One possible explanation for this could be the effect of the maximal section

	Algorithm (A1)		Algorithm (A3)
	Binary Search	Search using Estimation	Linear Search
Current Search	28266.56	4204.99	2056.66
Return to Task	150.00	150.00	295.00
next x_{eval}	2721.67	2721.67	3204.99
Overhead	31138.23	7076.66	5556.65

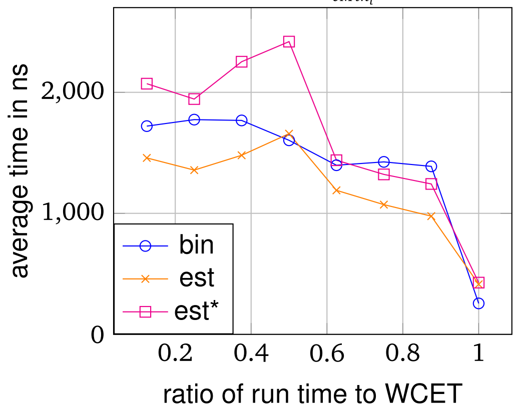
Table 6.4 – Calculation of the overhead to include in reachability checks. This overhead is calculated from the time for the remaining current calculation, the time until the return to the task, and to the overhead for the next migration point, which has already been measured in the previous subsection. The time is specified in nanoseconds.



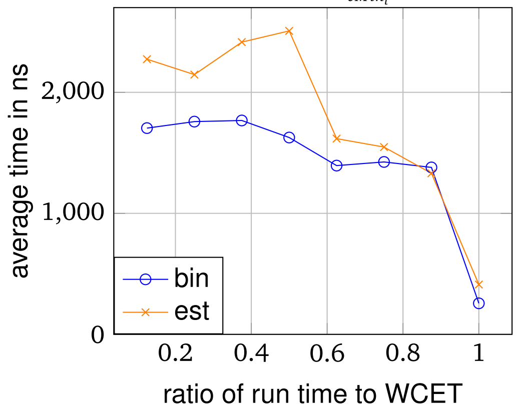
(a) Measurements for $\frac{cMax_i}{cMin_i} = 1.5$



(b) Measurements for $\frac{cMax_i}{cMin_i} = 2$



(c) Measurements for $\frac{cMax_i}{cMin_i} = 4$. Initially, the overhead was measured for a budget of 100000 with a maximal section WCET of 16. Additional measurements for a budget of 150000 and a maximal section section WCET of 24 were made, the results of which are deonoted as 'est*'.



(d) Measurements for $\frac{cMax_i}{cMin_i} = 9$

Figure 6.1 – Measured time for binary search and the search algorithm using the estimation depending on the ratio of run times to WCETs, measured for different ratios of maximal to minimal section WCET.

WCET on the time for the integer division that is needed in each loop iteration. Integer division is not provided by all ARM-v7 processors [Arm], and is therefore implemented by a library function. With the given budget size, the maximal section has a WCET of 16 time units, which is a power of 2 and could therefore lead to beneficial behaviour of the division function.

In order to test this, further measurements have been made for a task with the same ratio of maximal to minimal WCET and the same number of sections, but with a budget of 150000 time units assigned to the second partial task. These parameters result in a maximal section WCET of 24 instead of 16 time units, which could increase the time needed by the division function. As shown in Figure 6.1c, the measured time was indeed significantly higher than for the previous measurement, while equivalent measurements for binary search did not lead to any significant differences.

According to the results, the performance of the search algorithm using the estimation exceeds the performance of binary search for a ratio of maximal to minimal section WCET of 1.5, needs a similar amount of time for a ratio of 2, but leads to a higher overhead if this ratio increases. Aside from the variance of section WCETs, the suitability of this algorithm also depends on the behaviour of integer divisions on the current platform, so that even on the current platform, further measurements with more adverse maximal section WCETs would be required in order to make definite statements about the resulting overhead.

