WCET Optimization in a Shared Bus Scenario

Bachelor-Thesis

submitted on: 27.03.2013

Computer Science at Saarland University

Name:	Maximilian John
Matriculation number:	2531835
Degree Course:	Computer Science
Study Regulations:	Bachelor Computer Science 2006
First Supervisor:	Prof. Dr. Sebastian Hack
Second Supervisor:	Prof. Dr. Jan Reineke
Advisor:	Michael Jacobs

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit mit dem Titel "WCET Optimization in a Shared Bus Scenario" selbständig, ohne fremde Hilfe und ohne Benutzung andere als der von mir angegebenen Quellen angefertigt zu haben. Alle aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche gekennzeichnet. Die Arbeit wurde noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Saarbrücken, den 27.03.2013

Maximilian John

Abstract

In time-critical multi-core systems with shared buses we cannot apply the same worst-case execution time (WCET) analyses as with dedicated resources. This is due to additional blocking time occuring through bus interferences. We consider a setting with TDMA bus arbitration. The optimization of the wcet depends on two parameters: system schedule and bus schedule. None of the parameters can be optimized in isolation. We try to speed up the exhaustive optimization as well as to provide good heuristics.

Contents

Chapter 1

Introduction

1.1 Motivation

1.1.1 Timing analysis

In many practical applications of computer science, timing analysis plays an important role. Often times, there is someone who wants to know how fast a computer can deliver certain results, i.e. how fast a program terminates. When we talk about *execution time(et)* in the following, we mean exactly this. But we might not only be interested in this particular time because when we talk about the *execution time* of a program or a part of a program, this only refers to the time this processor spends to execute the particular program. But often if we consider a whole system, i.e. multiple parts of programs, then we do not only want to know how long the execution of every single part lasts, but also absolute points time when a calculation is requested and when it terminates. We call those two events *release* and *termination time*. Furthermore, we are interested in the difference between *release* and *termination* which is called *response time(rt)*. Obviously, to holds that

$$rt = release - termination$$

To illustrate the difference between those terms, consider figure 1.1. We can see two program parts τ_1 and τ_2. τ_2 can only start its computation when τ_1 is finished. That's why it cannot start directly after its release and thus execution and response time are different.

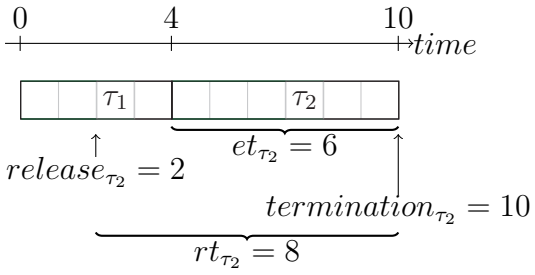


Figure 1.1: Timing terminology

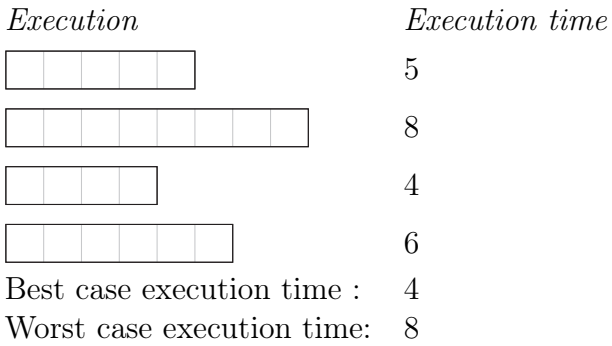


Figure 1.2: All possible execution times

There have been many approaches in the last decades to make timing analysis more and more precise. But how does timing analysis work conceptually? If you consider a program or at least a program part, you have to consider all the possible behaviour which might occur during execution. So often, there is not only just one possible time when we talk about how fast a program terminates. This is what makes the analysis so hard, we mostly have to predict a time interval where all the possible execution times are located in. Therefore, we can talk about the best and worst possible execution time as illustrated in figure 1.2. In this example, all possible execution times are listed, thus we can determine the best and the worst case. The commonly used names are *best-case execution time(BCET)* and *worst-case execution time(WCET)*. Analogously, we will further talk about *best-case response time(BCRT)* and *worst-case response time(WCRT)*. The analysis of the $WCET$ is a widely targeted challenge [cf. WEE$^+$08].

The most important field of application of timing analysis in general is the analysis of safety-critical applications. Saftey-critical applications often have critical constraints which must not be hurt by any possible execution of the program. In time-critical applications, those constraints relate to time deadlines. This means that an action must be happen within a given amount of time. We will futher call those actions *tasks*. For example, if we consider an air bag in a car, one might want the air bag to unfold within a few milliseconds. Such problems are tailor-made for the timing analysis. We want to determine if the $WCRTs$ of the considered tasks do nut exceed the deadlines. But we will define this problem more exactly in chapter 2.

1.1.2 Multi-cores

In today's computer systems, we often do not find just one single processor-core. As we want our systems to be more performant, we often use several processor-cores. One way is to take several independent single-core processors and let them run simultaneously to achieve a higher performance. But in this case, the construction is not only space-consuming, but also expensive. Thus, the common way is to use a multi-core system which means that you have several processors which share some resources such as data buses or shared cashes. The general concept of a multi-core system is shown in simplified terms in figure 1.3. We will take a closer look at the

effects of bus sharing. We aim to prevent data from being corrupted when several cores try to access the bus at the same time, thus we have to introduce an arbiter for this bus. The task of this arbiter is to grant or permit the bus access to its supplicants so that there is only one bus access at a time. When two processing units try to access the shared bus at the same time we call this behaviour *conflict*. Shortly summarized, the arbiter has to solve those conflicts.

There are several ways to do this. One could define rules statically which means that before the system is started we determine which processor may access the bus at each point in time. The advantage of this so called *Time Division Multiple Access(TDMA)* approach is that we can analyze each processor in isolation because we do no care about the behavior of the rest of the system. The obvious disadvantage is that we statically have to decide for a schedule. And finding an optimal *TDMA* schedule is a non-trivial challenge [cf. WT06] [HE05]. An alternative approach is the field of dynamic arbitrations where we have well-known approaches like *First-Come-First-Serve* or *Round Robin*. The advantage regarding such dynamic arbitrations is that we always grant access to a processor which really wants to access the bus. This is not true for static arbitrations because we can decide an arbitration without looking at actual resource situations. This property is called *work-conserving* [cf. FKY08]. But on the other hand, dynamic arbitration makes the analysis much harder because we have to consider all processors in parallel to look at all possible interferences. We decided to concentrate on the TDMA setting.

The single-access policy is obviously a disadvantage of a multi-core system as one core possibly must wait accessing the shared bus because the access is not granted. We call this characteristic *interference*. In single-core systems we do not have such interferences.

Similar interferences can also appear in shared caches. Such shared resource interferences are the reason why single-core timing analysis is not sound for multi-core systems any more because they are not taken into account. The main goal of the analysis of complex single-core systems [as GRG11] [WAB+10] [TFW00] is the determination of upper bounds on the $WCET$. A multi-core analysis [as PSC+10] mostly takes the results of a single-core analysis as input and concentrates on the effects of interferences. This is also what we want to do in this thesis. We will only concentrate on the interferences with the bus and ignore caches at all. According to *Amdahl's law* [cf. Amd67] the speedup of a multi-core system compared to a system with one single core strongly depends on the time which cannot be parallelized, so in our scenario the time where several processing units want to access the bus. Figure 1.4 illustrates this calculation. We see that it is very benificial to keep the non-parallelizable time low which will be one of our goals described in the next chapter. Several measurements of the average-performance of multi-core systems [as RGG+12] underline this impression. The consequence is that one has to adress this contention for shared resources [cf. ZBF10] [FSS07].

The thesis will be organized as follows: Chapter 2 gives basic definitions of our system model and describes our optimization problem. Chapter 3 presents some techniques to speed up the exhaustive optimization whereas chapter 4 evolves some heuristics. Finally, chapter 5 provides a brief outlook and chapter 6 summarizes.

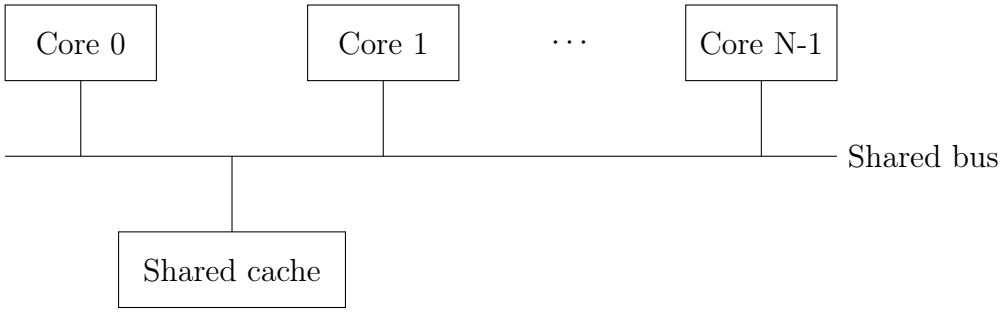


Figure 1.3: Concept of a multi-core system

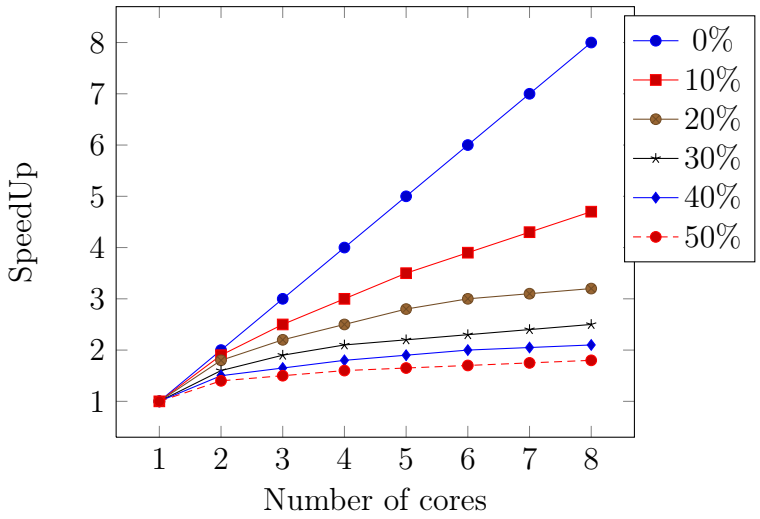


Figure 1.4: Amdahl's law effect

Chapter 2

System Model

We assume a system which behaves exactly as described by the model in this chapter. Later, we will see which properties a concrete system has to fulfil in order to fit into this model. We consider a system model with N equal processing units $PU = \{P_0, \cdots, P_{N-1}\}$ where $N > 1$ and TDMA bus which is shared between all processing units. Figure 2.1 illustrates our view of the model.

2.1 Tasks

Let $Tasks$ be the set of available tasks. In our model a task $\tau \in Tasks$ is described by its naive execution time and its bus accesses σ_i. As an example consider Figure 2.2(a). We see a task with a naive WCET of 6 and 2 accesses. The first access is released at time 1 and has a naive length of 1 whereas the second access is released at time 3 and has a naive length of 2. We will call the execution time $WCET$ although we assume that there is only one execution time to stay consistent with other approaches. To describe an access, we have to give information about its naive offset relative to the start of the task and its naive length relative to its own start. All those values are called *naive* because they are only safe in a single-core setting, so they do not consider any possible bus interferences. We want to represent this information in a more compact way. Therefore, we will now define some shortcuts for the important values. Let τ be a task. Then we can get the naive $WCET$ through $\tau.nWCET$. We can get the boolean value describing if the task is accessing the bus at a its relative position *pos* by using the shortcut $\tau.acc(pos)$. Figure 2.2(b) illustrates the usage of this shortcut.

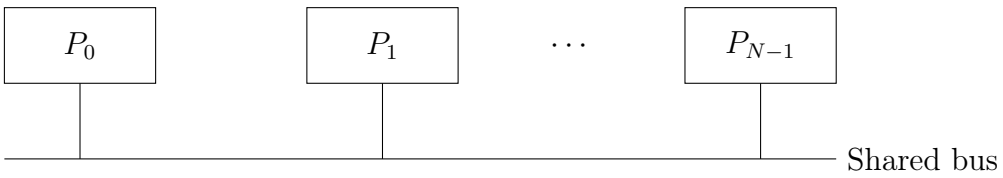
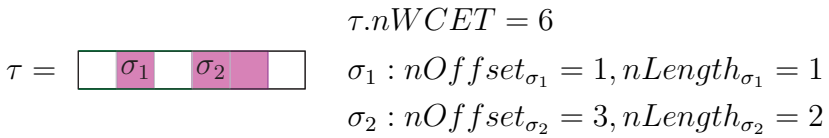


Figure 2.1: Conceptual view of the model

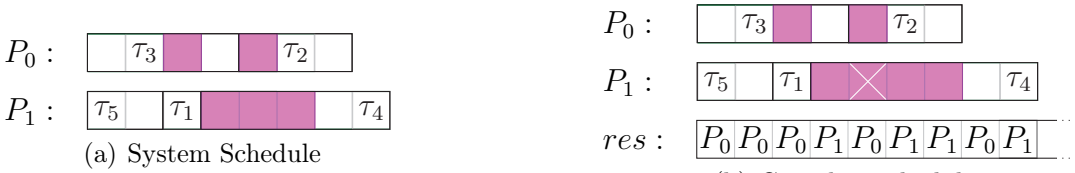


(a) A task with two accesses

$$\begin{aligned}\tau.acc(0) &= false\\ \tau.acc(1) &= true\\ \tau.acc(2) &= false\\ \tau.acc(3) &= true\\ \tau.acc(4) &= true\\ \tau.acc(5) &= false\end{aligned}$$

(b) Shortcuts for accesses

Figure 2.2: Task example



(a) System Schedule

(b) Complete schedule

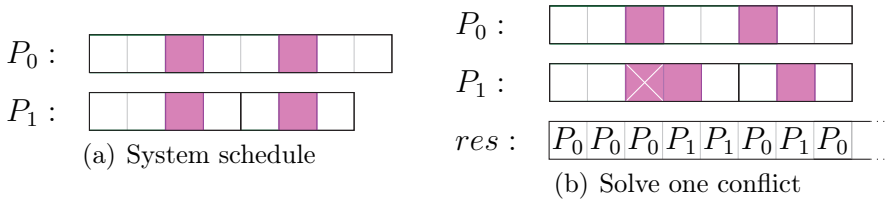
Figure 2.3: Schedule examples

2.2 System schedule

Intuitively, the system schedule describes which processing unit executes which tasks in which order. So more formally, let S be the set of all system schedules. A system schedule $sys \in S$ is identified by two properties. At first, a task assignment $\rho : Tasks \to PU$ which gives information about which task is executed on which processing unit and secondly, a task order per processing unit. To access this task order, we define the values $sys.pre(\tau)$ and $sys.succ(\tau) \in Tasks \cup \{null\}$ which delivers the predecessor or the successor of the task τ under sys, respectively. If a task has no predecessor or no successor, the corresponding shortcut will return $null$. Implicitely, we assume that all tasks are released at time 0. As example for a system schedule consider figure 2.3(a).

2.3 Resource schedule

Let R be the set of all resource schedules. A resource schedule $res : \mathbb{N} \to PU, res \in R$ is an assignment from time to processing units. So for every point in time $t \in \mathbb{N}$ the processing unit $res(t)$ is allowed to access the bus. This means that all other processing units which try to access the bus have to wait until they are scheduled. So for the example of figure 2.3(b) this means that when we block a task once we delay every following "square" on the same processing unit one slot to the right. We can do this because we assume that the tasks' accesses allow preemption. So if we have



(a) System schedule

(b) Solve one conflict

Figure 2.4: Timing Compositionality

an access with the length of 3 for example, and the access is blocked in the middle, it will not have to start from its very beginning again, but we can just restart the access at the point where it stopped. This blocking is denoted with a white cross in figure 2.3(b). We also make the assumption of timing compositionality. We assume the time between two accesses as fixed. When a task is blocked because it has an access at the current time, it may not occur that the task can already start with a later time block where it has no access. We will make this clear in an additional example as there does not exist a formal definition of timing compositionality so far [cf. WGR+09]. Nevertheless we want to assume it for our system. So consider the system schedule in figure 2.4(a). If we now add the resource schedule in figure 2.4(b), then the whole rest of the task and of course also tasks which come later on the same processing unit are shifted one block to the right.

From now on, if we talk about a schedule, we always mean a pair of system and resource schedule.

