UPPSALA UNIVERSITY

Department of Computer Systems

The Slack Method: A New Method for Static Allocation of Hard Real-Time Tasks

Peter Altenbernd, Hans Hansson

Technical Report
DoCS 97/95   ASTEC 97/12

November 6th, 1997
ISSN 0283-0574

Contents

Abstract

This article presents and evaluates the *Slack Method*, a new constructive heuristic for the allocation (mapping) of periodic hard real-time tasks to multiprocessor or distributed systems. The Slack Method is based on task deadlines, in contrast with other constructive heuristics, such as *List Processing*. The presented evaluation shows that the Slack Method is superior to list-processing-based approaches with regard to both finding more feasible solutions as well as finding solutions with better objective function values.

In a comparative survey we evaluate the Slack Method against several alternative allocation techniques. This includes comparisons with optimal algorithms, non-guided search heuristics (e.g. *Simulated Annealing*), and other constructive heuristics. The main practical result of the comparison is that a combination of non-guided search and constructive approaches is shown to perform better than either of them alone, especially when using the Slack Method.

1 Introduction

Many embedded (mechatronic) real-time systems are by nature distributed or parallel, and todays applications (e.g. cars) frequently involve many micro-processors in order to serve different special-purpose demands (e.g. for cruise control, ABS, engine management, etc.). The processors are typically interconnected to form powerful but complex computation environments.

In order to efficiently and safely manage complex control applications in such a networked environment, support for efficient static *allocation* (sometimes referred to as *multiprocessor scheduling*[1]) is required, for a set of communicating tasks with known computation times and hard deadlines. An ideal technique would be an allocation technique which, based on information about available resources, automatically assigning tasks to processors. This allocation should be optimal in the sense that it should allocate as few resources as possible, while guaranteeing that requirements on tasks (e.g. deadlines) are satisfied. Obviously, within large applications this problem cannot be overcome by manual allocation, since this is too unsafe, too time consuming, too inefficient etc..

Unfortunately, the above stated allocation problem is known to be NP-hard [32], meaning that it is likely to be computationally infeasible to find an optimal assignment for non-trivial scenarios. The practically useful approach in this type of situation is to look for a heuristic approach capable of finding an acceptable solution reasonably fast in most realistic cases. The *Slack Method* presented in this article is an example of such a heuristic method.

Most of the work in the field of task allocation is focused on load balancing with the hope to keep the network load low, e.g. [11, 16]. However, for the problem addressed in the context of real-time systems the most important aspect is to guarantee hard real-time requirements (i.e. deadlines), while (typically) minimising the use of hardware resources.

To cope with complexity, *non-guided search* heuristics (see Section 3.2), like Simulated Annealing, can be used. These methods always maintain a current assignment which is slightly modified in order to get a better one for the next iteration step. However, these techniques often have difficulties in finding a good assignment, or even in finding any feasible assignment (i.e. an assignment in which all deadlines are met) for large real-time applications. This is due

[1] In this article the term *allocation* is used for the assignment of tasks to processors, and the term *scheduling* is used for dispatching the tasks on a single processor. Further, we will use the term *allocation* for mapping the tasks off-line (statically).

to the unusual solution space for hard real-time allocation problems, with lots of very high peaks, and a number of solutions hiding behind them. This means that the non-guided search methods have problems settling into one of the solutions, and furthermore, in cases when a feasible assignment is found, large computation times are typically needed.

Another group of techniques to handle the complexity is *constructive* heuristics (see Section 3.3), which try to build a feasible assignment based upon some best-effort policy. These methods often handle scheduling and allocation simultaneously, e.g. by searching for a feasible static cyclic schedule with minimal length. They are often based on one or more global priority lists or stacks, where in each processing step, the task with the highest priority is allocated to a certain processor (*List Processing* (see Section 3.3), sometimes referred to as *List Scheduling*[2]). These methods often suffer from poor results with regard to resource-based objective functions, since the schedule length is their main concern.

This article mainly provides the following contributions:

1. A presentation of the *Slack Method*, a new constructive heuristic which is *not* based on List Processing. Instead, it uses *critical path clustering* (e.g. [29]) in connection with exploiting slack values. Since the Slack Method focuses mainly on what is most important in hard real-time systems, namely the deadlines, it is for such cases superior to List Processing. This is clearly demonstrated by a benchmark evaluation in Section 5.