6.2.4.2 Average-Case Overhead for Evaluation Points

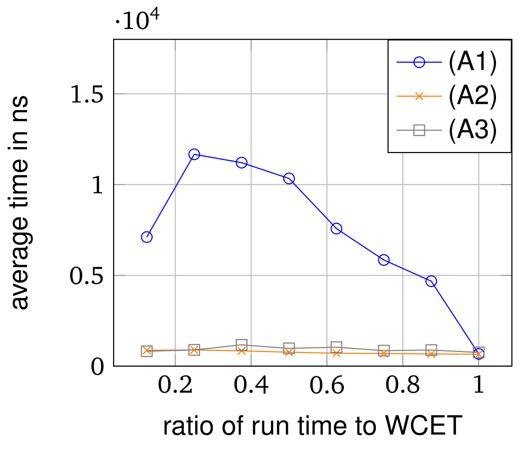
In order to approximate the average-case run-time overhead, the total overhead by all dynamic migration decisions is measured, and the number of evaluation points is counted. The number of evaluation points is interesting, since the overhead for each evaluation point can increase significantly, if address spaces for system and application are separated.

The overhead for evaluation points is measured from the call until the return of `vTaskEvaluate()`, with the time for `vTaskMigrate()`. Time-triggered evaluation points are measured from the start until the end of the respective code in the tick handler. The measured times are added for each partial task. When evaluation points are counted, the start of the task in `vEndTaskPeriod()` is also counted as evaluation point, since the first evaluation point has to be initialized. Evaluation points at which a task migrates between two cores are counted for both partial tasks.

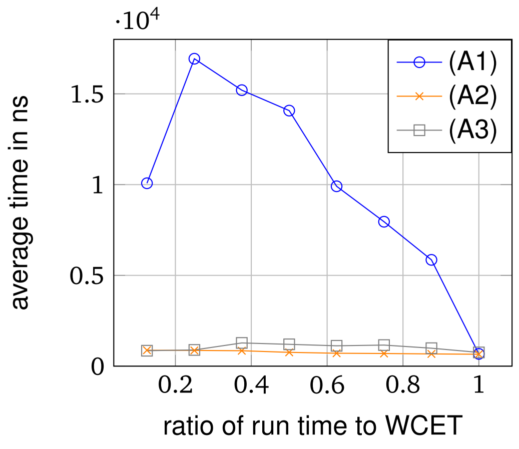
Since the behaviour of tasks with many sections and multiple migrations is interesting, the measured task is split into four partial tasks, each of which is has a budget of 10000 time units and is assigned 1000 sections. Measurements are made for ratios of maximal to minimal section length of 1.5 and 9, and for run times of $\frac{i}{8} * WCET$ with $i \in \{1, \ldots, 8\}$. For each configuration, 10 jobs are measured for each algorithm, with Algorithm (A1) using the search algorithm that uses the estimation.

The overhead for all evaluation points of a partial task is shown in Figure 6.2. As it is clearly visible, the overhead for Algorithm (A1) exceeds the overhead of the other algorithms by large, while Algorithm (A3) needs slightly more time that Algorithm (A2). With an increased ratio of maximal to minimal section WCET, the overhead of Algorithm (A1) increases significantly, which is due to the used search algorithm. Since Algorithm (A3) is more likely to need more recalculations of x_{eval} with an increasing variance of section WCETs, the overhead for this algorithm also increases. The large overhead of Algorithm (A1) can be explained by both the higher complexity of the calculation at each evaluation point, and the higher number of evaluation points.

The number of evaluation points for each algorithm is shown in Figure 6.3. For all run times smaller than the WCET, Algorithm (A1) needs the most evaluation points, which is expected, since with faster run times, the remaining budget is likely to be sufficient for multiple recalculations of x_{eval}. Algorithm (A3) needs slightly more evaluation points that Algorithm (A2), with a larger