2.4 WCET

As already proposed in the motivation section, the $WCET$ of a system - or now better said the $WCET$ of a schedule - is the worst possible execution time of this system. If we consider a concrete schedule, we could also talk about the *execution time* of a schedule because all tasks have a fixed naive execution time. But as we want to stay consistent with related work and with further presented approaches, we will talk about the $WCET$ here, too. Let (sys, res) be a schedule, then we can calculate the $WCET$ with respect to sys and res with the following formulas. First, we have

$$exec_{sys,res} : Tasks \times \mathbb{N} \times \mathbb{N}$$

$$exec_{sys,res}(\tau, pos, t) = \begin{cases} 0 & \text{if } pos \geq \tau.nWCET \\ 1 + exec_{sys,res}(\tau, pos, t+1) & \text{if } pos < \tau.nWCET \\ & \wedge \tau.acc(pos) \\ & \wedge res(t) \neq sys.\rho(\tau) \\ 1 + exec_{sys,res}(\tau, pos+1, t+1) & \text{else} \end{cases} \tag{2.1}$$

Equation (2.1) describes the situation when the cycle pos of the task τ is next to be executed at time t in the current setting of sys and res. If the task is finished,

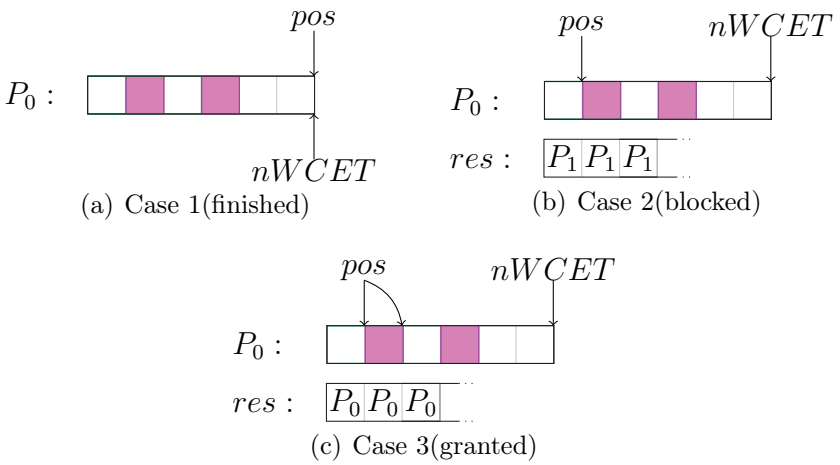
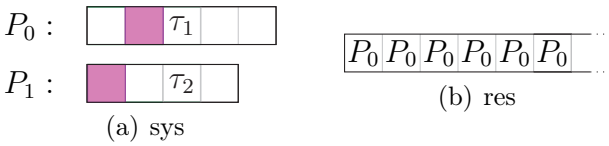


Figure 2.5: $exec_{sys,res}$ with an upcoming bus access



$$exec_{sys,res}(\tau_2, 0, 0) = 1 + exec_{sys,res}(\tau_2, 0, 1) = 1 + 1 + exec_{sys,res}(\tau_2, 0, 2) = \cdots = \infty$$

Figure 2.6: Infinite execution time

so the position has already reached the naive WCET of τ, then of course we do not add anything to the execution time of the task. Otherwise, if the task has an access and this access is not granted by res, the time grows by 1 whereas pos stays the same because the task is blocked. In the last case, there is either no access or the access is granted, so both t as well as the pos are increased by 1. Figure 2.5 gives an additional hint how to understand the parameters. Of course, this recursive formula does not need to terminate as we can see in figure 2.6 where $res(t) = P_0$ for all $t \in \mathbb{N}$. We can now use this formula to compute the $WCET$ as well as the $WCRT$ of a whole task.

$$WCET_{sys,res}(\tau) = \begin{cases} exec_{sys,res}(\tau, 0, 0) & \text{if } sys.pre(\tau) = null \\ exec_{sys,res}\left(\tau, 0, WCRT_{sys,res}\left(sys.pre\left(\tau\right)\right)\right) & \text{else} \end{cases} \tag{2.2}$$

$$WCRT_{sys,res}(\tau) = WCET_{sys,res}(\tau) + \begin{cases} 0 & \text{if } sys.pre(\tau) = null \\ WCRT_{sys,res}(sys.pre(\tau)) & \text{else} \end{cases} \tag{2.3}$$

We see that we have a nested recursion to compute both values. Furthermore, we recognize that the bus and the processing units start at time 0 and are additionally

Note 2.1. Related Work
There is already another system model [in PSC+10] adressing a similar problem. The authors of this work describe tasks as *superblocks*. A *superblock* is described by the amount of computation time and the amount of bus access time, but one does not know the actual distribution of computation and accesses inside a block. If we want to adapt our system model to theirs, we would have to construct an own superblock for each access and a superblock for each pure computation time. But the main focus of their work lies on the analysis of upper bounds on the $WCET$ [cf. SPC+11] [SCT10]. Thus they are more interested in superblocks containing both computation and accesses.

synchronized. We also see that the starting time of a task is exactly the $WCRT$ of its predecessor if it exists - 0 otherwise. To compute now the $WCET$ of one processing unit, we need to sum up the $WCETs$ of all tasks which are executed on the processing unit under consideration. This brings us to the following equation.

$$WCET_{sys,res}(p) = \sum_{\tau \in Tasks | sys.\rho(\tau))=p} WCET_{sys,res}(\tau) \tag{2.4}$$

$$WCRT_{sys,res}(p) = max\{WCRT_{sys,res}(\tau) | \tau \in Tasks, sys.\rho(\tau)) = p\} \tag{2.5}$$

To compute the $WCRT$ of a processing unit, it would be also enough to take the $WCRT$ of the last task on this processing unit. We have to mention here that $WCET$ and $WCRT$ are always identical in our scenario. Finally, we only need to take the maximum over all processing units to get the $WCET$ or $WCRT$ of the whole system.

$$WCET_{sys,res} = \max_{p \in PU}\{WCET_{sys,res}(p)\} \tag{2.6}$$

$$WCRT_{sys,res} = \max_{p \in PU}\{WCRT_{sys,res}(p)\} \tag{2.7}$$

2.5 Challenge

We have the number of processing units and the set of tasks as constant system parameters. Our goal now is to optimize the WCET of the whole system. The part of the system we can vary is the pair of system and resource schedule. This means our overall goal in all further contexts will be to find an optimal schedule pair which minimizes the $WCET$ of the whole system. Formally, we have to find sys_b, res_b such that

$$WCET_{sys_b,res_b} = min\{WCET_{sys,res} | sys \in S, res \in R\}$$

2.5.1 Cyclic dependency between system and resource schedule

If we want to optimize the system schedule in isolation, we have to know the actual WCETs instead of the naive ones which depend on the additional blocking time which further depends on the resource schedule. So we need to know the resource schedule to construct an optimal system schedule. But on the other hand, if we try to optimize the resource schedule in isolation, we need to know the actual starting times of tasks and accesses which again depend on the system schedule. So we have a cyclic dependency wherefore we have to optimize both parameters at one go.

Chapter 3

Optimization Algorithm

3.1 Complexity

In this section we want to prove that the problem we have to solve is NP-hard. The starting point of our reduction chain will be the *Partitioning*-problem as this is NP-hard [after GJ90]. See the reduction chain in figure 3.1.

This surely needs some further explanations.

1. If we restrict Multiprocessor-Scheduling to two processors and additionally to the minimal deadline, we exactly have the Partitioning-problem. The minimal deadline means here the sequential execution time divided by the number of processors, so 2 in this case. As Multiprocessor-Scheduling contains an NP-hard problem, it is at least as hard as the contained problem.

2. If we additionally consider a shared bus, we can just restrict this problem by allowing no accesses. Then we have just the Multiprocessor-Scheduling problem.

3. The third edge makes the step from a decision problem to a optimization problem. Whenever we have an NP-hard decision problem, the according optimization problem is already NP-hard [after GJ90]. This is due to the fact that we can solve our decision problem when we know the optimal possible value.

4. The last step just addes an additional task to a problem which is already NP-hard. We do not only want to know the optimal WCET of a system, we are also interested in a schedule which leads to this WCET. If we know this optimal schedule, we can easily determine the WCET of it and obtain the optimal WCET like this. So we already solved the smaller problem.

Now we have proven that it is NP-hard to find an optimal schedule. But every step in the reduction chain was more or less a generalization. The second edge for example restricted our problem to the case where no accesses are available. But this is actually a scenario we are not interested in, so the proof aims exactly at the part of the problem which we do not consider. Thus we are not happy with this

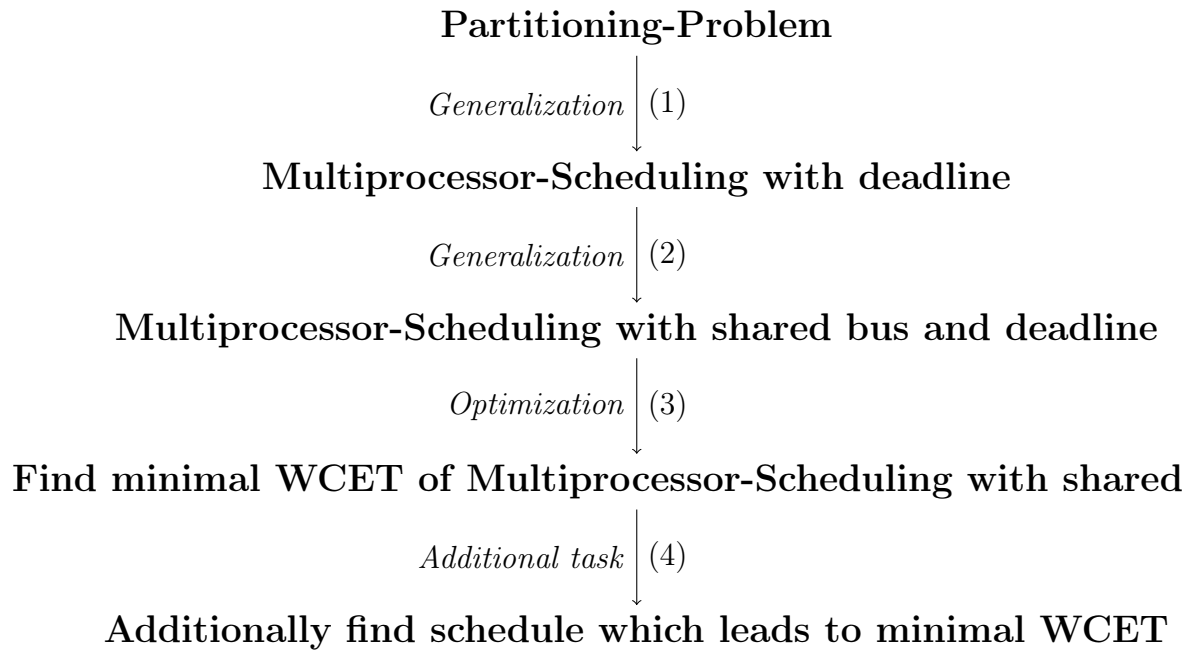


Figure 3.1: NP-hardness reduction chain

reduction, but a proof which includes more parts of our whole problem would grow beyond the bounds of this work. But we do not differ from the community with this NP-hardness proof as the step from Partitioning to Multiprocessor-Scheduling is also a strong restriction to only two processors and one particular deadline. This proof is also done [in GJ90] like this. As it common, we assume that a big part of our problem is NP-hard although we only really showed it for a small part.

3.2 Conceptual approach

We defined the setting as well as the challenge. We want to find sys_b, res_b such that

$$WCET_{sys_b,res_b} = min\{WCET_{sys,res} | sys \in S, res \in R\}$$

This formula can be translated straight-forward into Algorithm 1.

Data: S: Set of all system schedules, R: Set of all resource schedules
1 $(sys_b, res_b, WCET_b) \leftarrow$ (null,null,∞);
2 **foreach** $(sys, res) \in S \times R$ **do**
3 **if** $WCET_{sys,res} < WCET_b$ **then**
4 $(sys_b, res_b, WCET_b) \leftarrow (sys, res, WCET_{sys,res})$;
5 **end**
6 **end**
7 **return** $(sys_b, res_b, WCET_b)$;

Algorithm 1: Conceptual Approach

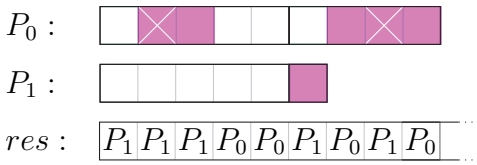


Figure 3.2: "Bad" resource schedule

We recoginize that there are still several issues we should resolve. The first one really prevents us from imlementing the algorithm as it is now and the other ones should just be solved to inhibit stupid decisions.

Issue 1: Infinity As a resource schedule is a function $res : \mathbb{N} \to PU$, we have to consider infinitely many possible resource schedules. Additionally we saw that the execution time of a task can be infinite. Now it must be our goal to restrict the search space of the resource schedules while still keeping optimality. A first approach could be to throw away all resource schedules which do not complete the system within an upper bound $WCET_b^{UB}$ of $WCET_b$. We can do this as we can always find a schedule where the system terminates in a finite amount of time. A possible upper bound would just be the sequential execution of all tasks. So we can restrict res to $res : \{0, \cdots, WCET_b^{UB} - 1\} \to PU$ where $WCET_b^{UB} = \sum_{\tau \in Tasks} WCET_{naive}(\tau)$. As we have a finite number of tasks with each a finite naive $WCET$, we know that $WCET_b^{UB} < \infty$.

Issue 2: Non-reasonable resource schedules In Figure 3.2 we see a resource schedule we actually also do not want to take into consideration when looking for an optimum. This resource schedule is bad because at time 2 P_0 requires the bus and a processor (here P_1) which does not have an access at 2 is scheduled. Formally, we have the following situation:

$$\exists t \in \mathbb{N} : res(t) = P_1 \wedge P_1 \text{ has no access at } t \wedge P_0 \text{ has an access at } t \tag{3.1}$$

At the very beginning, we stated that a big advantage of dynamic bus arbitration is the *work-conserving* property. We would now like to define constraints to our TDMA resource schedule which restricts schedules that hurt this property. To achieve this we negate a generalization of statement (3.1) which described the bad behaviour and construct a condition out of it:

Definition 3.1. *Reasonability*
A resource schedule res is *reasonable* if and only if $\forall t \in \mathbb{N}$:
$res(t) = P_i \Rightarrow$ P_i has an access at time $t \vee$
$\forall P_j \in PU \setminus \{P_i\}.P_j$ has no access at time t

Lemma 3.1. *Let sys be a system schedule.*
If there is a resource schedule res with $WCET_{sys,res} = K \in \mathbb{N}$*, there is a reasonable resource schedule* res_r *with* $WCET_{sys,res_r} \leq K$*.*

Proof. With the following algorithm, we can construct a reasonable schedule:

1. If res is reasonable, terminate
2. Assume, res is not reasonable
   $\Rightarrow \exists t \in \mathbb{N}.res(t) = P_i \wedge \exists P_j \in PU, i \neq j.$
   P_j tries to access the bus and P_i not.
3. Construct res_n as follows:
4. Take the first $t \in \mathbb{N}$ where point 2 holds
5. $res_n(t') = res(t') \forall t' \neq t$
6. $res_n(t) = P_j$ (P_j as described above)
7. Start at point 1 with res_n

Now it is still to be shown that this procedure terminates and does not enlarge the WCET. At point 2 res is *reasonable* until time $t-1$ and at point 6 res_n is *reasonable* until time t. Furthermore as P_j is scheduled one time slot more and the rest of res did not change, we have $WCET_n(P_j) \leq WCET(P_j)$. All other processing units are not affected as they either do not need the bus at time t or they were not scheduled anyway under res. The rest of res_n is equal to res, so we have $\forall i \neq j.WCET_n(P_i) = WCET(P_i)$. This means that our new schedule is reasonable for a longer time for the start (at least 1 cycle more) and the new $WCET$ has not increased, so the algorithm will terminate as we consider K to be a natural number. After termination reasonability holds because this is the termination criterion. □

We recognize that we solved the infinity-issue with reasonability, too. If a schedule does not terminate within $WCET_b^{UB}$, then there was a non-reasonable blocking. This means, we can never construct a reasonable schedule which leads to a $WCET$ which is greater than $WCET_b^{UB}$. This criterion gives us a hint on how to iteratively construct all reasonable resource schedules given a system schedule. We just have to solve conflict for conflict which can be illustrated by a conflict tree shown in figure 3.3.

We also see in this conflict tree that such a reasonable resource schedule including question marks describes many concrete reasonable resource schedules. Thus, we call the constructed resource schedules *idealized* as we can replace any question mark with an arbitrary processing unit and still obtain a reasonable resource schedule. It does not matter which processing unit to schedule at a point in time where we know that noone wants to access the bus. One can imagine that if the system is terminated, we do not need to schedule any resources, either, so at this point there are also only question marks in the schedule. A nice feature of this construction approach is that we do not have to calculate the $WCET$ of a schedule from scratch

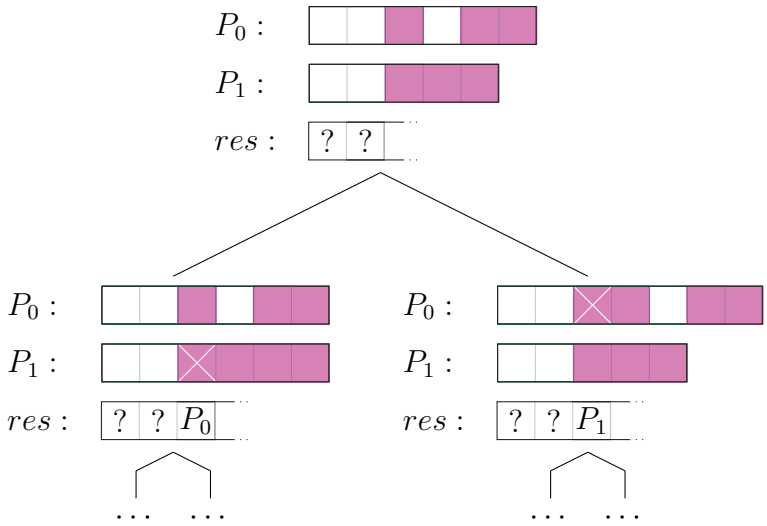


Figure 3.3: Conflict tree of resource schedule construction

again as the conceptual computations are the same. The blocking or non-blocking of a task corresponds exactly to the two cases in the $exec_{sys,res}$-formula. More exactly, we can interpret the conflict tree such that every leave represents a finished idealized resource schedule and the $WCET$ is just equivalent to the height of the path we have to go to this schedule. This more detailed construction is sketched in figure 3.4.