2. A survey of related work on different allocation techniques, slightly focusing on non-guided search (see Section 3).

3. An evaluation of the Slack Method, in which a set of benchmark application scenarios are used to compare the behaviour of the Slack Method with that of some of the currently most popular allocation methods. This evaluation shows that a combination of the Slack Method with a non-guided search method outperforms both purely constructive, and purely non-guided search techniques.

The rest of the article is organised as follows: Firstly, after a detailed problem definition and a list of basic assumptions in Section 2, the article deals with related work on allocation techniques: optimal approaches in Section 3.1, non-guided search techniques in Section 3.2, and constructive methods in Section 3.3. The Slack Method is presented in Section 4. In Section 5 some comparative experimental results are discussed. Finally, Section 6 gives concluding remarks.

2 Basic Assumptions and Problem Description

The following task model is assumed:

- A set V of strictly periodic tasks. Each task $p \in V$ has
  - a *period length*, T_p
  - a *worst-case execution time*, C_p (to be estimated by timing analysis programs (e.g. [3, 9, 21, 24, 40, 47])),

[2] In order to not get confused with the notions of *allocation* and *scheduling*, the term *List Processing* was preferred in this article.

  - a relative *deadline* (the *hard real-time condition*), D_p, before which p has to complete its computation, and which is required not to exceed the period, i.e. $D_p \leq T_p$.

- Within each period a task may receive data from one or more tasks at its beginning, and it may send data to one or more tasks at its end. A task is not invoked before its input data is completely received. Precedence constraints result from communicating tasks that share the same periodicity. Communication between tasks with different periods is asynchronous, non-blocking, and will hence not impose any precedence constraints.

- The tasks are to be executed on a multiprocessor or distributed system consisting of a set X of identical processors.

- To reduce complexity, we treat allocation as an issue separated from that of scheduling. Scheduling is locally performed for each processor, using fixed priority preemptive scheduling, with a *deadline monotonic* priority assignment [33] (giving higher priority to tasks with shorter deadlines). The communication-imposed precedence constraints can be mapped to deadlines which then can be used as the basis for priority assignments [2, 13]. Deadline monotonic scheduling can be used both for building off-line cyclic schedules as well as being a policy of a run-time scheduler.

- In real systems there are also communication delays that could be associated to precedence relations. In order to simplify the presentation, communication delays are ignored in this article. However, they could be included, which is explicitely remarked when needed.

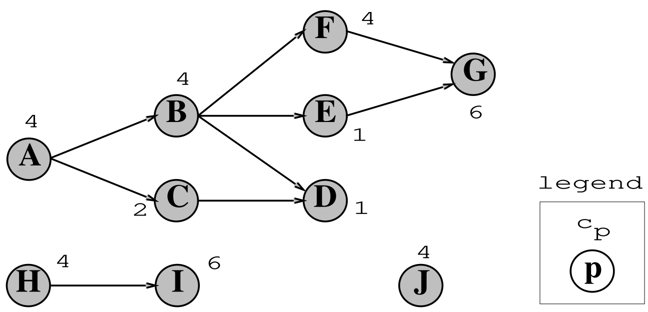

Figure 1: An example task graph

The tasks form a simple, directed, acyclic, labeled *graph* $G = (V, E)$ where V is the set of tasks, and E is the set of edges representing precedences, i.e. E is a set of ordered pairs, where (p_1, p_2) means that there is a precedence relation requiring p_1 to be executed before p_2. We will use $(p_1 \rightarrow p_2)$ to denote (p_1, p_2). The labels at the vertices represent the computation times (C_p) of the tasks. An example of a graph representing a number of tasks with the same period length is shown in Fig. 1. Similary, the network of processors is represented by a graph $N = (X, L)$, where X is the set of processors, and L is the set of communication links.

The *hard real-time allocation problem* is the problem of assigning a set of communicating periodic tasks with hard deadlines to a given network of identical processors, so that it is guaranteed that no deadline will be missed when running the system. Typically, for an embedded system the allocation has an associated objective function (e.g. minimising the number of processors needed) to distinguish good assignments from less good ones. However, in this article we do not focus on a particular objective function since such at function highly depends on the application context.

3 Related Work

This section gives an overview of related work on methods dealing with the hard real-time allocation problem.