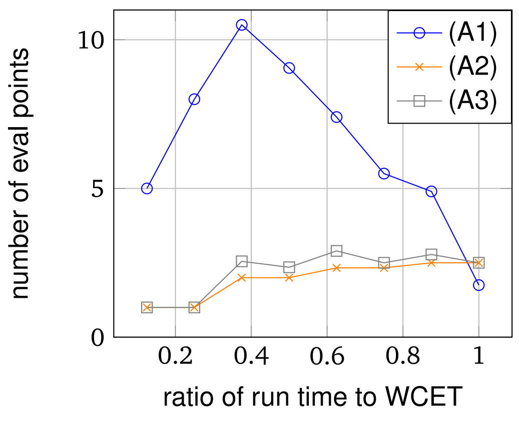


(a) Measurements for $\frac{cMax_i}{cMin_i} = 1.5$

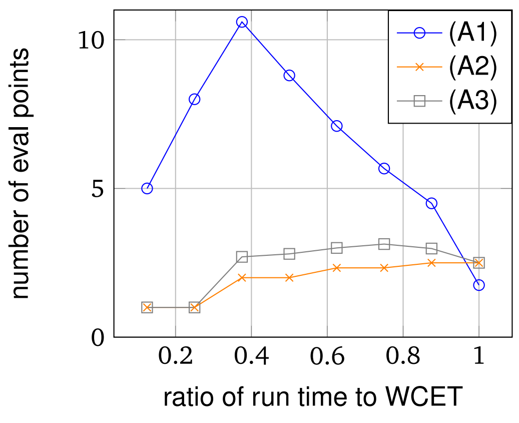


(b) Measurements for $\frac{cMax_i}{cMin_i} = 9$

Figure 6.2 – Total measured overhead dynamic migration decisions depending on run times. Measurements were made for all algorithms using evaluation points and for different ratios of maximal to minimal section WCET.



(a) Measurements for $\frac{cMax_i}{cMin_i} = 1.5$



(b) Measurements for $\frac{cMax_i}{cMin_i} = 9$

Figure 6.3 – Average number of evaluation points per partial task depending on section run times. Measurements were made for all algorithms using evaluation points and for different ratios of maximal to minimal section WCET.

difference for an increased variance of section WCETs. This is due to the larger remaining budget, when the first x_{eval} is reached. In comparison, the number of evaluation point needed by the other algorithms does not differ significantly with differences in section WCETs.

6.3 Selection of Evaluation Points

Since the goal of dynamic migration decisions is to delay migration as far as possible, the effectiveness of the given algorithms in this regard is measured by comparing the selected migration points for different algorithms. Since Algorithm (A2) is the only algorithm that chooses potentially non-optimal migration points, only this algorithm and Algorithm (A3) are measured in order to estimate the impact of the non-optimal choice.

The measured task is split into two partial tasks with a budget of 500 and 4000, respectively. A larger budget was chosen for the second partial task in order to force the task to migrate even with short run times, so that chosen mirgation points can be compared for all measured run times. Since Algorithm (A2) leads to worse results with fewer migration points and larger differences in section lengths, measurements are made for all combinations of $\frac{cMax_i}{cMin_i} \in \{1.5, 9\}$ and 20, respectively 200 sections assigned to the second partial task. For each combination, 10 measurements are made for run times of $\frac{i}{8} * WCET$, with $i \in \{1, \ldots, 8\}$

As shown in the Figure 6.4, the difference between both algorithms decreases with more migration points, although Algorithm (A2) makes non-optimal choices in both cases. With both 20 and 201 sections assigned to the second partial task, the the difference increases with an increasing variance of section WCETs.

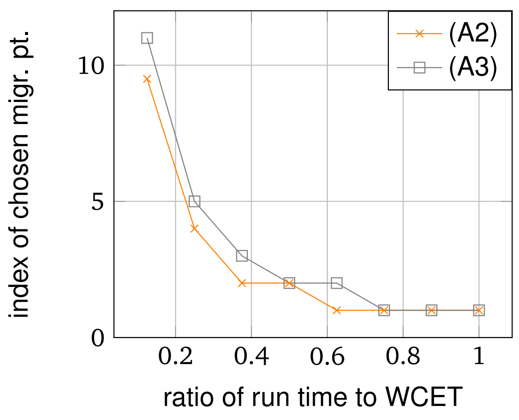
6.4 Response times

In order to determine the impact of dynamic migration decisions on the run time of a task, the response times of all algorithms are compared to the response times with fixed migration points. Fixed migration points are implemented by a call of `vTaskMigrate()` at the appropriate points in the application code.