Issue 3: Redundant system schedules As already mentioned at the beginning, we consider all processing units to be equal. Formally, we can define the relation $\sim$ on system schedules.

Definition 3.2. $\sim$
Let sys_1, $sys_2 \in S$ be two system schedules. We define that $sys_1 \sim sys_2$ if and only if there is a bijective processing unit renaming function $\pi : PU \to PU$ such that $\pi(sys_1) = sys_2$

Lemma 3.2. $\sim$ *is an equivalence relation.*

Proof. This obviously holds as the renaming functions are permutations on the set PU, so there exist the identity, the inverse function and the permutations are closed under composition. □

Now we can derive the following theorem.

Theorem 3.3. *Let $sys_1, sys_2 \in S$ be system schedules with $sys_1 \sim sys_2$, then they lead to the same locally optimal $WCET$.*

Proof. Let res_1 be the optimal resource schedule for sys_1. As $sys_1 \sim sys_2$, there is a renaming function π such that $\pi(sys_1) = sys_2$. Now we construct res_2 such that

$$\forall t \in \mathbb{N} : res_2(t) = \pi\left(res_1(t)\right)$$

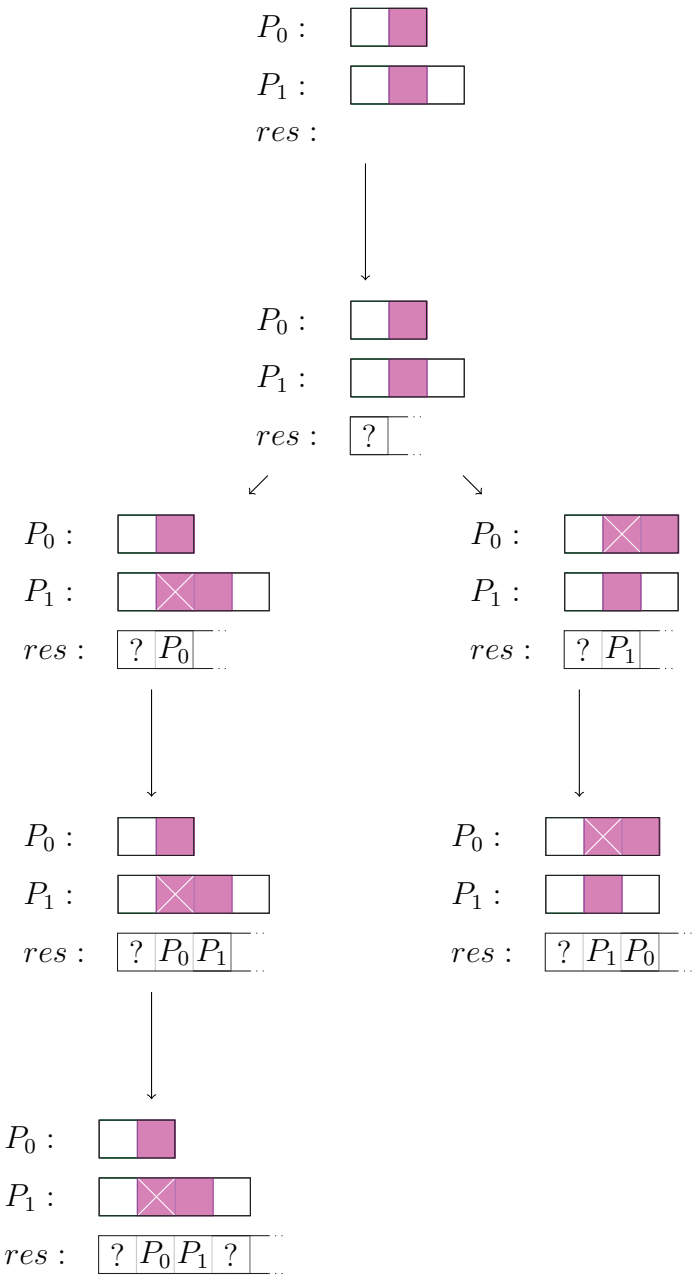
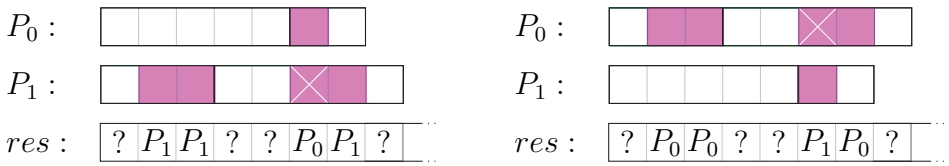


Figure 3.4: Detailed construction of reasonable resource schedules



(a) Schedule A (b) Schedule B

Figure 3.5: Redundant system schedules

Obviously it holds:

$$WCET_{sys_2,res_2} = WCET_{sys_1,res_1}$$

□

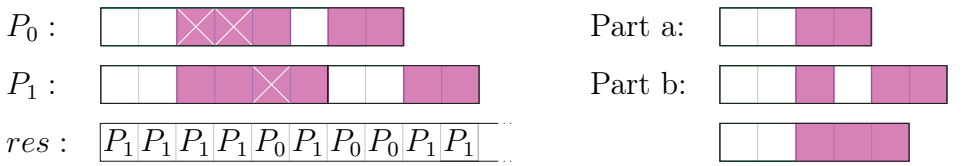
So if we have two schedules where just the processing unit names are swapped as in figure 3.5, we do not have to take both into consideration when looking for the optimum. One system schedule of each equivalence class is enough to find the optimal schedule. This brings us to the idea that we can split the construction of all possible (non-redundant) system schedules into two parts. At first we construct all possible task partitionings with up to N partitions to define which tasks are executed on the same processing unit. The concrete assignment from partition to processing unit can then be done randomly. Additionally, we need all possible task orders for each partitioning to define the execution order on the processing unit. If we build our system schedule like this, we always construct one system schedule per equivalence class.

With all this knowledge, we can derive a first reasonable approach as can be seen in Algorithm 2.

```
   Data: Tasks: set of Tasks, n: number of processing units
 1 (sys_b, res_b, WCET_b) ← (null,null,∞);
 2 P ← allPartitionings(Tasks, n);
 3 foreach part ∈ P do
 4 |   S ← nonRedundantSystemSchedules(part);
 5 |   foreach sys ∈ S do
 6 |   |   R ← reasonableResScheduleRepresentants(sys);
 7 |   |   foreach res ∈ R do
 8 |   |   |   WCET ← WCET_{sys,res};
 9 |   |   |   if WCET < WCET_b then
10 |   |   |   |   (sys_b, res_b, WCET_b) ← (sys, res, WCET);
11 |   |   |   end
12 |   |   end
13 |   end
14 end
15 return (sys_b, res_b, WCET_b);
```

Algorithm 2: First reasonable approach

But now, let us get more concrete regarding the building of system and resource schedules.



(a) Currently best schedule(WCET 10)

(b) Next regarded partitioning(naive WCET 11)

Figure 3.6: Prune a partitioning

3.3 Traversing the search space

In the conceptual approach, we assumed the sets of possible system and resource schedules as just given. In the reasonable approach, we suggested how and where we compute those sets. But in the actual implementation we do not want to precompute the whole set of system schedules for example because this would just be much too space-consuming. That's why we now want to talk about iterators because we want to define how we can iteratively build this set and talk about the traversing order.

3.3.1 Exhaustive iterators

The trivial way to iterate the search space would just be to compute one object at a time and go one step in the optimization loops deeper. We call this iterator $Iter^{ex}_{part}$, $Iter^{ex}_{sys}$ and $Iter^{ex}_{res}$ for the respective objects of the algorithm.

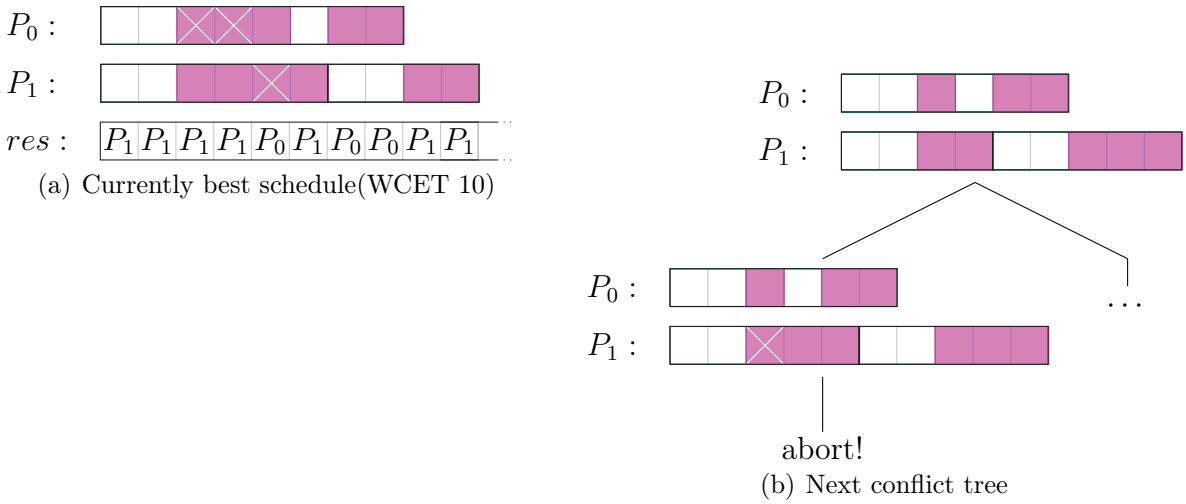
We already talked about the construction of the resource schedules with the conflict tree, but we did not say anything yet about how our implementation builds the partitionings and system schedules.

There are several possibilities to enumerate all possible partitionings of a set of n objects. We use an algorithm which starts with the partitioning which contains only one partition with all objects. In further steps, the partitionings become finer and finer. This has the advantage that when we have two schedules leading to the same $WCET$, we prefer the one with the coarser partitioning as we regard this first. This means that we possibly achieve a lower number of processing units.

3.3.2 Lower bound operators

As you might already imagine we consider many needless schedules applying this method because some system schedules or even some partitionings are already guaranteed to result in worse WCETs than the currently best found complete schedule. Also some resource schedules might fulfil this property so that we can prune whole subtrees of the conflict tree. So we will take a closer look on that now.

For both iterators we can find points where computations can be pruned. Starting with the system schedule we see that whenever the naive WCET of a partitioning(the maximum of the naive WCET sums of tasks in one partition) reaches the currently best WCET value, we do not have to compute all task permutations because we already know that we cannot achieve better than the naive WCET. An



(a) Currently best schedule(WCET 10)

(b) Next conflict tree

Figure 3.7: Prune a subtree

example for this is shown in figure 3.6 Like this, we do not only save the computation of the system schedules, but also the according computation of all resource schedule representants belonging to those system schedules.

For the resource schedule iterator, we can use a classical branch-and-bound technique to step through the tree. So whenever we recognize that during the building of a resource schedule the current height of the tree reaches the currently best WCET, we do not traverse the conflict tree further down, so we prune the whole subtree. See an example for this in figure 3.7. We also see that we can reject subtrees much earlier when we add the naive rest to the current height of the conflict tree. This will also give us a hint on how big the outcoming WCETs will be at least.

These mentioned thechniques do not only work for the naive WCET, but for all lower bounds. Thus, we want to make this operation a bit more generic and define lower bounds LB on subsets $M \in \mathcal{P}(S \times R)$. This means $LB(M) \leq \min_{(sys,res)\in M}\{WCET_{sys,res}\}$ must always hold. If we have such a lower bound, we can apply the rule:

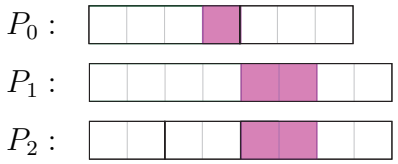
$$LB(M) \geq WCET_b \Rightarrow \text{Prune}M$$

We can apply this rule on every level of the algorithm, starting from the task set right up to the resource schedule level. If we now transfer our examples to this pattern, we get the following.

Definition 3.3. *Naive WCET lower bound LB_n*
Let *part* be a partitioning.

$$LB_n(part) = \max_{p\in part}\{\sum_{\tau\in p}\tau.nWCET\}$$

$LB_n(part)$ is a lower bound on $\{WCET_{sys,res}|sys \in S(part), res \in R\}$, which means that we can prune all system schedules belonging to the given partitioning and of course also every resource schedule which could be derived from those system schedules. $S(part)$ denotes the set of all derivable system schedules given the partitioning *part*.



(a) An exemplary schedule

	$comp_{start}$	$comp_{end}$	$\#accesses$
P_0 :	3	3	1
P_1 :	4	2	2
P_2 :	4	2	2
	↓min	↓min	↓sum
bound :	3+	2+	5 = 10

(b) The important values

Figure 3.8: Computation of the lower bound LB_{bp}

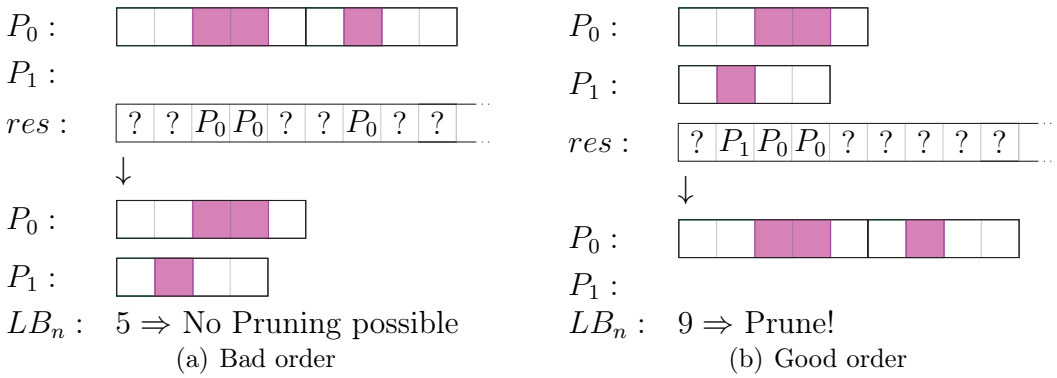
A big disadvantage of the previous bound is that it neglects all accesses. Thus, if we have many accesses in the tasks the sum of all accesses gives us a better bound, but this bound is even a bound on the whole task set. This means, if we reach this bound, we can abort the whole computation completely. Of course, this bound is probably very often far away from actual situations. But we can still improve it by adding the smallest computation times at the start and the end of each processing unit and contain a bound which respects the task accesses. This brings us to the lower bound LB_{bp}.

Definition 3.4. *Sum of accesses plus the boundary points* LB_{bp}
LB_{bp} takes the computation time at the beginning and at the end of each processing unit and takes the minimum of them which means the time until the first and after the last access. Then we add the sum of all accesses in our setting and obtain a lower bound.

This lower bound is illustrated again in figure 3.8. In subfigure 3.8(b) the important values are listed for the exemplary schedule in 3.8(a). There we see the computation time at the start ($comp_{start}$) and at the end ($comp_{end}$) and also the number of accesses for each processing unit. We add the minima of the computation time values and the cumulative number of accesses to obtain the value of the bound. What we explore is that we increased the lower bound of the naive WCET which is 8 in this example to 10, so we got a more realistic approximation to the real WCET. This is of course not always the case, but works well if we have lots of accesses beside the boundary points of each processing unit.

We can also do this computation one abstraction level earlier if we apply it to the partitions. But then we do not know which task is the first and last on the specific processing unit, so we have to consider the start and end computation times of all tasks and take the minimum there. But this will often introduce much pessimism and thus often be far away from the actual WCET if we consider a concrete schedule. We can use this lower bound of course anyway if we can predict that every system schedule of a partitioning will not improve the currently best WCET. But we easily see that the bigger the set is which we can prune the less often we can apply this pruning operation.

Another bound is - as already mentioned - the height of a partially finished resource schedule in addition to the naive rest execution time as already seen in figure 3.7.