3.1 Optimal Algorithms

The term *optimal* has two meanings in the context of allocation. Firstly, it sometimes defines the ability of finding a feasible assignment whenever one exists, e.g. using branch-and-bound [50], or network flow techniques [17]. Secondly, it sometimes means to find the best possible assignment, with regard to a certain objective function e.g. as in [7, 38]. However, the main problem with the latter methods is complexity, even when the goal is slightly relaxed to that of finding an assignment which with high probability fulfills certain constraints, as in [26], or by reducing the problem to a polynomial time sub-problem, as in [7]. The number of possible allocations is $O(n^m)$ (n = number of processors, m = number of tasks). For the largest example which was tested during this work (*bus7* in Section 5) this gives n^{1710} ($> 10^{301}$ if $n \geq 2$) possible allocations. However, the largest test example found in the published papers on optimal algorithms is of only 80 tasks. This may be sufficient for todays applications but probably not for more complex future demands. Hence optimal algorithms are not further considered in this article.

3.2 Non-Guided Search Techniques

Non-Guided Search Techniques are general purpose optimisation heuristics which have successfully been applied to a number of complex optimisation problems, such as the travelling salesman's problem, or floor planning. The following sections present a number of well-known non-guided search techniques (Steepest Descent, Tabu Search, Simulated Annealing, Threshold Accepting and Genetic Algorithms) with the purpose of showing how they can be applied to the hard real-time allocation problem. A comparative evaluation is presented in Section 5.

Many of these methods have also been considered in Nicholson's survey of non-guided search techniques [36], where the real-time allocation problem was used as an application example. However, Nicholson's paper does not deal with the difficulties these techniques often have in finding a good assignment, or even in finding any feasible assignment, for large real-time applications.

Each non-guided search has to be given an initial assignment from where the search starts. A reasonable initial assignment in this context is to assign each subgraph (= set of dependent tasks) to a single processor. Then, within each iteration step different derivable alternatives of changing the current assignment are examined. A *derivable alternative* is a move of a task to a different processor (or in the general case a change of any other property depending on the concrete problem to solve). To check whether an alternative is feasible or not, a run-time efficient schedulability test has be used (such as the one presented in [2]). This test has also to include the schedulability analysis of the communication network if communication delays are included (as in [48]). In order to help the search finding better assignments, the number of deadline misses is included as a penalty into the objective function value. However, only feasible alternatives are reported as new solutions.

Although the following sections cover a number of non-guided search methods, the survey cannot be comprehensive due to the huge number of publications presenting related, or new methods in this field of research. However, as the overview shows, the considered methods mainly suffer from the same disadvantages when dealing with hard real-time problems, in particular with unusual solution space. Hence, it is likely that this is the case for any other non-guided search technique. Further, tackling the hard real-time allocation problem with the help of methods based on Integer Linear Programming (as in [7]), or Neural Networks (as in [8]) was not considered due to complixity reasons (for more details please refer to [4]).

3.2.1 Steepest Descent

The simplest non-guided search approach is to consider an arbitrary derivable alternative to the current assignment, and use it for the next iteration step if it improves the objective-function value. This approach is called *Steepest Descent* since the objective function value is steadily reduced (or *Hill Climbing* if the objective function value is steadily increased). The algorithm runs until no further improvement is possible.

One disadvantage of Steepest Descent is its in-ability of overcoming walls between local optima. Surprisingly, Steepest Descent sometimes delivers quite good results, since compared to the other non-guided search techniques, its inability to overcome local optima may become an advantage. The other methods try to use random factors to overcome walls, which sometimes leads to a situation where the method jumps away just before reaching a local optimum.

The main problem with Steepest Descent is its run time when applied to large applications. Especially, to verify the termination condition ("*until no further improvement is possible*") can be very time consuming, since all possible derivable alternatives have to be examined. Hence, a simplification of the termination condition is usually required. Unfortunately, such conditions a typically quite difficult to derive.

An even bigger problem is, that when started with a non-feasible initial assignment, Steepest Descent sometimes runs into a "dead end", without having found any feasible assignment, even in cases when many such exist.

3.2.2 Tabu Search

Tabu Search [18] has evolved from Steepest Descent in order to improve run-time behaviour, as well as to escape local optima. It uses a short-term memory of recent solutions to guide the search. Like Steepest Descent, Tabu Search considers an arbitrary derivable alternative of the current assignment that will be chosen if better. In order to overcome local optima the value of the elected alternative might also be worse. In contrast to Steepest Descent, Tabu Search does not consider those alternatives which are currently present in the short-term memory, the *tabu list*. The tabu list is the result of user defined rules of which derivable alternatives are *tabu*. An alternative can become tabu for a couple of reasons, e.g. when the move has been tried previously. The alternative will then remain tabu for a number of iterations. In addition, Tabu Search can allow backtracking to previous assignments.