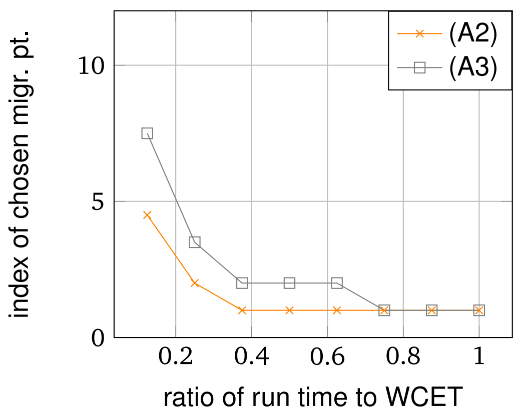
The response times are measured from the start of a job in `vEndTaskPeriod()` until its end in the same function. In order to avoid interference of the migration delay that is caused by the polling for arrived tasks in the tick handler, the time from the call until the return from `taskYIELD()` is excluded. In order to avoid an overflow of the timer register, the System Timer is used for these measurements, which provides a 64-bit counter that is incremented with a frequency of 1 MHz [Bcm].

In the previous measurements, section tun times were realized by polling for a calculated value of the current tick counter of the system. With this method, the exact run time of a section can vary with the time that has passed since the last tick interrupt. If the execution of the sections is included in the measurements, these effects can lead to section differences in section run times that depend on the overhead between sections. In order to avoid this, the implementation of sections has been changed for the measurements of response times, so that the intended run time is now approximated by decrementing an integer variable in a loop. While this method still leads to inaccuracies of the run times, these inaccuracies are now independent of the overhead of dynamic migration decisions.

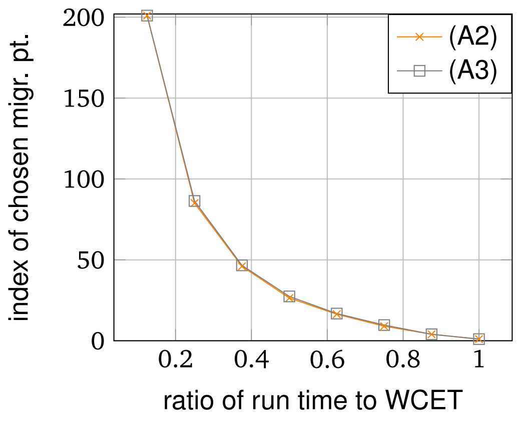
Measurements are made for two task, both of which are split into four partial tasks, with a ratio of maximal to minimal section WCET of 9. While the first task has a budget of 5000 timer ticks and 1000 assigned sections for each partial task, the partial tasks of the second task have an assigned budget of 50 ticks and are split into 10 sections. As in the previous measurements, response times



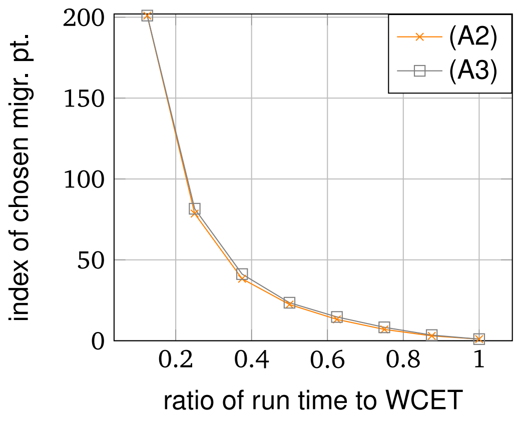
(a) 21 sections, $\frac{cMax_i}{cMin_i} = 1.5$