(a) Bad order (b) Good order

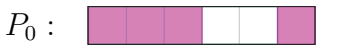
Figure 3.9: Sorting schedules

Definition 3.5. *Temporary WCET lower bound* LB_{tr}
LB_{tr} = Temporary WCET of res_{pre} as LB on $\{(sys, res) | res_{pre}$ prefix of $res\}$

We can compute the WCET of any schedule. If the schedule is still incomplete, we do of course not have the actual WCET, but a lower bound for it. So we can calculate this lower bound during the build of the resource schedule after every conflict and get increasing lower bounds which might reach our currently best value so that we can abort the computation of this part of the conflict tree.

3.3.3 Sorting schedules

As the introduced LB operators allow us to prune a lot of the search space, we want to shape our iteration order in a way which lets us apply this pruning as often as possible. Figure 3.9 shows a small example where the order of the schedules is important for the runtime of the algorithm. In general, we could sort the order of all pairs, but this would be too time- and space-consuming. Therefore, we will just sort the partitionings and thereafter will just incrementally build the concrete schedules. To achieve this we have to define a comparison operator for partitionings to obtain a traversing order as we cannot predict the sequence of considered partitionings with the set structure which we have at the moment. So in the current situation we want to define metrics on partitionings. These metrics have to fulfil two conditions. On the one hand we obviously want the outcoming order allow us to prune a lot of the search space, but on the other hand the metric has to be computed efficiently because otherwise the overhead would exceed the bonus won through the smaller search. So, we have a trade-off between efficiency and effectiveness. The extreme case for example would be the usage of the actual minimal WCET as metric on the partitionings as this will probably give us a very good iteration order but to compute this metric, we have to solve the initial problem itself. So we see that both conditions have to balance each other. The first idea of an efficient metric could therefore be the naive WCET LB_n as described in definition 3.3. This can be computed fast and it gives us a hint on how good a concrete schedule can be. But the obvious disadvantage at this metric is that it neglects all bus accesses.



$UB_{mb}(P_0) = 3 + 6 = 9$

$UB_{mb}(P_1) = 4 + 7 = 11$

$\Rightarrow UB_{mb} = 11$

Figure 3.10: Example: maximum blocking metric

Another possibility is the maximum blocking metric $WCET_{mb}$. Intuitively, the $WCET_{mb}$ adds all accesses on other processing units to the naive WCET of the partition under consideration. We obtain an upper bound on the best possible schedule according to this partitioning with this metric because no processing unit can be blocked longer than there are accesses on other processing units available as we regard only reasonable schedules.

Definition 3.6. UB_{mb}
Let $part = \{p_0, \ldots, p_m\}$ be a partitioning and let $nA(P_i)$ be the total naive access length occuring on partition P_i.

$$UB_{mb}(P_i) = WCET_{naive}(P_i) + \underbrace{\sum_{P \in part \setminus \{P_i\}} nA(P)}_{\text{other accesses}}$$

And finally

$$UB_{mb}(part) = \max_{P_i \in part} \{UB_{mb}(P_i)\}$$

To get a better intuition of the maximum blocking metric, regard figure 3.10.

Of course we could have also implemented this approach to the field of system and resource schedules, but we made the experience that the overhead of precomputing and sorting all of the possible schedules is almost every time much higher than the gain of execution time through pruning some search space. One particular annoying fact is that the exhaustive algorithm would prune many of the system schedules which a sorting approach would have to presort anyway. So a possibility for further work would be to integrate pruning into the precomputation phase. One would have to compute an upper bound for system schedules which are already computed and compare them to lower bounds of partitionings for which we have not calculated the according system schedules yet. So we could also prune those while precomputing objects for the sorting phase.

3.4 Evaluation

3.4.1 The test suites

We have build a parameterized random test case generator which gets the following input:

1. Minimum and maximum number of processing units and tasks

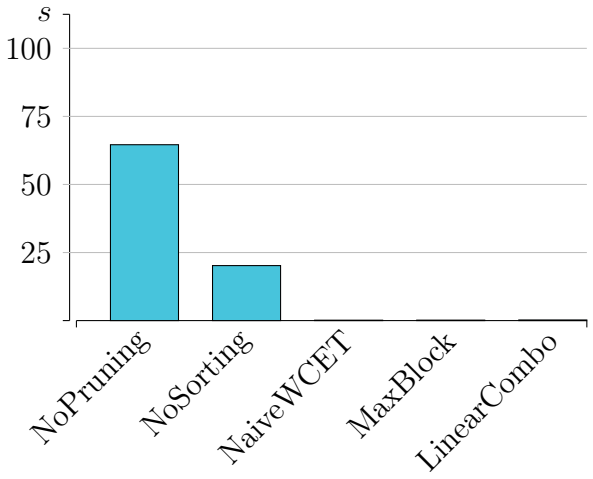
2. Minimum and maximum amount of a task's naive WCET

3. Probability if a task τ has a bus access at time t ($\forall t = 0, \ldots, \tau.nWCET$)

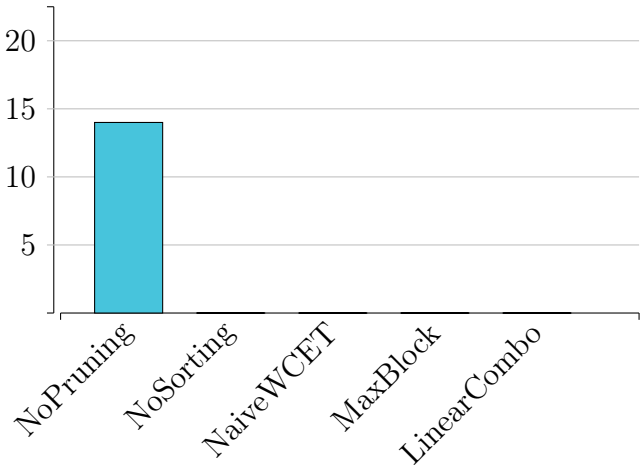
We generated three test suites of each 100 test cases. The numbers of processing units were uniformly distributed in the interval $[2, 4]$ whereas the task length is quadratically distributed on the interval $[3, 30]$ to make smaller naive WCETs more probable. The access probability for the tree test suites were $10\%, 25\%$ or 50% respectively. Therefore, we call the test suites T_{10}, T_{25} and T_{50}. We decided for those parameters to keep the test cases small enough so that we can always evaluate the exhaustive algorithm. We assume that these heuristics which perform better than others on these smaller test suite will also perform better on larger settings. But even though they are quite small, we had to delete some cases out of T_{50} because the exhaustive optimization could not handle them in less than 15 minutes. Those cases mostly had 4 processing units and many long tasks so that the computation of the resource schedule enlarged the runtime too much.

To compare the different techniques we presented about how to to speed the exhaustive algorithm we considered two performance indicators. At first of course the execution time(in milliseconds) to compute the result, but secondly also the number of schedules which were really computed to see how much of the search space the technique could prune. We already mentionned that in our implementation the calculation of a schedule's $WCET$ is neglectable because this is already done during the construction of the schedules. Therefore we additionally want to have a more implementation-independent value than the execution time and thus we decided for the number of computed schedules. We compared once the exhaustive algorithm without any pruning to the one which applies the LB operators and to the approaches which had a sorting step before the actual computation phase. Figure 3.11 lists all presented approaches for the exhaustive optimization, evaluated on T_{10} and we see that the results of the algorithm which does not prune anything are so bad compared to the other ones that the bars of the pruning ones' values appear to be equal to the zero line. Therefore, figures 3.12, 3.13 and 3.14 just show the pruning algorithms because it only makes sense to compare those ones to each other. Each figure shows the evaluation on one of the three test suites. On the first bar, we see the iterator which does not have a pre-sorting phase, but considers the partitionings in our default order. All the other bars show the iterators with a pre-computing step before the actual computation. The name of the bar gives a hint which metric was used for the sorting. LinearCombo means the combination of both metrics with each a weight of 50%. Additionally, we simulated an approach which delivers the best possible order. We achieved this by running the evaluation twice and put the result of the first run at the first place of the second run's search space. This means we found the optimum as early as possible and we want to find out now how much this optimal order is able to prune in the different test suites. A line in the diagram indicates the number of schedules which had to be computed in this approach.

We can see that the lower the bus load in a test suite the better the sorting iterators. When evaluating the test suite T_{10}, the pre-sorting has a high impact to the execution time of the algorithm. There we find good results earlier and thus have to compute much fewer schedules which results in a significantly lower

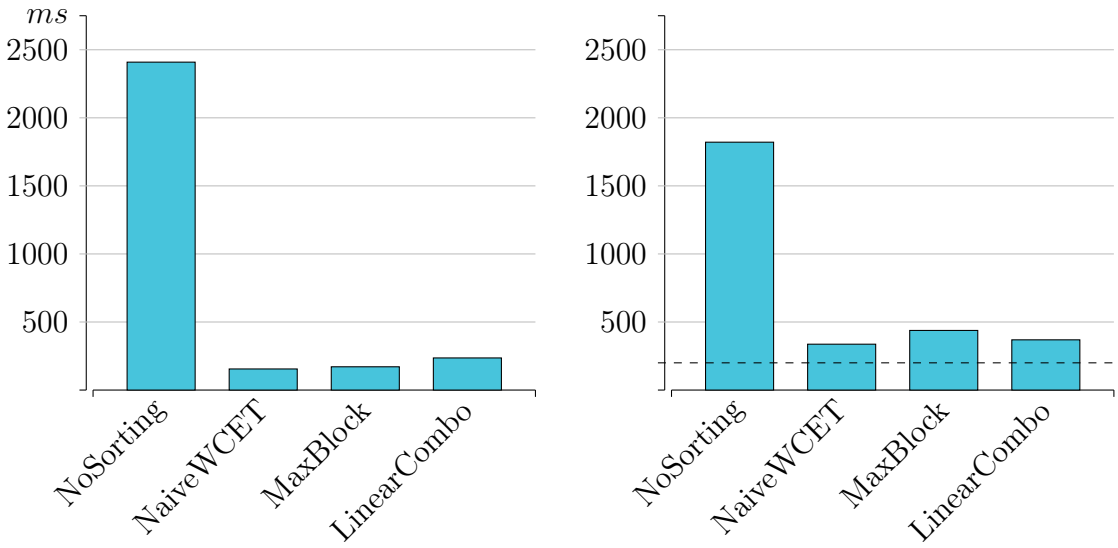


(a) Execution time in s



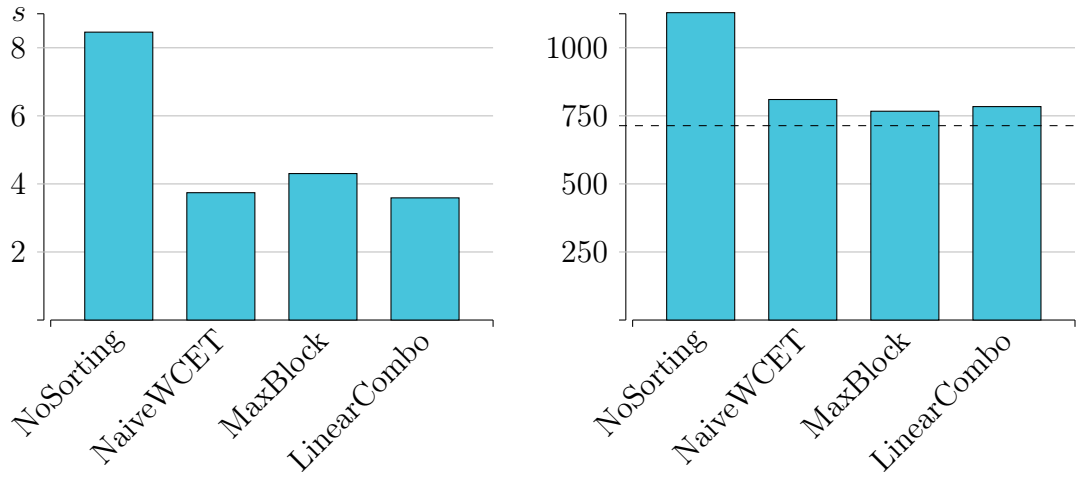
(b) Computed schedules times 10^6

Figure 3.11: Evaluation of the exhaustive optimization on T_{10}



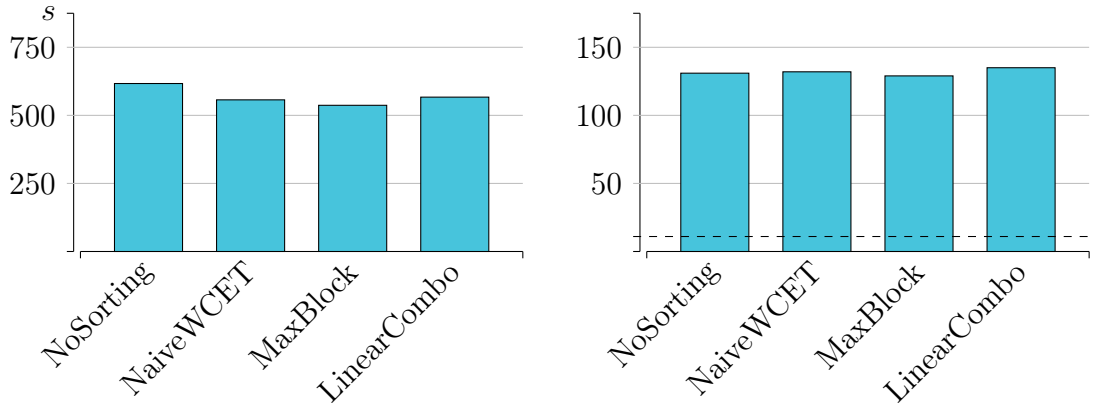
(a) Execution time in ms (b) Computed schedules

Figure 3.12: Evaluation of the pruning iterators on T_{10}



(a) Execution time in s (b) Computed schedules times 10^3

Figure 3.13: Evaluation of the pruning iterators on T_{25}



(a) Execution time in s

(b) Computed schedules times 10^6

Figure 3.14: Evaluation of the pruning iterators on T_{50}

execution time. In every test suite another metric turned out to be the best one for this setting. But the differences were too small to talk about significance.

We see that our pruning and sorting techniques really work well on the test suites with a low bus load. We can approximate the best possible order quite exactly.

But a question which arises when regarding the results is why the sorting iterators become worse when the bus load becomes higher. One answer is that the sorting metrics rely on naive assumptions which means they do not take the access distribution into consideration as this would require to know the system schedule, but we do only sort on the partitioning level because of the known reasons. Another possible answer is that our lower bound operators on partitionings and system schedules also work on quite naive assumptions. Thus, they become worse if the bus load becomes higher as those lower bounds are further away from the currently best $WCET$. Therefore, pruning is often only possible on the resource schedule level when applying the operator LB_{tr}. But as we do not sort the resource schedules we do not profit from sorting at this iteration step. Nevertheless pruning surely still makes sense. We tried to evaluate the NoPruning-approach to T_{50}, but after an hour, not even half the test suite was progressed. So we see, we do still profit from pruning, but not from sorting any more because it does not lead to significantly more pruning than the default order.

It is also not an option to sort the resource schedules in our implementation as we would have to precompute them all. The only thing we could prune then, would be the calculation of the $WCET$ of a given complete schedule. But as already mentioned earlier, this calculation is already done when constructing the resource schedule, so we do not really prune anything.

An open challenge here is to construct lower bounds as well as sorting metrics which also work well for cases where we have a high bus load. We tried to do so, but one problem was that we have to consider the task order to get more realistic bounds or at least realistic sorting metrics.

Chapter 4

Heuristics

We saw many improvements of the exhaustive optimization approach, but as shown in figure 4.1 this approach still does not scale to larger problems because the runtime still grows exponentially. Thus the need of effective and efficient heuristics is obvious because the search space of the exhaustive algorithm is far too large. Thus we will now leave the field of optimality and talk about the runtime and effectiveness of several heuristics in this chapter.