In [39] Tabu Search is used for the allocation of non-real-time tasks: A Tabu List for (task, processor) pairs is maintained. Such a pair is placed in the list whenever the move of a certain task to a certain processor has lead to a worse assignment. The same approach can be used to handle the hard real-time allocation problem. For handling further properties additional lists have to be used.

The run time of Tabu Search is reduced by forcing elements to stay longer in the list, with the drawback of delivering worse results. So it is quite difficult to find a good trade-off, i.e. a reasonable number of iterations an element has to stay on the list. Further, backtracking requires too much memory for very large applications, and hence should be avoided. In addition, though improved, Tabu Search mainly suffers from the same disadvantages as Steepest Descent, i.e. finding an appropriate termination condition is difficult, and it has a tendency to run into a "dead end", without having found any feasible assignment.

3.2.3 Simulated Annealing

Simulated Annealing [30] is currently probably the most popular non-guided search technique. It borrows ideas from statistical physics. Its main property is the incorporation of random jumps from local optima to potential new solutions. As the algorithm progresses, this ability is lessened, by reducing a *temperature factor*, making longer jumps less likely. This is called *cooling* the system until the system *freezes* into a low objective function value. However, our experimental evaluation has shown that better results are delivered with regard to the hard real-time allocation problem, when the concept of strictly *cooling* the system is relaxed, i.e. *increasing* the temperature is allowed when no more improvements are achieved up to a maximum temperature. Within each iteration step a derivable alternative is evaluated: Alternatives with better objective function values are accepted in any case, while alternatives with worse function values are accepted with regard to the temperature factor.

In [12, 14, 35, 49] Simulated Annealing is used for a system with hard deadlines, where deadline misses are included as a penalty into the objective function value, while in [34] the goal is simply to minimise the schedule length. While [35, 49] only deal with the actual allocation problem, [12, 14] handle the allocation and the scheduling problem simultaneously, and [34] first handles allocation and then scheduling. Furthermore, these methods differ in their optimising goals: minimum communication load in [35, 49], minimum schedule length in [12, 34], and minimum jitter in [14].

Simulated Annealing is very sensitive to its parameters: the initial temperature, the way the temperature is modified, and the termination criterion. It is very difficult to find parameters which fit well for all applications. The actual advantage of Simulated Annealing is its ability to escape local optima by random jumps. However, this ability sometimes becomes a disadvantage, since, especially within larger examples, Simulated Annealing tends to jump from peak to peak rather than settling in between, resulting in large computation times or non-feasible assignments.

3.2.4 Threshold Accepting

Threshold Accepting [15] is very similar to Simulated Annealing. Like Simulated Annealing it slightly modifies the current assignment, and accepts the new one if it is a better assignment. If the new assignment is not better, Threshold Accepting only accepts it if its objective function value is within a *threshold* value for the old one. As the algorithm progresses, the threshold is further reduced, making jumps less likely.

As with Simulated Annealing, Threshold Accepting delivers better results with regard to the hard real-time allocation problem, when the concept of strictly *reducing* the threshold is relaxed, i.e. *increasing* the threshold is allowed when no more improvements are achieved up to a maximum value.

The inventors of Threshold Accepting claim that *"Threshold Accepting yields better results than Simulated Annealing"* [15]. However, Threshold Accepting mainly suffers from the same problems as Simulated Annealing, i.e. sensitivity to variations of parameters (the initial threshold, the way the threshold is modified, and the termination criterion) as well as problems in finding feasible assignments for large application examples.

3.2.5 Genetic Algorithms

Conceptually, Genetic Algorithms [19] are based on Darwin's evolution theory of the survival of the fittest. Darwin's theory is applied by regarding a number of potential solutions as *individuals* forming a *population*. Couples of individuals can create *children*, i.e. new individuals. The creation of a new individual is performed by applying a *crossover* operator to the *genes* of the parents, i.e. the properties of the parents are combined. In addition, the genes of the new individual may *mutate*, i.e. they are slightly modified. As in nature this theory relies on the *selection* of the fittest, i.e. the best solutions will succeed, and on the assumption, that strong parents are likely to have strong children.