(b) 21 sections, $\frac{cMax_i}{cMin_i} = 9$



(c) 201 sections, $\frac{cMax_i}{cMin_i} = 1.5$



(d) 201 sections, $\frac{cMax_i}{cMin_i} = 9$

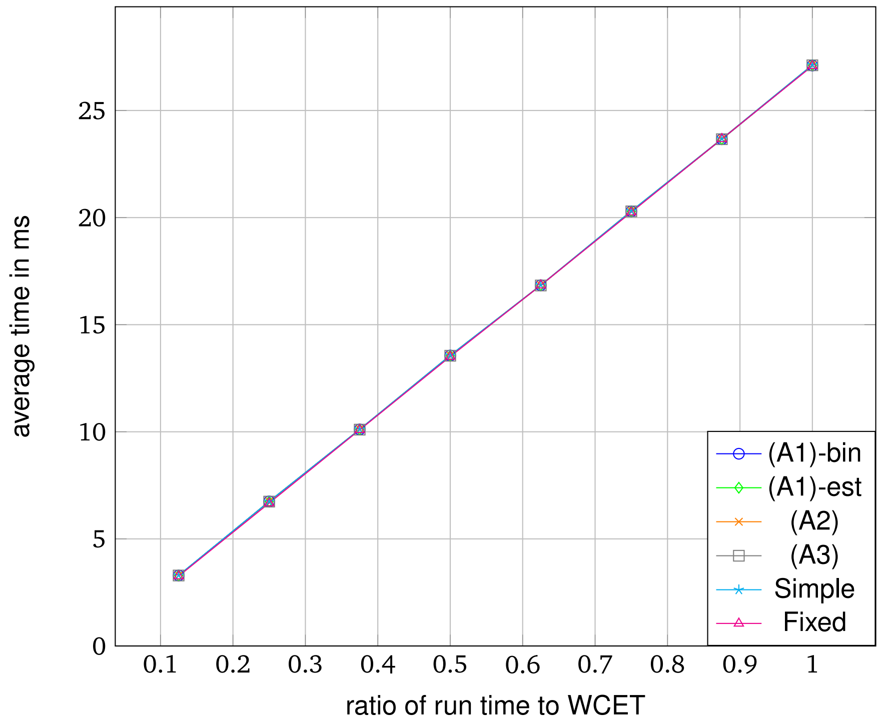
Figure 6.4 – Average chosen migration point by Algorithms (A2) and (A3), depending on run times.

are measured for different run times, in steps of $\frac{1}{8} * WCET$. For each configuration, 10 jobs are measured for each algorithm.

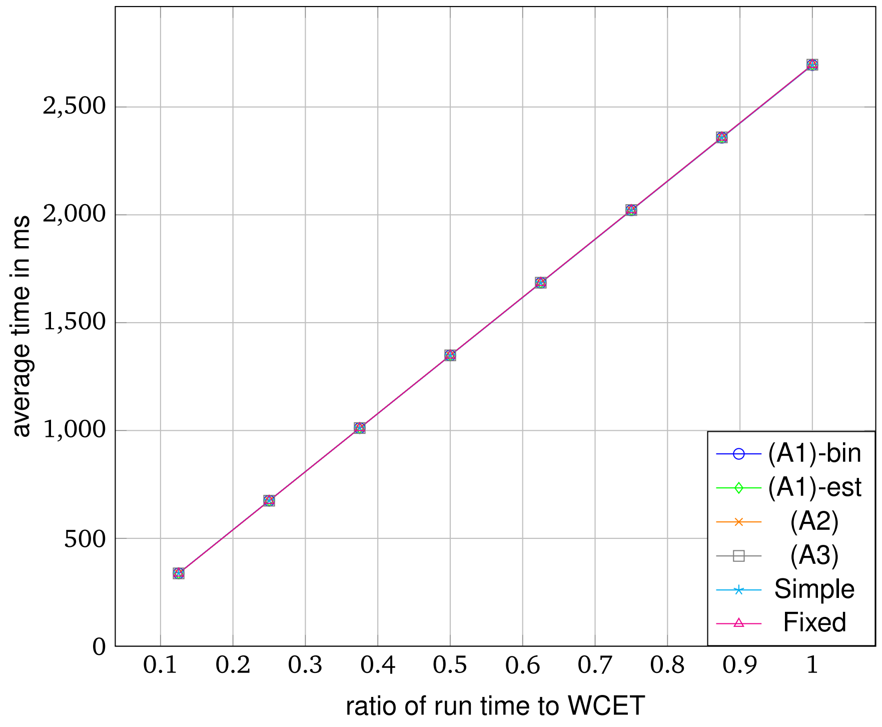
As shown in Figure 6.5, no significant differences can be determined, regardless of the number and run times of sections. Since `taskYIELD()` has been excluded from measurements, most of the adverse effects of task migration, i.e. the actual cost of data transfer, are not included in the measurements. This means that the measured results represent the overhead for dynamic migration decisions without most of the benefits of avoiding task migration. If migration overhead is included, the response times are likely to improve, if migration can be avoided by dynamic dynamic migration decisions.

As shown in Figure 6.6, at least one task migration can be avoided by all algorithms for dynamic migration decisions for run times of less than three fourth of the assigned WCETs of the example task. For run times of less than one fourth of the given WCET, both tasks could even be finished without migrating at all, with one exception of Algorithm (A2), in which a non-optimal choice leads to one task migration for the task with fewer sections.