4.1 Consider fewer schedules

Consider again the evaluation of the sorting approaches on T_{25} in figure 4.2. We added an additional value describing the number of schedules which were computed until the optimum was found. This brings to our first idea for a simple heuristic. We can just take an arbitrary approach which we worked out so far and stop the calculation after a given number of steps. This can be applied on every iteration level of the algorithm, means on partitioning, system schedule and/or resource schedule level. So we can define for every level a maximum number of steps which the algorithm should look at. But if we do this, the order in which schedules are constructed depends on the implementation default order which is not benificial as we could see in the evaluation of the sorting iterators. Thus, it could be senseful to first sort the objects and then look at some of the best schedules. Best here means of course just best with respect to a given metric. This has the disadvantage that we have to precompute all possible objects and their respective metric values at the level we want to sort which affects the runtime of this heuristic significantly. And

# **Tasks**	**Bus load**	**Runtime(s)**
10	10%	> 10
10	20%	> 15
10	50%	> 60
10	80%	> 600
20	50%	> 300

Figure 4.1: Runtime of bigger examples with 4 processing units

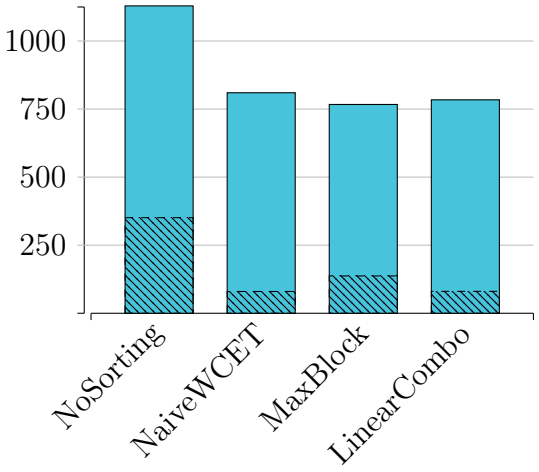


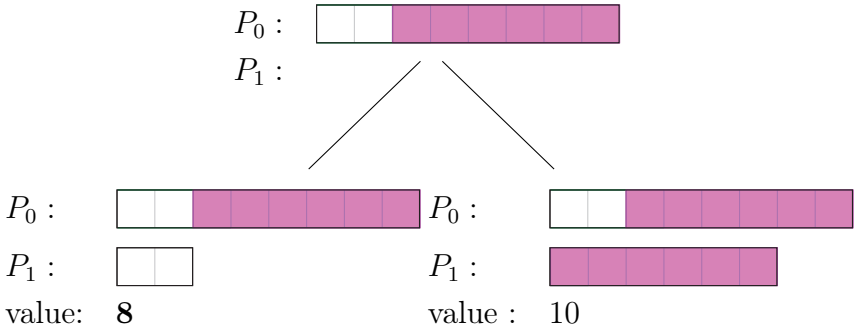
Figure 4.2: Computed schedules (times 10^3) until the optimum was found

another overhead is the sorting phase, too. We implemented the sorting approach on the partitioning level and always stopped after the first partitioing. As we only consider the best partitioning with respect to a given metric, we could also just try to compute or at least approximate this partitioning on the fly instead of computing all and searching the best. If we want to minimize our simple metrics of a partitioning we could also use an ILP solver [NW88] to solve this problem. The authors of [NR98] implemented similar problems with an ILP solver.

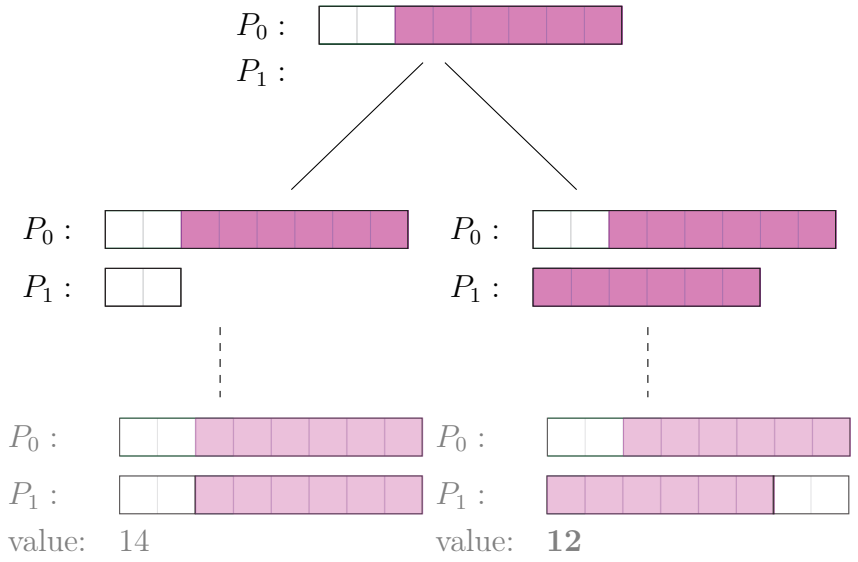
4.2 On-the-fly construction

The considered heuristic is bad for two reasons. If we sort the partitionings or system schedules this approach does not scale to larger problems because the precomputation phase is just too time consuming. On the other hand, if we leave sorting and precomputing out and stop the algorithm after m steps, the iteration order is bad. Overall, we either have a bad performance when we do sort, or we have a bad quality when we do not sort. But as we want a heuristic which combines quality and performance, we have to leave our inflexible pattern and compute schedules in another way. So there is a need to construct a heuristic which computes a result on the fly.

The first intuition is that we construct a metric which decides at every iteration step which task we have to place to the first free cycle. We also implemented the possibility to place several tasks at once and decide afterwards which combination was the best. So to say, we can look ahead in the decision tree of the task placement. This approach is illustrated in figure 4.3. There we have a 2-processor setting with 3 tasks to place. We see the decision after the first task is already placed with a lookahead of 1 and 2. The calculation of the metric values will be explained in section 4.3



(a) Lookahead 1



(b) Lookahead 2

Figure 4.3: Incremental task placement

Related work There is already an approach [RAEP07] which computes a whole schedule on the fly. In this approach, tasks are also placed step by step to the first free processing unit. The main difference of our system model and theirs is that they have a dependency graph for the tasks. They always place the task with the longest transitive dependency chain. So their approach does not completely fit into our model, but we adapted it as well as we could because it is one of the few approaches which can be compared to ours. As we do not have tasks which depend on each other, we adapt their heuristic via placing the task with the highest naive $WCET$. In this approach we can easily see that the distribution of the accesses is neglected when deciding for a task placement, so we see a possible point of improvement here.

After the placement is finished, we optimize the resource schedule in the way described in chapter 3. The derived algorithm can be seen in 3. The method

FirstTerminatingProc(sys, res) returns the processing unit with the smallest $WCRT$ in the given setting.

```
   Data: Tasks: set of all tasks, n: number of processors, met_sys: metric for
         system schedule
 1 sys ←"empty schedule";
 2 while Tasks not empty do
 3 |   proc_free ← FirstTerminatingProc(sys, res);
 4 |   (sys_b, τ_b, met_b) ← (null,null,∞);
 5 |   foreach τ ∈ Tasks do
 6 |   |   sys_tmp ← schedule τ to sys at proc_free;
 7 |   |   met_tmp ← met_sys (sys_tmp);
 8 |   |   if met_tmp < met_b then
 9 |   |   |   (sys_b, τ_b, met_b) ← (sys_tmp, τ, met_tmp);
10 |   |   end
11 |   end
12 |   sys ← sys_b;
13 |   Tasks ← Tasks \ {τ_b};
14 end
15 res ← such that WCET_sys,res is minimal;
16 return (sys, res);
```

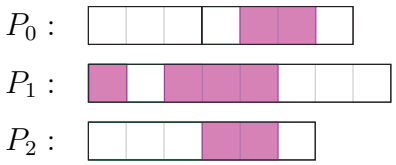
Algorithm 3: On the fly construction of a system schedule

4.3 Metrics

The effectiveness of this heuristic strongly depends on the given metrics with which we build the system schedule. That is why we will discuss this in the following pages.

4.3.1 System schedule metrics

$Metric_{conflicts}$ Our goal is to construct metrics which take into consideration the access distribution because we can use all the information the system schedule delivers. Our first idea was to count the number of naive conflicts - i.e. the number of conflicts in the first naive setting - and add this number to the naive WCRT. Regard an example for the calculation in figure 4.4. We cannot only regard the number of naive conflicts because then the sequential schedule would always get the best metric value and this is not what we want. It is important to mention that a conflict where m processors are involved counts as $m - 1$ conflicts because we need at least this number of steps in the resource schedule to solve the conflict. After evaluating this metric on several test cases we experienced that we did not only construct a metric, but also a safe and relatively tight upper bound on the best possible WCET. So we derive the following theorem proven for a two processor setting.



(a) A system schedule

Conflicts:	3
Naive WCET:	8
Sum:	11

(b) Metric calculation

Figure 4.4: Calculation of $Metric_{conflicts}$

Theorem 4.1. *Consider a scenario with two processors. Let sys be a system schedule with a naive WCET of* w^{naive} *and c conflicts in the naive scenario. Then there is a resource schedule res with* $WCET(sys, res) \leq w^{naive} + c$.

Proof by induction.
Let $c = 0$
Then it is obviously clear that there is a res with

$$WCET_{sys,res} = w^{naive} \leq w^{naive} + 0 = w^{naive} + c$$

Suppose the theorem is true for a $c \geq 0 (I.H.)$, prove for $c + 1$:
So sys has $c + 1$ conflicts in the first naive scenario (no blocking introduced at all) $\Rightarrow \exists t \in \mathbb{N}$ where the first conflict is at time t. Solve the conflict by granting the bus to the first processing unit. Through blocking we get a new situation where we can consider tow cases.
Case 1: There is no further conflict
$\Rightarrow$ There is obviously a reasonable resource schedule res with

$$WCET(sys, res) = w^{naive} + 1 \leq w^{naive} + (c + 1)$$

Case 2: There is at least one further conflict
Again $\exists t' \in \mathbb{N}, t' > t$ so that the first occuring conflict in this new scenario is at time t'. Now solve the conflict by granting the bus to the second processing unit. Now the first processing unit is blocked, thus the offset of the two processing units at time $t' + 1$ is the same as at the beginning at time t'. That is why there are at most c conflicts in the remaining scenario. The remaining naive $WCET$ is

$$w_r^{naive} = w^{naive} - t' \tag{4.1}$$

So we can estimate the actual $WCET$ of the rest with

$$w_r \overset{I.H.}{\leq} w_r^{naive} + c \overset{(4.1)}{=} w^{naive} - t' + c \tag{4.2}$$

The preliminary schedule has a $WCET$ of

$$w_P = t' + 1 \tag{4.3}$$

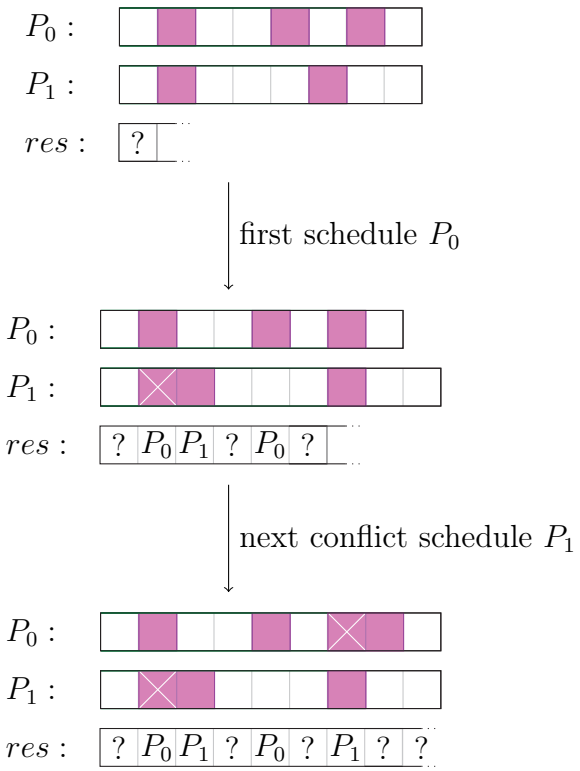


Figure 4.5: Alternating resource schedule

So considering the whole schedule we get a $WCET$ of

$$\begin{aligned} w =& w_p + w_r \\ \overset{(4.2)}{\leq}& w_p + w^{naive} - t' + c \\ \overset{(4.3)}{=}& t' + 1 + w^{naive} - t' + c \\ =& w^{naive} + (c+1) \end{aligned}$$

□

The resource schedule constructed in theorem 4.1 is sketched in figure 4.5.

In all generated test cases for more than two processing units the described bound was safe, but we did not find a nice way to prove it at a similar level of formalism as for two processing units. The difference is that the resource schedule cannot be constructed alternating like described above, but there is also an easy constructive criterion for the resource schedule to let the bound hold. We just always have to schedule the processing unit which has been scheduled the least times.

Even if we could not convince every reader that we obtain a safe bound through this metric and even if there is an artificial example which is not covered by our test suite where the metric is not a bound, we can though use it as metric and this was our actual goal. In further contexts we will call this metric $metric_{conflicts}$. We can even improve the metric and also tighten the bound by presolving a maximum of m

Note 4.1. Alternating resource schedule heuristic res_{alt}
For a given system schedule sys with a naive $WCET$ of n and a total number of naive conflicts of c, we can construct the resource schedule res due to the following rule:
Solve every conflict by granting the bus to the processing unit which was preferred the least times when previous conflicts were solved. So we do not count those points in time where we have a straight-forward decision (no conflict). We assume and our evaluation confirmed that we can always predict:

$$WCET_{sys,res} \leq n + c$$

The heuristic's runtime is in $\mathcal{O}(WCRT_b^{UB}(sys))$.

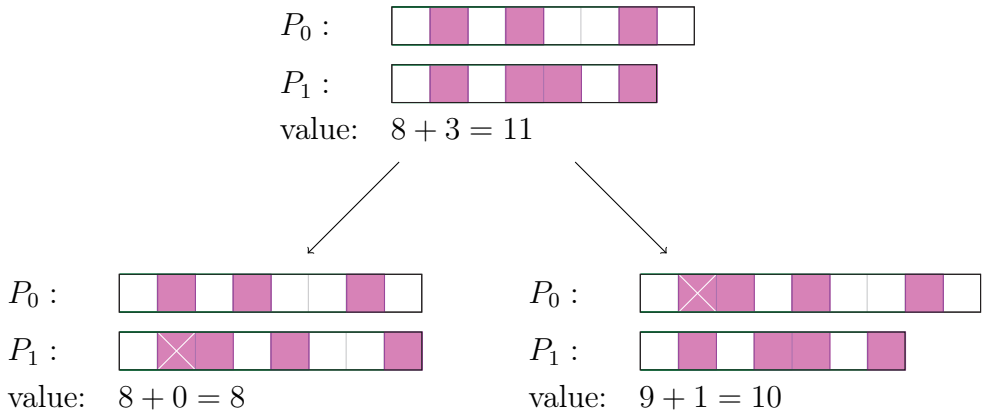


Figure 4.6: $metric_{conflicts}$ with a presolving size of 1

conflicts, calculate the $metric_{conflicts}$ after each solving step and take the minimum as actual value. An example why this could be helpful is illustrated in figure 4.6. In the initial setting we have a value of 13 and after one conflict is solved we get a value of 9, so this might help getting better results when deciding the placement of a task.

$Metric_{penalty}$ The last metric punished a long WCRT and a high number of conflicts. But when incrementally constructing a schedule we probably do not want that the first tasks we place to have a short naive WCET and a low number of accesses such that in the end we have only long tasks with lots of accesses left. This means many conflicts and/or a high difference between the completion time of different processing units as we cannot "fill up" critical cycles with small tasks with a low bus load. So we also want to punish situations where the bus is unused for longer times to avoid problems when only a few tasks are left. The idea is that the perfect situation would be that we have one access each cycle which means no conflicts and a uniform distribution of the accesses. So we might want exactly such schedules.

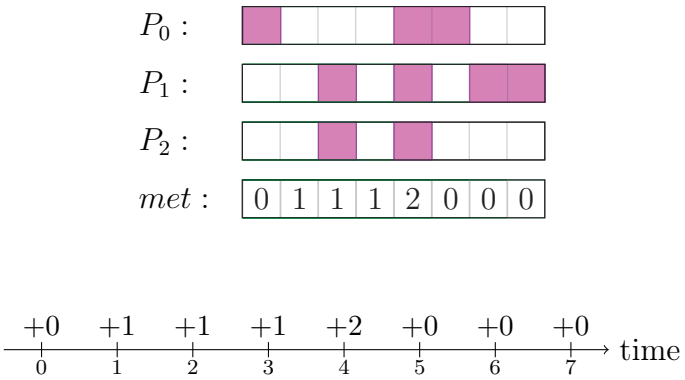


Figure 4.7: Evaluation of $metric_{penalty}$

The idea of the next metric which we call $metric_{penalty}$ is that we step through every cycle t and regard the situation on all processing units. Let met be the current value of our metric (initially 0). We can find three different cases:

1. There is no access at time t. This means wasted time, so we increase met by 1

2. There is exactly one access at time t. This is perfect, so we do not add a penalty.

3. There are n accesses at time t with $n > 1$ which means we have $n - 1$ more accesses than can be served, so we increase met by $n - 1$

We evaluate the metric on an example, shown in figure 4.7.