In [6, 20, 43] Genetic Algorithms have been used for the allocation of real-time tasks. Within each iteration step two of the current assignments are randomly chosen for the purpose of creating two new individuals (assignments) using the crossover operator. In [6, 20] the representation to perform the crossover operator is based on list schedules on each processor. These lists may be seen as chromosomes, where the tasks inside represent genes. These lists (chromosomes) are crossed over resulting in new lists (chromosomes). The tasks (genes) before a randomly chosen *crossing-over point* keep their position, while the tasks (genes) behind it swap sides. So the crossing over of two allocations will result in two new allocations. In addition, a potential new solution may be slightly modified (mutation), meaning that a task may be moved to another processor, or that any other property is modified. If the objective function values of some children are better than those of their parents, they will replace their parents (*selection*).

In [43] the use of the chromosomes is different: Each assignment is represented by only one chromosome which involves all information in an encoded manner. This technique is more compact and it is easier to compute the crossover point, but it requires more effort to encode and decode the chromosomes.

Genetic Algorithms mainly suffer from the same disadvantages as other non-guided search techniques, i.e. sensitivity to variations of parameters (the population size, the number of crossing-over points, the selection criterion, the termination criterion, the mutation probability, etc.) as well as problems in finding feasible assignments for large applications. In addition, the basic assumption, that strong parents are likely to have strong children, certainly is not true for the allocation of hard real-time tasks. In that problem the crossing-over is more or less an arbitrary mix rather than a combination of good properties. Hence, as our experiments show, Genetic Algorithms deliver better results when using a high mutation probability. Furthermore, when dealing with large applications, the program might get into trouble with memory limitations, so it only allows a small population (about ten). On the positive side, Genetic Algorithms escape local optima quite well.

3.3 Constructive Heuristics

The purpose of constructive heuristics is to find any feasible assignment by some simple best-effort strategy, usually avoiding to spend too much time in optimising the objective function. Most constructive heuristics known to the authors are related to *List Processing* [1, 10, 28, 37, 41, 42, 44].

These methods use one or more global priority lists (or stacks) in order to decide which task to be placed next on which processor. The various List-Processing approaches mainly differ in the priority function used to build the list(s). List Processing is very sensitive to the implementation of the priority function, and to the way the destination processor is chosen. Problems increase when different criteria have to be fulfilled concurrently. Whenever the programmer wants to improve the behaviour with regard to a certain condition, another condition is often missed. This may be overcome by allowing the algorithm to backtrack to former assignments in order to try a different policy, which usually leads to a drastical increase of run time. Furthermore, in [1, 41] it is reported that backtracking often fails to improve the results of List Processing. The evaluation of a rerun policy in [44] was inconclusive since no clear improvements were observered, because the method was only demonstrated using a quite small example (*tind* in Section 5). If used without backtracking or a rerun policy, List Processing is very fast but suffers from the risk of not finding any feasible assignment, or resulting in a poor allocation objective function value.

By merging a number of tasks (called *reduction* or *clustering*) [28, 29, 37, 45, 46, 42] the size of the task graph can be considerably reduced so that the allocation heuristic processes much faster. Clustering may also be used to preprocess List Processing. Most of these techniques are based on *critical path clustering* (e.g. [29, 45, 42] which is also used in one of the reduction steps of the Slack Method. Critical path clustering is based on a simple method to find the longest path in a graph, the *Critical Path Method* (CPM) (e.g. [25]). However, none of the mentioned clustering techniques focuses on deadlines, which is supposed to be most important in hard real-time systems. In contrast, this is exploited by the Slack Method.

4 The Slack Method

This section introduces a new constructive heuristic, called the *Slack Method*, for static allocation of hard real-time tasks. The Slack Method has two goals: First, the construction of any feasible assignment, and second, to limit the use of resources. Though mainly focusing on the first, the resulting assignment is likely to correspond to a reasonable fulfillment of the objective to minimise resources usage.

The Slack Method (see overview in Fig. 2) is divided into two sub-steps named *Graph Reduction* and *CP-Mapping*. During the Graph Reduction different *transactions* (= sets of tasks with the same period) are treated separately. This is desirable in multi-period systems since it avoids exhaustive exploration of all task activations during the least common multiple (LCM) of the period lengths. The sizes of the graphs are reduced by the *"Graph Reduction 1"* step (see Fig. 2) using the *critical path clustering*, and then by the *"Graph Reduction 2"* step (see Fig. 2) exploiting slack values. Slack values (i.e. the amount of time a task can be delayed without missing its deadline) are used to fill slack intervals with other tasks. The result is a more compact graph with guaranteed timing properties (called the *reduced critical-path (CP) graph* in Fig. 2). Due to its smaller size this graph can more efficiently than the original task graph