If task migration can be avoided in more realistic scenarios with a higher load of the task set and a larger working set of the migrating task, response times are likely to be improved by dynamic

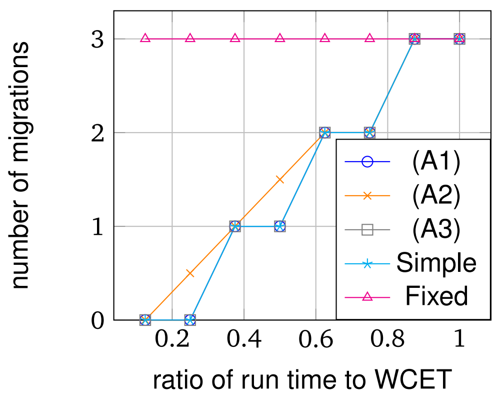


(a) Response times for all algorithms, measured for a task split in four parts, with each a budget of 50 and 10 assigned sections

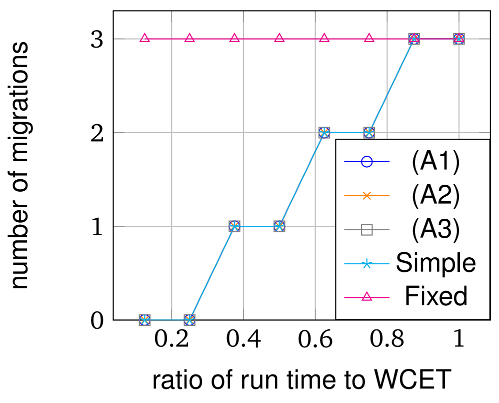


(b) Response times for all algorithms, measured for a task split in four parts, with each a budget of 5000 and 1000 assigned sections

Figure 6.5 – Response times for all algorithms, depending on run times



(a) Number or migrations for each algorithm, counted for a task split in four parts, with each a budget of 50 and 10 assigned sections



(b) Number of migrations for each algorithm, counted for a task split in four parts, with each a budget of 5000 and 1000 assigned sections

Figure 6.6 – Number of migrations for all algorithms, depending on run times. Since Algorithm (A1) has the same results for both search algorithms, these results are summarized under (A1)

migration decisions, although more comprehensive time measurements are needed in order to make more definite statements.

7 FUTURE WORK

While the presented approaches for dynamic migration decisions have been designed, analyzed and implemented, various enhancements of the current status are still possible for the presented concepts, as well as the practical implementation and the measurements.

The presented concepts rely currently on various limitations, which are not necessarily needed. As an example, the task sets for which migration decisions can be made are still restricted by the assumptions stated in Chapter 2, such as the assumption that all tasks are independent from each other. These restrictions do not necessarily reflect the behaviour of real-world applications. The implications of lifting these assumptions, and the required adaptions to dynamic migration decisions could be explored.

Furthermore, in the current approach, migration decisions for a given task ignore the run-time behaviour of all other tasks, as well as otherwise available slack. If these information were considered, it might be possible to increase the budget of the migrating task by stealing slack from other tasks, so that task migration can be delayed further. A concept for this and its implications for preserving schedulability could be designed.

Aside from theoretical concepts, the current state of the implementation still leaves room for various optimizations. Task migration, as well as time-triggered evaluation points can be implemented using interrupt mechanisms instead of polling in the tick handler, and code optimizations, such as function inlining, can be applied to the skipping of migration points.

In order to get more comprehensive results for the required overhead, the presented algorithms can be implemented and measured on different hardware platforms and operating systems, using task sets with a higher load and generally more comprehensive scenarios. Measurements could be made for platforms with a higher overhead for system calls, in order to estimate the benefits of using evaluation points. Additionally, static code analysis can be used to determine the actual overhead that needs to be included by the partitioning algorithm and dynamic migration decisions.

Furthermore, the presented approach could be combined with already existing approaches to reduce CPMD. With both migration targets and potential migration points statically known, the mechanism for software-assisted cache-line migration presented in [Sar+09] could be leveraged to further reduce migration overhead.