4.3.2 Resource schedule metric

The approach of Algorithm 3 is bad for at least one reason. When doing the task placement, we look at a naive situation because the offset between task to place and the ones already placed on other processing units does not need to correspond to the schedule with the actual best resource schedule. So our intuitive advantage of taking the access distribution into consideration might decrease due to the naive look at the setting. So we have to integrate the resource schedule optimization into the task placement which leads us to Algorithm 4. After we decided for a task placement, we build the resource Schedule further until the place where there are tasks on all processing units available. We also see that we can use the resource schedule for the evaluation of the placement metric. The exact utilization of the different metrics is explained in subsection 4.3.3. So far, we always optimized the resource schedule with respect to the $WCET$. But now we have to refine our definition of optimality when regarding a schedule where further tasks still have to be placed. We want to construct a resource schedule such that the schedule stays flexible for further task placements. If we always consider a schedule where not all tasks are placed yet the same way as a complete schedule and thus optimize the resource schedule with respect to the total WCRT we will often not get optimal results. This fact is

illustrated in figure 4.8(a). At this example two out of three tasks are already placed and the resource schedule is build until one processing unit is finished. Thereafter the last task is placed and one can see that our computed result is far from the optimal one which we see in figure 4.8(b).

```
   Data: Tasks: set of all tasks, n: number of processing units, met_sys, met_res:
         metrics for system and resource schedule
 1 (sys, res) ←"empty schedule";
 2 while Tasks not empty do
 3 |   proc_free ← FirstTerminatingProc(sys, res);
 4 |   (sys_b, τ_b, met_b) ← (null,null,null,∞);
 5 |   foreach tau ∈ Tasks do
 6 |   |   sys_tmp ← schedule τ to sys at proc_free;
 7 |   |   met_tmp ← met_sys (sys_tmp, res);
 8 |   |   if met_tmp < met_b then
 9 |   |   |   (sys_b, τ_b, met_b) ← (sys_tmp, τ, met_tmp);
10 |   |   end
11 |   end
12 |   sys ← sys_b;
13 |   res ← complete res such that met_res is best;
14 |   Tasks ← Tasks \ {τ_b};
15 end
16 return (sys, res);
```

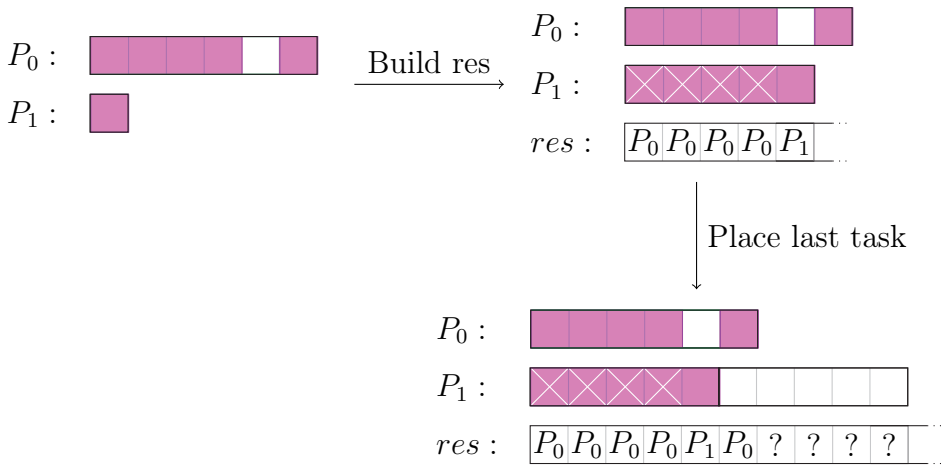
Algorithm 4: On the fly construction of a schedule with integrated resource schedule

Due to the fact that we do not want to block processing units more than really needed we define the whole blocking time as $metric_{blocking}$ for a resource schedule. As this should only be a metric for incomplete schedules we have to make a difference between the first $n-1$ task placements and the last one if we have n tasks. If we do it like this we can apply the $metric_{blocking}$ for the first $n-1$ placements and the WCRT as metric for the last placement to get the optimal result at least for the last part of the schedule.

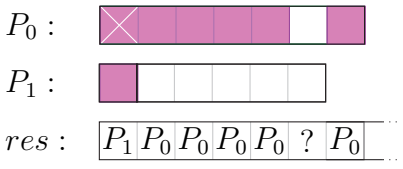
The disadvantage of this approach is that we still have to build the conflict tree for one resource schedule as we did it in the exhaustive algorithm which means that we have an exponential worst case runtime for this resource schedule. We have a heuristic which builds the resource schedule efficiently as described in note 4.1, but in a further work one could optimize this step of the approach. Our main focus still relies on the choice of the system schedule.

4.3.3 Utilization of the metrics

We want to compare our metric values in a fair way. If we iteratively place tasks, we want to estimate how good this placement was. The metrics in the two approaches are used in a different way. We will explain how the evaluation of the metrics work for each approach.



(a) Bad resource schedule



(b) Optimal result

Figure 4.8: Incremental build of system and resource schedule

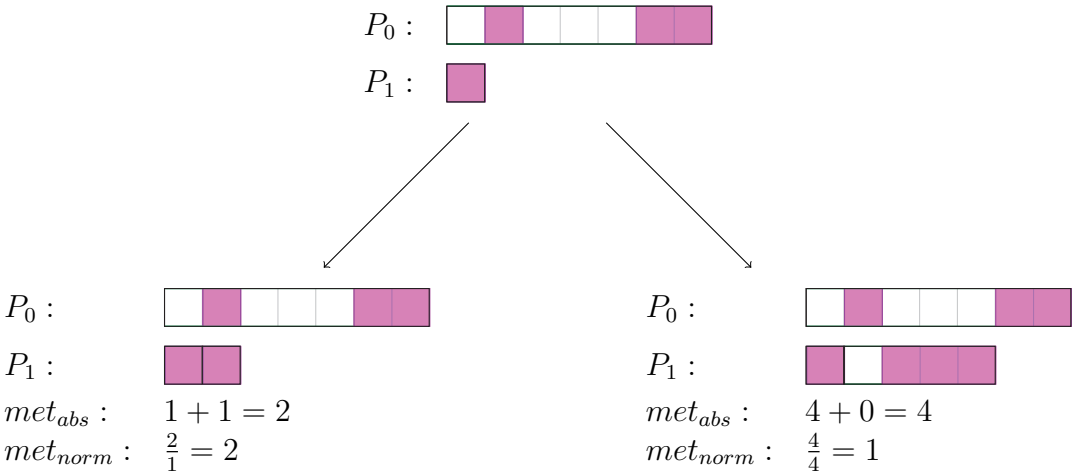
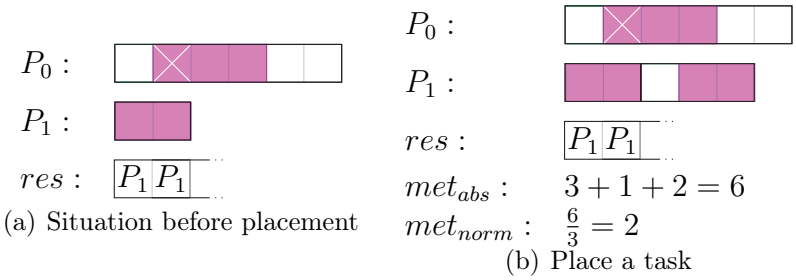


Figure 4.9: Utilization of $metric_{conflicts}$ in the non-integrated approach



(a) Situation before placement

(b) Place a task

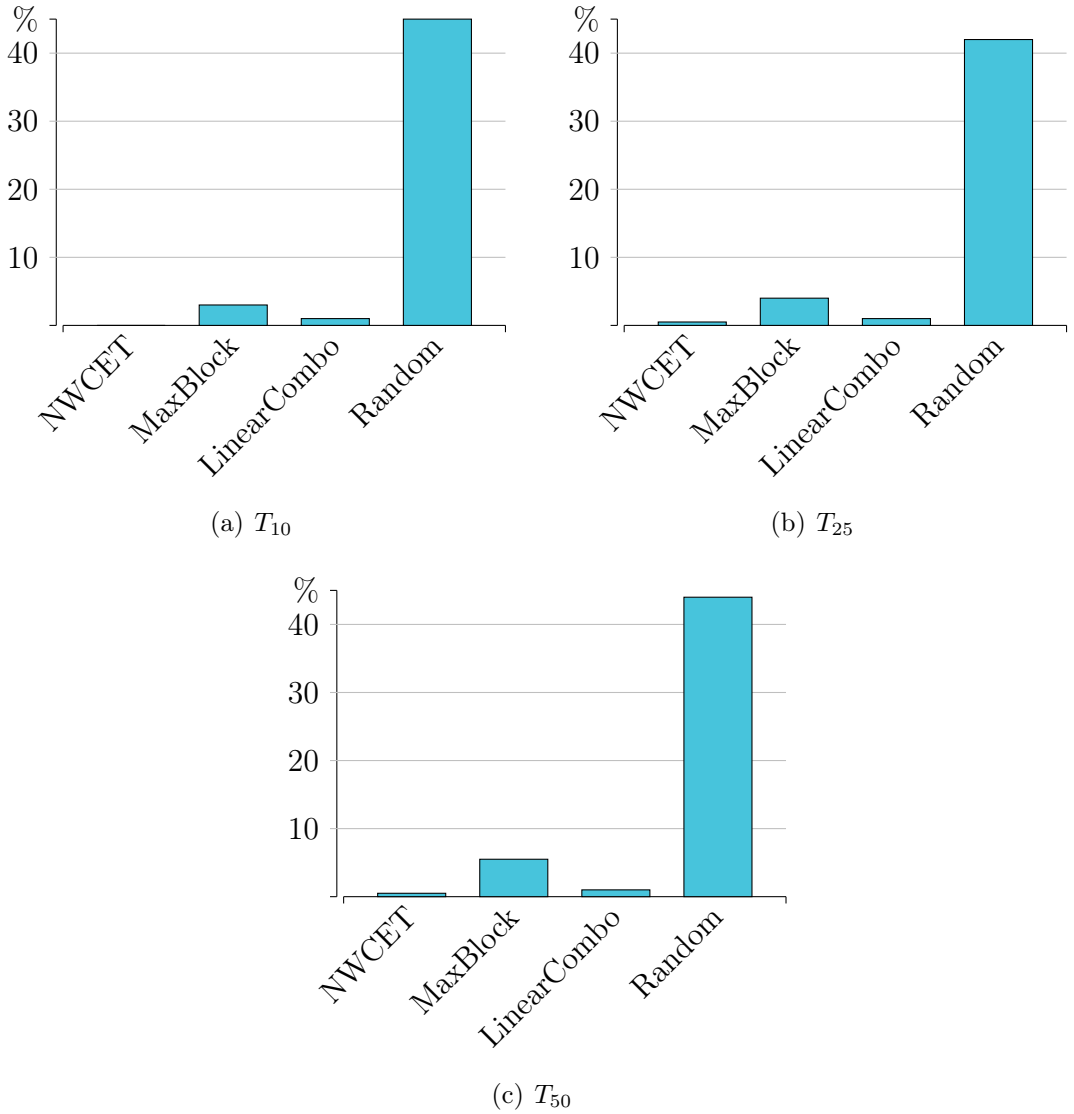
Figure 4.10: Utilization of $metric_{conflicts}$ in the integrated approach

Non-integrated resource schedule

In general, we evaluate all metrics only at the interval of the won *busy time*. The *busy time* describes the latest point in time where all processing units execute tasks. We do this as it does not make sense to talk about interferences at cycles where not all processing units execute tasks. On the other hand, we also do not need to evaluate the metrics at the parts of the schedule where all processing units have been busy before the placement as this part and so the metric value stays constant at this interval. Additionally, we normalize the absolute metric values met_{abs} against this won busy time and finally obtain the normalized metric value met_{norm}. Otherwise, small tasks with a relatively bad metric value would possibly be prefered over long tasks with a relatively good metric value. This fact is sketched in figure 4.9. We see that we can place two tasks. One task with a naive $WCET$ of 1 and 1 access and the other task with a naive $WCET$ of 4 and 3 accesses. If we would use the absolut metric values, we would get 2 for the first placement as we have a naive $WCET$ of 1 and 1 conflict until the first processing unit is not busy anymore. For the other placement we would have an absolute metric value of 4 (naive $WCET$ of 4 and 0 conflict). So we would decide for the smaller task. But if additionally normalize the values against the won busy time - 1 in the first case, 4 in the second one -, then we decide for the larger task which leads to a better overall solution.

Integrated resource schedule

The main difference of the approach which builds the resource schedule integrated with the task placement is that we can use more information when evaluation the metric for the system schedule. When we place a task at time t, we know about the resource schedule until time $t - 1$. So we have to evaluate the metric only from time t on and we have to consider the actual task offsets which are introduced by the resource schedule. After this, we can add the length of the resource schedule to the calculated metric value to obtain a more realistic value. This value is then again normalized as described for the other approach. Figure 4.10 sketches again the utilization. We have a naive rest $WCET$ of 3 and 1 conflict in the time time interval which we are interested in. This is the time from 2 (res is build) until 5 (all processing units busy). Finally, we add the length of res which is 2 as already mentionned and obtain a total absolute value of 6 which we normalize in a further



(a) T_{10} (b) T_{25}

(c) T_{50}

Figure 4.11: Percentual error of "Taking-One-Partitioning" approach

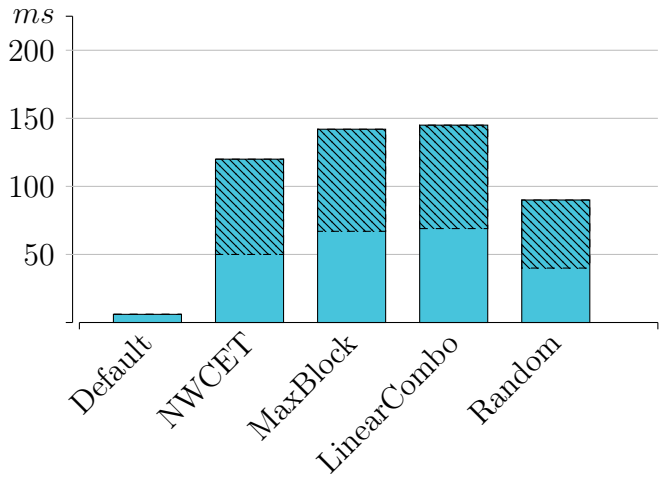
step as already described earlier.

4.4 Evaluation

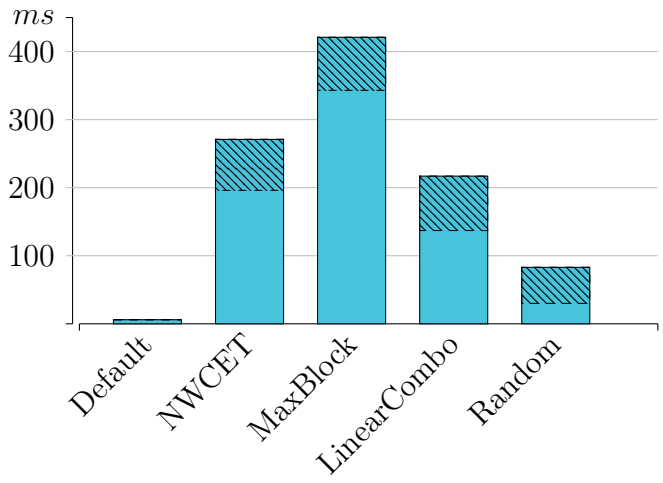
We have again the same test suites as for the evaluation of the exhaustive optimization to be able to compare the results. The definition of the test suites was explained in subsection 3.4.1.

4.4.1 Consider fewer schedules

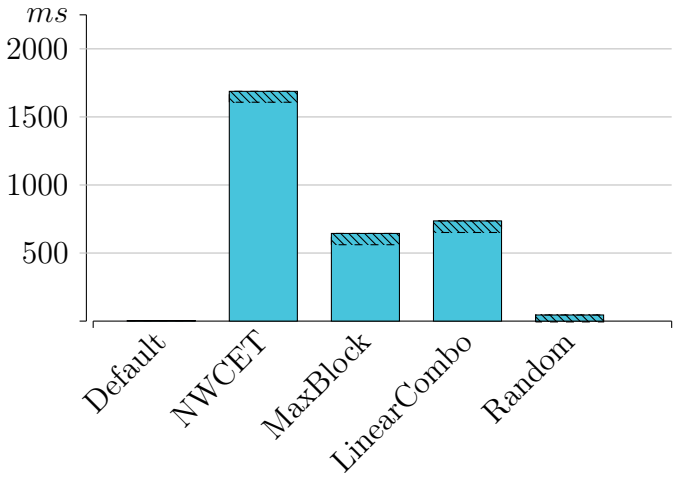
At first, we want to take a look at the approach which always just considers the first partitioning. We want to compare two values. At first, we want to know



(a) T_{10}



(b) T_{25}



(c) T_{50}

Figure 4.12: Execution time of "Taking-One-Partitioning" approach (in ms)

how effective our heuristic, so figure 4.11 shows the average relative error of the outcoming result compared the the optimal one. Figure 4.12 shows the execution times of the different approaches.

The different bars stand for the approaches on how to choose this one partitioning which we want to consider. The first three are captioned with the metric with which are used to sort the set of precomputed partitionings. LinearCombo means here a linear combination of both NWCET and MaxBlock with each a weight of 50%. The bar indicated with "Random" describes exactly what one might expect, it randomly chooses an arbitrary partitioning. We also evaluated the approach which takes the first partitioning of the default implementation order, but there the relative error was so high that we did not list it in the diagrams. This comes from the fact that the algorithm which iterates all possible partitionings for a given task set and a given number of processing units starts with the partitionings which has just one partition with every task in it as already described in chapter 3. So we always get the sequential WCET in this scenario. The dashed parts of the bars indicate the time which was consumed by the precomputation- and search-phase.

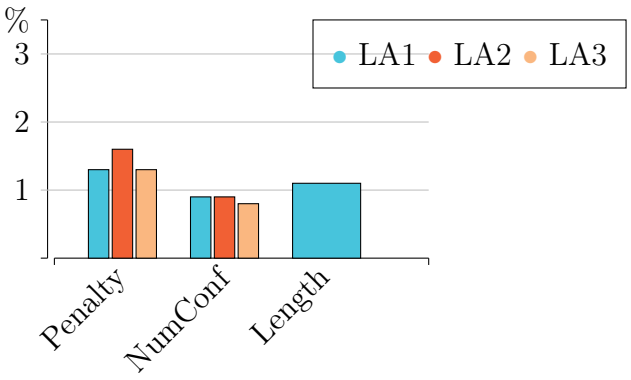
So, what we see is that the naive WCET is a really good metric to approximate the quality of a partitioning. But we also have to consider the fact that we still compute and consider every possible system schedule according to this one partitioning. And for every system schedule we build the whole conflict tree to get the optimal resource schedule. This means that we can of course get really effective heuristics here and the evaluation agrees with that, but we still traverse much of the search space which makes the heuristic not scalable which is underlined by figure 4.12. The runtime grows significantly when the access concentration becomes larger.

4.4.2 On-the-fly construction

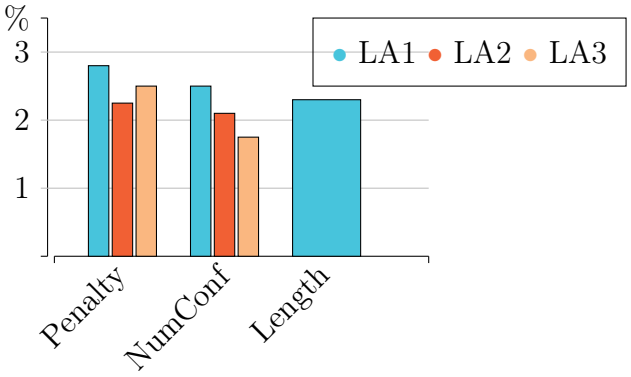
Non-integrated resource schedule

Now let us take a closer look at the technique which incrementally builds the system schedule and then finally at the end of the algorithm an optimal resource schedule for this given system schedule is computed. Figure 4.13 and 4.14 show the results of this evaluation. We considered three settings in our evaluation. The first two use the metrics described in section 4.3 in the chapter Heuristics. The bars are captioned with Penalty for the $metric_{penalty}$ and NumConf for the $metric_{conflicts}$. The third bar captioned with Length stands for an algorithm which is similar to the one described in [RAEP07]. As already mentionned earlier we assume that we can get better results than that as we take the access distribution of the tasks into consideration.

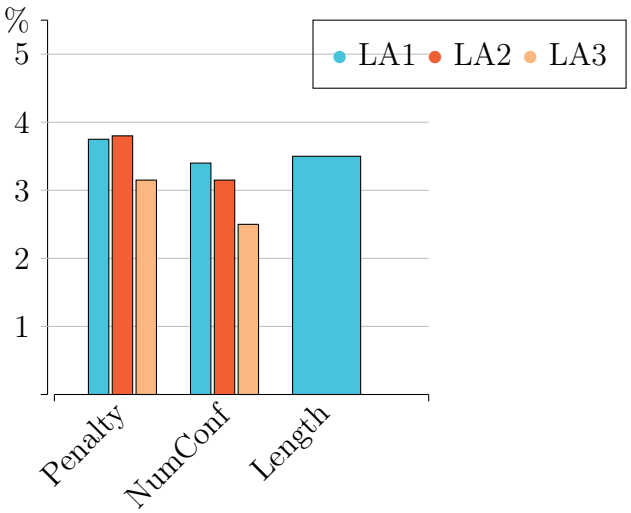
We see that increasing the lookahead leads to significantly better results, but on the other hand also to much higher execution times. The two metrics which we constructed behave quite similarly, the $metric_{conflicts}$ still perform a bit better. Overall, the comparison of the three metrics comes to the same result no matter which test suite we regard. Although the average error becomes higher when we increase the bus load, the win of precision is similar over all test suites. This is probably due to the fact that we build the resource schedule in the end. The



(a) T_{10}

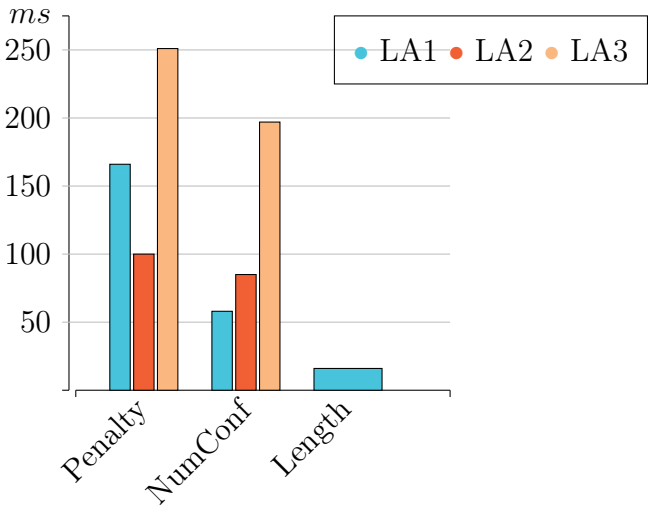


(b) T_{25}

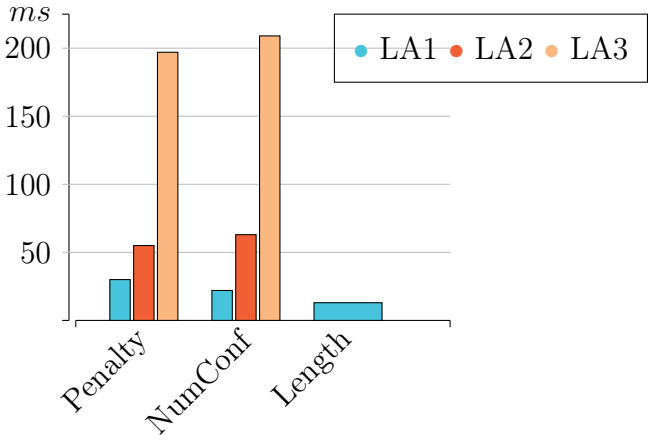


(c) T_{50}

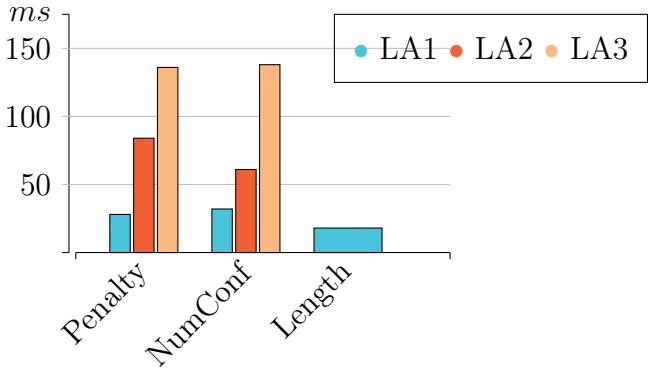
Figure 4.13: Relative error of task placement without integrated resource schedule



(a) T_{10}



(b) T_{25}



(c) T_{50}

Figure 4.14: Execution time of task placement without integrated resource schedule

advantage of our metrics is that they take the access distribution into consideration, but if we always consider naive scenarios, then this advantage possibly becomes smaller when we already placed many tasks. This problem is especially high when we have a high bus load.

Integrated resource schedule

We suppose that the disadvantage just described is eased when constructing the resource schedule in parallel. Figure 4.15 and 4.16 show the results of the integrated approach with the different metrics.

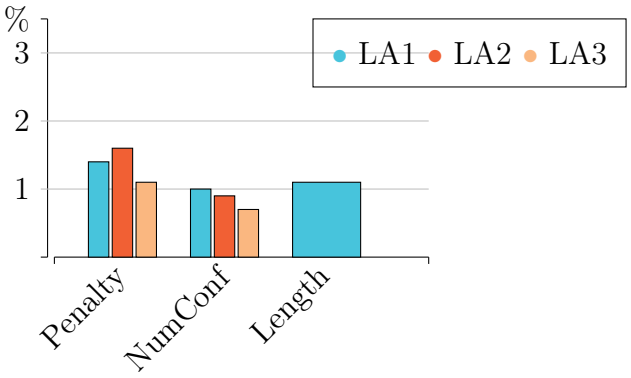
The results of the integrated approach comparing the three different placement metrics do quite coincide with the evaluation before. But we have a higher percentual error, especially in T_{50}. The reason for this is that we can lose a lot of precision when we try to optimize the resource schedule when not all tasks are placed yet. We do not really save time when we build the resource schedule in the integrated version because the conflict tree is still built completely after every placement. So the runtime of the resource schedule optimization is the same for each approach.

Thus, there are two open tasks for future work. On the one hand, one could simplify the resource schedule heuristic to obtain a better runtime which makes the approach more scalable. On the other hand, one could also try to find more effective resource schedule metrics so that the outcoming percentual error decreases. In both cases the quotient of quality and time would increase.

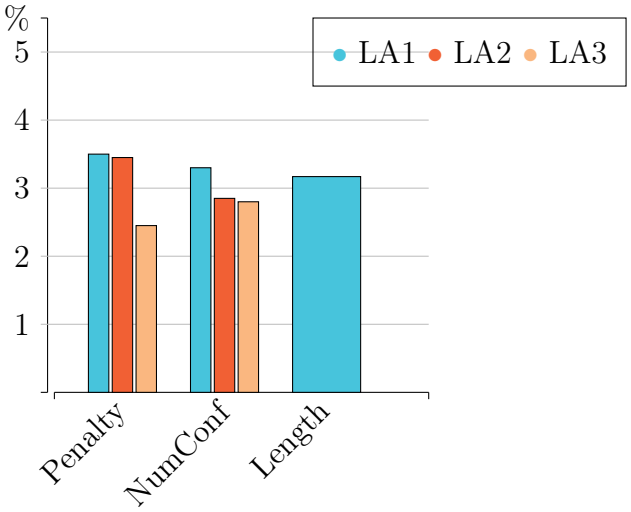
4.4.3 Alternating resource schedule heuristic

We already talked about an alternative resource schedule heuristic in note 4.1. As the results of the integrated on-the-fly heuristic were not as good as we expected when we use the $metric_{blocking}$ for building the resource schedule, we evaluated the same approach with this alternative resource schedule heuristic, too. The results are shown in figure 4.17(a). As the Length-Heuristic provided terrible results with this resource schedule construction (up to 85% average error), we do not list it again in this figure. Additionally, we only show the results for the LA2-approach. If we only take a lookahead of 1, the results are similarly bad as the one of the Length-heuristic. But the approach seems to significantly profit from increasing the lookahead. The results for the two different system schedule metrics were nearly identical, so we do not list them both again.

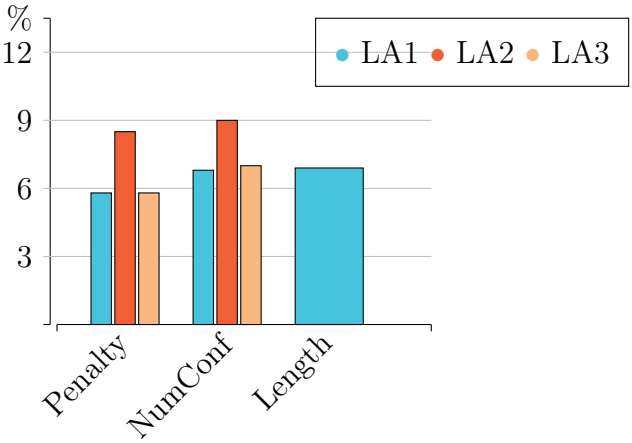
The execution times for the different test suites are listed in figure 4.17(b). We have a much lower runtime than with the previous approaches and the quality of the results are equal or even better in some cases. It is also worth mentioning that this heuristic approach is the only one which scales to larger problems. The task placement as well as the resource schedule construction have polynomial runtime.



(a) T_{10}

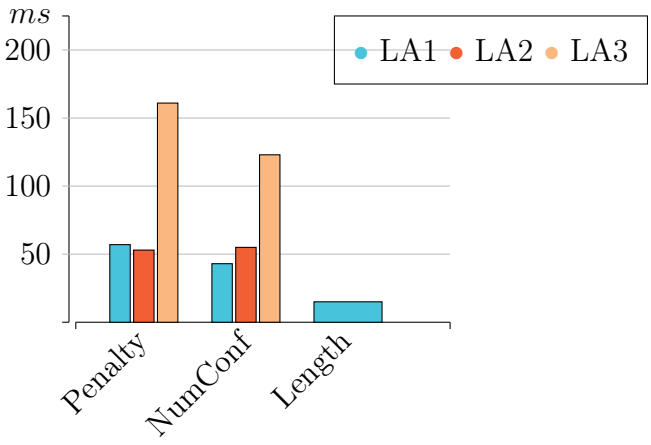


(b) T_{25}

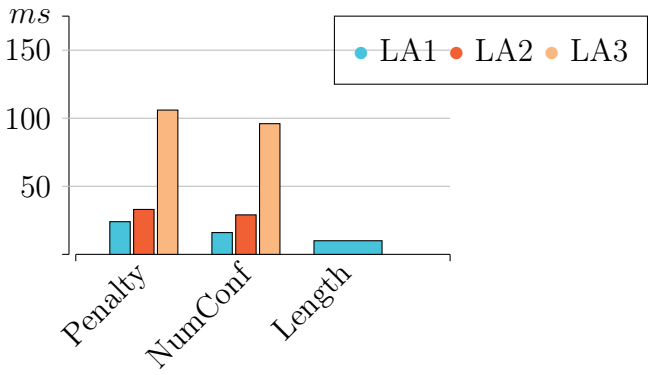


(c) T_{50}

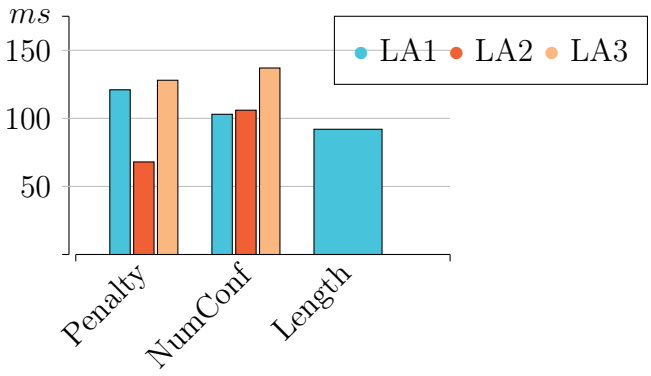
Figure 4.15: Relative error of task placement with integrated resource schedule



(a) T_{10}

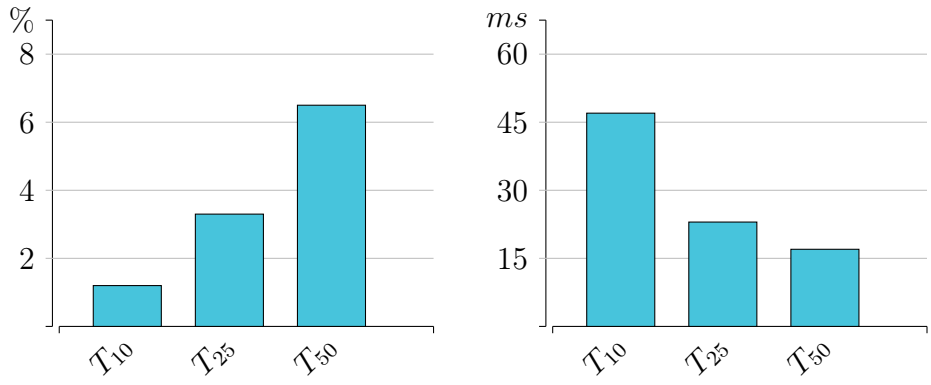


(b) T_{25}



(c) T_{50}

Figure 4.16: Execution time of task placement with integrated resource schedule



(a) Relative average error

(b) Execution time

Figure 4.17: Task placement with integrated alternating resource schedule

4.5 Conclusion

Now we have presented many different heuristics and one might ask the question which one is te best. The answer is as often: It depends! We have seen that different approaches reach a different quality within a particular amount of time. Thus, the decision which approach to take depends on the desired quality as well as the desired runtime. We assume that the quality of our heuristics is similar when evaluating them on larger examples. We can of course not provide evidence for this because searching for the actual optimal result would last too long. But if our assumption holds, we may define a quotient q of quality and runtime to obtain a comparable value of all approaches. More exactly, the measurement works like this.

$$q = \frac{\text{Quality}}{\text{Runtime}} = \frac{\frac{1}{\text{Error}}}{\text{Runtime}} \tag{4.4}$$

Error means the percentual average increase of the optimal $WCET$. If for example the $WCET$ was increased by 4%, the error value is $1,04$. Therefore, the optimal quality is 1. Figure 4.18 describes the values for quality and runtime of the different approaches. Integrated OTF 1 describes the integrated approach where the resource schedule was optimized with respect to $metric_{blocking}$. Integrated OTF 2 describes the same approach with the alternative resource schedule constructon. We assume that the runtime and like this the quotient becomes much worse for bigger examples with the exception of the last approach listed as this is the only approach which has polynomial runtime for the construction of both system and resource schedule.