8 CONCLUSION

In this thesis, different algorithms for dynamic migration decisions have been presented. These algorithms are based on an already existing approach to reduce the migration overhead in semi-partitioned scheduling. In this approach, task migration is limited to a statically defined migration point, which was chosen by the partitioning algorithm out of a set of predefined potential migration points with low migration overhead. Since this point is determined statically, each migrating task always migrates at the same point, regardless of its actual run times, which makes it impossible to avoid migration, even if the current job needs less time than assigned to the current core.

In order to avoid this problem, and to prevent unnecessary migrations, dynamic migration decisions were presented in this thesis. Instead of migrating at a statically assigned point, dynamic migration decisions try to choose the latest migration point that can be chosen without leading to deadline misses. In order to guarantee that dynamic migration decisions do not lead to deadline misses, two conditions were defined, which were used to verify all presented algorithms for dynamic migration decisions.

For making dynamic migration decisions, four algorithms were defined. A simple approach was presented that needs reachability checks at each migration point, and in order to reduce the number of these checks, three further algorithms were presented that limit reachability checks to designated evaluation points. Depending on the algorithm, evaluation points are defined as points in the code, points in execution time of the current job, or a combination of both.

All algorithms were analyzed with regards to efficiency of budget use and required overhead. For all algorithms, the assigned budget can be used more efficiently with more migration points and smaller sections. With a given set of migration points, all algorithms that perform reachability checks immediately before migrating can use their budget optimally, while the quality for the approach using only time-triggered evaluation points depends on the differences between section WCETs. For all algorithms overhead needs to be considered by both the partitioning algorithm and dynamic migration decisions. This overhead was shown to be constant and relatively low. The average-case overhead at run time is at most logarithmic to the number of available migration points and is lower for algorithms that use time-triggered migration points.

A prototype for the presented algorithms was implemented on a Raspberry Pi v2, using the existing partitioned-scheduling algorithm of the FreeRTOS port piRTOS. Based on this, task migration was introduced, and all presented algorithms for dynamic migration decisions were implemented.

For this implementation, both overhead and response times were measured. According to measurement results for the given implementation, a relatively low amount of time needs to be included in the budget of each partial task for all algorithms. While the average-case overhead is significantly higher for the algorithm without time-triggered evaluation points, it is still relatively small. When the response times where measured, the impact of the additional overhead was neglible, while migration could still be avoided in many cases.

Based on the presented concepts and implementation, various improvements can be made, such as optimizations of the current implementation to further reduce the overhead, of extensions of the presented concepts, so that dynamic migration decisions can be improved or applied in more realistic scenarios.

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

REFERENCES

[ABB08] B. Andersson, K. Bletsas, and S. Baruah. “Scheduling Arbitrary-Deadline Sporadic Task Systems on Multiprocessors.” In: *2008 Real-Time Systems Symposium*. 2008, pp. 385–394.

[ABD05] J. H. Anderson, V. Bud, and U. C. Devi. “An EDF-based scheduling algorithm for multiprocessor soft real-time systems.” In: *17th Euromicro Conference on Real-Time Systems (ECRTS’05)*. 2005, pp. 199–208.

[Arm] *ARM Architecture Reference Manual*. ARMv7-A and ARMv7-R. 110 Fulbourn Road, Cambridge, England CB1 9NJ: ARM Limited.

[AT06] B. Andersson and E. Tovar. “Multiprocessor Scheduling with Few Preemptions.” In: *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA’06)*. 2006, pp. 322–334.

[Aud+00] Neil Audsley et al. “Hard Real-Time Scheduling: The Deadline-Monotonic Approach.” In: *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software* 24 (Nov. 2000). DOI: `10.1016/S1474-6670(17)51283-5`.

[Bar16] Richard Barry. *Mastering the FreeTROS™ Real Time Kernel - a Hands On Tutorial Guide*. Real Time Engineers Ltd., 2016.

[BBA10] A. Bastoni, B. Brandenburg, and James Anderson. “Cache-related preemption and migration delays: Empirical approximation and impact on schedulability.” In: *OSPERT* (Jan. 2010), pp. 33–44.

[BBA11] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. “Is Semi-Partitioned Scheduling Practical?” In: *2011 23rd Euromicro Conference on Real-Time Systems*. 2011, pp. 125–135.

[BBB15] Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor Scheduling for Real-Time Systems*. Springer Publishing Company, Incorporated, 2015. ISBN: 3319086952.