Approach	T_{10}	T_{25}	T_{50}
Best exhaustive	1	1	1
Best take-one-part	1	$\frac{1}{1.01}$	$\frac{1}{1.015}$
Non-Integrated OTF	$\frac{1}{1.01}$	$\frac{1}{1.02}$	$\frac{1}{1.03}$
Integrated OTF 1	$\frac{1}{1.01}$	$\frac{1}{1.03}$	$\frac{1}{1.06}$
Integrated OTF 2	$\frac{1}{1.013}$	$\frac{1}{1.032}$	$\frac{1}{1.063}$

(a) Quality

Approach	T_{10}	T_{25}	T_{50}
Best exhaustive	200	4000	500000
Best take-one-part	100	300	1500
Non-Integrated OTF	100	50	150
Integrated OTF 1	50	40	100
Integrated OTF 2	45	20	17

(b) Runtime

Approach	T_{10}	T_{25}	T_{50}
Best exhaustive	0.005	0.00025	0.000002
Best take-one-part	0.01	0.0033	0.00066
Non-Integrated OTF	0.0099	0.02	0.0065
Integrated OTF 1	0.02	0.025	0.0094
Integrated OTF 2	0.022	0.048	0.055

(c) Quotient

Figure 4.18: Quality vs. runtime

Chapter 5

Future work

5.1 Concrete realization of a model

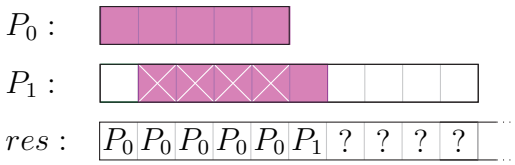
As the attentive reader already noticed, we did several assumptions to our model which influenced further design decisions of algorithms and heuristics.

A straight-forward concretization of our system model could be a system fulfilling the following properties.

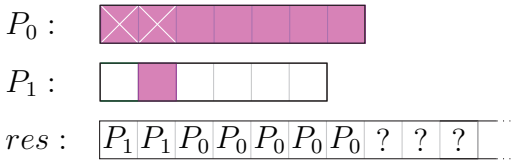
- Tasks are single-traced in the sense of control flow, so there are no conditional branches
- Bus accesses allow preemption
- If a task on a particular core is blocked, the whole state of the core stays unchanged which leads to timing compositionality
- Bus and processing units start at time 0
- Bus cycle as well as the cycles of the processing units have equal length and are synchronized
- The resource schedule can be chosen freely
  As we only want to consider finitely many tasks with finite length, this is possible with finite memory

5.2 Non-preemptive tasks

We made the assumption that the accesses of our tasks are preemptive which lead us to the term of reasonability as introduced in definition 3.1. If we do not allow preemption any more, several things could go wrong. We could possibly miss the optimal schedule if we just consider reasonable resource schedules with our reasonability definition. If we say we do not allow preemption this means that accesses really must not be preempted. If an access is granted the schedule must ensure that this access can be finished, too. One example which illustrates the loss of optimality in the other case is shown in figure 5.1 where we have on the one hand the optimal



(a) Optimal "reasonable" schedule



(b) Really optimal schedule

Figure 5.1: Non-preemptive accesses

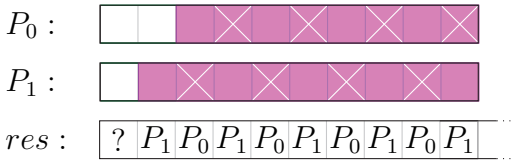


Figure 5.2: Infinite "reasonable" schedule

schedule within our reasonability definition and on the other hand the really optimal result.

We can also consider a different scenario of non-preemptive accesses. If we would only say that an access which is not completely finished within its length just has to start at its beginning again than we can always find an optimal schedule within our reasonability. If we have a non-reasonable schedule then a schedule which starts the accesses at the non-reasonable points and does not complete them leads to the same behaviour. But in this scenario we cannot predict that a reasonable resource schedule leads to a finite $WCET$ of the system because it is possible that two access interrupt each other in a way such that they are restarted again and again. Figure 5.2 sketches such a schedule. We actually have two accesses, each of length 2, but as the resource schedule alternates between them, they have to restart from scratch again and again.

We can easily fix this problem by changing our definition of reasonability to the following one.

Definition 5.1. *$Reasonability_{non-preemptive}$*
A resource schedule res is *$reasonable_{non-preemptive}$* if and only if:

$\forall t \in \mathbb{N}$: Access σ with a naive length of l and a start time of t is blocked
$\Rightarrow$ $\exists$ access σ' with start time $t' \in [t, t + l)$ such that σ' is not blocked

This definition allows us to interrupt or block accesses if this is needed as well as it prevents from alternating restarting of accesses which leads to infinite execution times. So if we adapt our algorithm to this term of reasonability, we can easily handle non-preemptive accesses in our model.

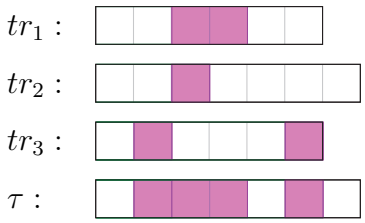


Figure 5.3: Merge by simple overlay

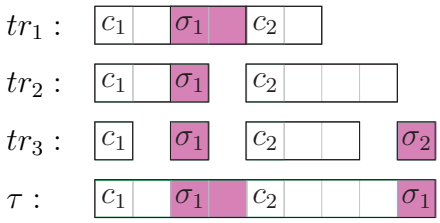


Figure 5.4: Merge by full-aligned overlay

5.3 Allow conditional branches

Another important question is how can we now let branching tasks fit into our model and like this into our analysis? If we want to stay sound, we have to consider all feasible traces. But as analysing all possible execution traces would be too time-consuming, we define *merge*-operators ME

$$ME : \mathcal{P}(Traces) \rightarrow Tasks$$

which takes n traces $tr_i \in Traces, i = 1, \ldots, N$ as input and computes a task $\tau \in Tasks$ such that property (5.1) is always fulfilled.

$$\forall (sys, res) \in S \times R \ \forall i = 1, \ldots, N : WCET_{sys,res}(tr_i) \leq WCET_{sys,res}(\tau) \quad (5.1)$$

There are several ways to define such *merge*-operators. We will present three of them.

Overlapping merge

Intuitively the first operator just overlays the traces because formally we place an access at time t if and only if there is an access at time t in one of the traces. This merging step is illustrated in figure 5.3. Tis can introduce pessimism as there might be an infeasible sequence of bus accesses.

Align all components

Another possibility is to enumerate the computation as well as te access blocks. Then we can identify according components and align them before we do again overlay the traces. This procedure is shown in figure 5.4 with the same traces as in the last example.

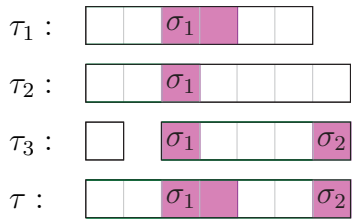


Figure 5.5: Merge by access-aligned overlay

Align the accesses

We can easily recognize that the last merge operator has some big disadvantages. When we align the last access σ_3, we see it is not needed that the last access is placed so late. Furthermore, we do not need to align computation time blocks. If a trace's access is finished although the merged task's access is not, it should be allowed that the trace can start with its computation block as it does not matter if the bus is granted in this time or not. This means that it is also enough to align the accesses and then do the overlay step as shown in figure 5.5. We will stay sound with this *merge*-operator and the outcoming task will never have a longer $WCET$ than the task computed with the full-aligned overlay merge.

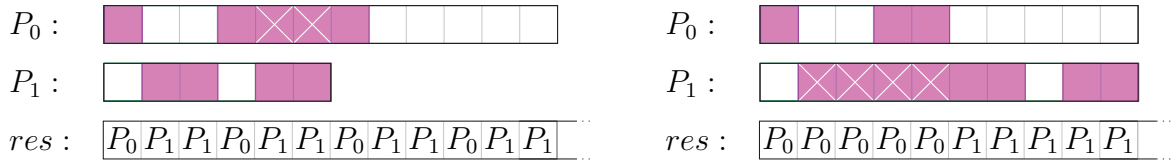
Of course all those merge operators introduce some pessimism. Either we construct an infeasible sequence of accesses or we increase the naive $WCET$ or possibly even both. But as we are mostly interested only in heuristic schedules anyway, there might also be an unsafe *merge*-operator which leads to better results. Therefore this is one open point of possible improvements.

5.4 Less granular resource schedules

We made the assumption that we can construct our resource schedule as exact as we need to. But it might be the case that the hardware does only allow periodic schedules. There again our definition of reasonability will keep us away from the optimal result as can be seen in figure 5.6.

The example considers a scenario where every processing unit has one time block per period. The length and the order of those blocks are variable. After a period is finished, the following period behaves of course exactly the same. This allows only a few reasonable resource schedules and we might also consider cases where no reasonable resource schedules are possible like in figure 5.7. In this example, we make the only available reasonable decisions until point in time 4. But those decision already defined the periodic schedule as the first period is already finished. So we cannot vary the further points in time any more which leads to a non-reasonable decision at the last two blockings of P_0. So we see that there is no reasonable schedule possible.

The general problem here is that every decision in the beginning affects interferences in the end, so we can only decide if the decision was optimal when we have completed the schedule.



(a) Optimal "reasonable" schedule

(b) Really optimal schedule

Figure 5.6: Periodic resource schedules

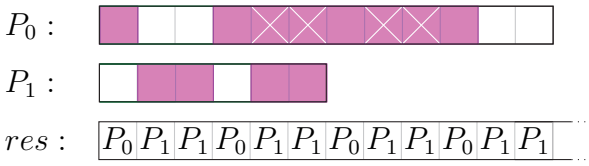


Figure 5.7: No reasonable periodic resource schedule possible

Chapter 6

Summary

We introduced and evaluated many things, so we want to give a short summary at the end of this thesis. In the very beginning, we adapted the *work-conserving* property which is characteristic for dynamic arbitrations to our *TDMA* arbitration. This led us to the term of reasonability. In the further sections, we introduced some techniques on how to speed up the exhaustive optimization algorithm to find an optimal schedule. We saw that we can reach a significant speedup when the bus load stays low. But the main goal of the thesis was to construct effective and efficient heuristics to approximate the optimal schedule. We saw that we can reach good results if we take the access distribution into consideration when deciding how to place the tasks in a system schedule. We characterized settings where our heuristics performed better than another heuristic which did not consider accesses at this part of their heuristic. We also saw a short overview which should ease the decision about which one of the presented approaches to take. And finally, we provided a brief outlook on how to realize a concrete system to fit into our abstract model.

Bibliography

[Amd67] Amdahl, Gene M.: Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. New York, NY, USA : ACM, 1967 (AFIPS '67 (Spring)), 483–485

[FKY08] Funaoka, Kenji ; Kato, Shinpei ; Yamasaki, Nobuyuki: Work-Conserving Optimal Real-Time Scheduling on Multiprocessors. In: *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*. Washington, DC, USA : IEEE Computer Society, 2008 (ECRTS '08). – ISBN 978–0–7695–3298–1, 13–22

[FSS07] Fedorova, Alexandra ; Seltzer, Margo ; Smith, Michael D.: Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In: *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. Washington, DC, USA : IEEE Computer Society, 2007 (PACT '07). – ISBN 0–7695–2944–5, 25–38

[GJ90] Garey, Michael R. ; Johnson, David S.: *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA : W. H. Freeman & Co., 1990. – ISBN 0716710455

[GRG11] Grund, Daniel ; Reineke, Jan ; Gebhard, Gernot: Branch Target Buffers: WCET Analysis Framework and Timing Predictability. In: *Journal of Systems Architecture* 57 (2011), Nr. 6, 625–637. `http://dx.doi.org/10.1016/j.sysarc.2010.05.013`. – DOI 10.1016/j.sysarc.2010.05.013. – ISSN 1383–7621

[HE05] Hamann, Arne ; Ernst, Rolf: TDMA Time Slot and Turn Optimization with Evolutionary Search Techniques. In: *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*. Washington, DC, USA : IEEE Computer Society, 2005 (DATE '05). – ISBN 0–7695–2288–2, 312–317

[NR98] Narasimhan, M. ; Ramanujam, J.: Improving the computational performance of ILP-based problems. In: *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*. New York, NY, USA : ACM, 1998 (ICCAD '98). – ISBN 1–58113–008–2, 593–596

[NW88] Nemhauser, George L. ; Wolsey, Laurence A.: *Integer and combinatorial optimization*. New York, NY, USA : Wiley-Interscience, 1988. – ISBN 0–471–82819–X

[PSC+10] Pellizzoni, Rodolfo ; Schranzhofer, Andreas ; Chen, Jian-Jia ; Caccamo, Marco ; Thiele, Lothar: Worst case delay analysis for memory interference in multicore systems. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. 3001 Leuven, Belgium, Belgium : European Design and Automation Association, 2010 (DATE '10). – ISBN 978–3–9810801–6–2, 741–746

[RAEP07] Rosen, Jakob ; Andrei, Alexandru ; Eles, Petru ; Peng, Zebo: Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In: *Proceedings of the 28th IEEE International Real-Time Systems Symposium*. Washington, DC, USA : IEEE Computer Society, 2007 (RTSS '07). – ISBN 0–7695–3062–1, 49–60

[RGG+12] Radojković, Petar ; Girbal, Sylvain ; Grasset, Arnaud ; Quiñones, Eduardo ; Yehia, Sami ; Cazorla, Francisco J.: On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. In: *ACM Trans. Archit. Code Optim.* 8 (2012), Januar, Nr. 4, 34:1–34:25. http://dx.doi.org/10.1145/2086696.2086713. – DOI 10.1145/2086696.2086713. – ISSN 1544–3566

[SCT10] Schranzhofer, Andreas ; Chen, Jian-Jia ; Thiele, Lothar: Timing Analysis for TDMA Arbitration in Resource Sharing Systems. In: *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA : IEEE Computer Society, 2010 (RTAS '10). – ISBN 978–0–7695–4001–6, 215–224

[SPC+11] Schranzhofer, Andreas ; Pellizzoni, Rodolfo ; Chen, Jian-Jia ; Thiele, Lothar ; Caccamo, Marco: Timing Analysis for Resource Access Interference on Adaptive Resource Arbiters. In: *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA : IEEE Computer Society, 2011 (RTAS '11). – ISBN 978–0–7695–4344–4, 213–222

[TFW00] Theiling, Henrik ; Ferdinand, Christian ; Wilhelm, Reinhard: Fast and Precise WCET Prediction by Separate Cache and Path Analyses. In: *Real-Time Systems* 18 (2000), May, Nr. 2/3

[WAB+10] Wilhelm, Reinhard ; Altmeyer, Sebastian ; Burguière, Claire ; Grund, Daniel ; Herter, Jörg ; Reineke, Jan ; Wachter, Björn ; Wilhelm, Stephan: Static Timing Analysis for Hard Real-Time Systems. In: *VMCAI*, Springer Verlag, 2010, S. 3–22

[WEE+08] Wilhelm, Reinhard ; Engblom, Jakob ; Ermedahl, Andreas ; Holsti, Niklas ; Thesing, Stephan ; Whalley, David ; Bernat, Guillem

; Ferdinand, Christian ; Heckmann, Reinhold ; Mitra, Tulika ; Mueller, Frank ; Puaut, Isabelle ; Puschner, Peter ; Staschulat, Jan ; Stenström, Per: The Worst-case Execution Time Problem—Overview of Methods and Survey of Tools. In: *ACM Transactions on Embedded Computing Systems (TECS)* 7 (2008), Nr. 3. `http://dx.doi.org/10.1145/1347375.1347389`. – DOI 10.1145/1347375.1347389

[WGR+09] Wilhelm, Reinhard ; Grund, Daniel ; Reineke, Jan ; Schlickling, Marc ; Pister, Markus ; Ferdinand, Christian: Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-critical Embedded Systems. In: *IEEE Transactions on CAD of Integrated Circuits and Systems* 28 (2009), July, Nr. 7, S. 966–978. `http://dx.doi.org/10.1109/TCAD.2009.2013287`. – DOI 10.1109/TCAD.2009.2013287

[WT06] Wandeler, Ernesto ; Thiele, Lothar: Optimal TDMA time slot and cycle length allocation for hard real-time systems. In: *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*. Piscataway, NJ, USA : IEEE Press, 2006 (ASP-DAC '06). – ISBN 0–7803–9451–8, 479–484

[ZBF10] Zhuravlev, Sergey ; Blagodurov, Sergey ; Fedorova, Alexandra: Addressing shared resource contention in multicore processors via scheduling. In: *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. New York, NY, USA : ACM, 2010 (ASPLOS XV). – ISBN 978–1–60558–839–1, 129–142