[BBY13] G. C. Buttazzo, M. Bertogna, and G. Yao. “Limited Preemptive Scheduling for Real-Time Systems. A Survey.” In: *IEEE Transactions on Industrial Informatics* 9.1 (2013), pp. 3–15.

[Bcm] *BCM2835 ARM Peripherals*. Broadcom Europe Ltd. 406 Science Park Milton Road Cambridge CB4 0WW: Broadcom Corporation, 2012.

[Bur+12] A. Burns et al. “Partitioned EDF scheduling for multiprocessors using a C=D task splitting scheme.” In: *Real-Time Systems* 48.1 (2012), pp. 3–33. ISSN: 1573-1383. DOI: `10.1007/s11241-011-9126-9`. URL: `https://doi.org/10.1007/s11241-011-9126-9`.

[DB11] Robert I. Davis and Alan Burns. “A Survey of Hard Real-Time Scheduling for Multiprocessor Systems.” In: *ACM Comput. Surv.* 43.4 (Oct. 2011). ISSN: 0360-0300. DOI: 10.1145/1978802.1978814. URL: https://doi.org/10.1145/1978802.1978814.

[DL78] Sudarshan K. Dhall and C. L. Liu. “On a Real-Time Scheduling Problem.” In: *Operations Research* 26.1 (1978), pp. 127–140. ISSN: 0030364X, 15265463. URL: http://www.jstor.org/stable/169896.

[Dor+10] François Dorin et al. “Semi-Partitioned Hard Real-Time Scheduling with Restricted Migrations upon Identical Multiprocessor Platforms.” In: *CoRR* abs/1006.2637 (2010). arXiv: 1006.2637. URL: http://arxiv.org/abs/1006.2637.

[HP09] Damien Hardy and Isabelle Puaut. “Estimation of Cache Related Migration Delays for Multi-Core Processors with Shared Instruction Caches.” In: *17th International Conference on Real-Time and Network Systems*. Ed. by Laurent George and Maryline Chetto andMikael Sjodin. Paris, France, Oct. 2009, pp. 45–54. URL: https://hal.inria.fr/inria-00441959.

[Jar19] JarvisAPI. *piRTOS*. https://github.com/JarvisAPI/Pi-RTOS. 2019.

[Kla+19] Tobias Klaus et al. “Boosting Job-Level Migration by Static Analysis.” In: *Proceedings of the 15th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. Ed. by Adam Lackorzynski and Daniel Lohmann. Stuttgart, 2019, pp. 17–22. URL: https://www4.cs.fau.de/Publications/2019/klaus_19_ospert.pdf.

[KYI09] S. Kato, N. Yamasaki, and Y. Ishikawa. “Semi-partitioned Scheduling of Sporadic Task Systems on Multiprocessors.” In: *2009 21st Euromicro Conference on Real-Time Systems*. 2009, pp. 249–258.

[LL73] C. L. Liu and James W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.” In: *J. ACM* 20.1 (Jan. 1973), 46–61. ISSN: 0004-5411. DOI: 10.1145/321738.321743. URL: https://doi.org/10.1145/321738.321743.

[LW82] Joseph Y.-T. Leung and Jennifer Whitehead. “On the complexity of fixed-priority scheduling of periodic, real-time tasks.” In: *Performance Evaluation* 2.4 (1982), pp. 237 –250. ISSN: 0166-5316. DOI: https://doi.org/10.1016/0166-5316(82)90024-4. URL: http://www.sciencedirect.com/science/article/pii/0166531682900244.

[Ras] *Raspberry Pi Model B*. https://www.raspberrypi.org/products/raspberry-pi-2-model-b/?resellerType=home. Accessed: 2020-11-28.

[Sar+09] Abhik Sarkar et al. “Push-Assisted Migration of Real-Time Tasks in Multi-Core Processors.” In: *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES ’09. Dublin, Ireland: Association for Computing Machinery, 2009, 80–89. ISBN: 9781605583563. DOI: 10.1145/1542452.1542464. URL: https://doi.org/10.1145/1542452.1542464.

[And+14] J. H. Anderson et al. “Optimal semi-partitioned scheduling in soft real-time systems.” In: *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*. 2014, pp. 1–10.

[Ber+06] S. Bertozzi et al. “Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study.” In: *Proceedings of the Design Automation Test in Europe Conference*. Vol. 1. 2006, pp. 1–6.