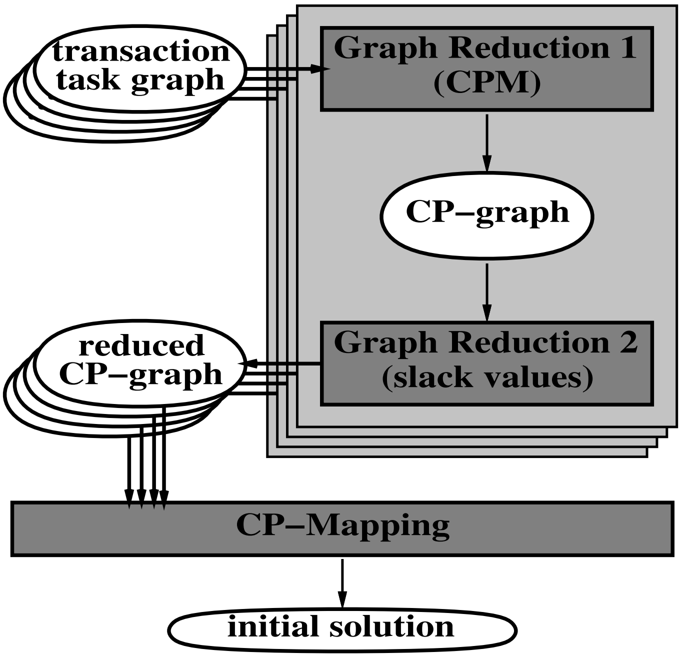

Figure 2: Overview of the Slack Method

be mapped on the processor network. Each reduced CP-graph (representing one transaction) is mapped by the *CP-Mapping* onto the processor network. In the following sections we will present all steps in more detail.

4.1 Graph Reduction

In this step the critical path clustering is used to transform the graph to a so-called *CP-graph* which is a smaller graph that can more easily be mapped on the processor network. In general, graph reduction means clustering of communicating (or precedence-constrained) tasks. To simplify the presentation, communication delays are ignored during our graph reduction, but could be included into the corresponding definitions 2 and 3 below. Anyway, communication interference is intentionally ignored during graph reduction, which is one of the reasons why the Slack Method is considered to be a heuristic.

We will now introduce some concepts needed to define the reductions.

A *path* is a sequence of edges through the graph, starting with a source vertex (i.e. a vertex without predecessors) and ending with a sink vertex (i.e. a vertex without successors), i.e.

Definition 1 (path): *A* path *P in a graph $G = (V, E)$ is a sequence of vertices $(p_1, p_2, \ldots, p_{|P|})$ such that $\forall i : 1 \leq i < |P|$: $(p_i \to p_{i+1}) \in E$, and $\nexists q$: $(q \to p_1) \in E$, and $\nexists q$: $(p_{|P|} \to q) \in E$. Furthermore, we use V_P to denote the set of vertices of P, and E_P to denote the set of edges of P.*

The *static offset value*[3] (earliest start time) of a task p is the length of the longest partial path (i.e. incorporating the computation times of the vertices on that path) from a source vertex (i.e. a vertex without predecessors) to p, i.e.

[3]User given offset values can obviously easily be included.

Definition 2 (static offset): *The* static offset value *of task p, o_p, is defined as:*

$$o_p = \begin{cases} 0 & \text{if} \not\exists q : (q \to p) \in E \\ MAX\{o_q + C_q \mid (q \to p) \in E\} & \text{otherwise} \end{cases}$$

The *static deadline* of a task p defines the last completion time of p, such that p and all subsequent tasks of p have a chance of meeting their deadlines, i.e.

Definition 3 (static deadline): *The* static deadline *of task p, δ_p, is defined as:*

$$\delta_p = MIN\{\{\delta_q - C_q \mid (p \to q) \in E\} \cup \{D_p\}\}$$

Since the static deadline takes deadlines of subsequent tasks into account it may be distinct from the deadline D_p of the task model (see Section 2): $\delta_p \leq D_p$. This means that the static deadline adds structural information to the actual deadline D_p. The static deadline will be used as a point in time that must not be exceeded when applying the Slack Method.

The *task slack* is the time difference between the earliest possible finishing time of a task and its static deadline, i.e.

Definition 4 (task slack): *The* slack *of task p, s_p, is defined as:*

$$s_p = \delta_p - (o_p + C_p)$$

All static offsets, static deadlines, and slack values in the example in Fig. 1 are given in Fig. 3. In the example it is assumed that the deadline of all sink vertices is 18. All these values can be computed in linear time with regard to the size of the graph ($O(|V| + |E|)$).

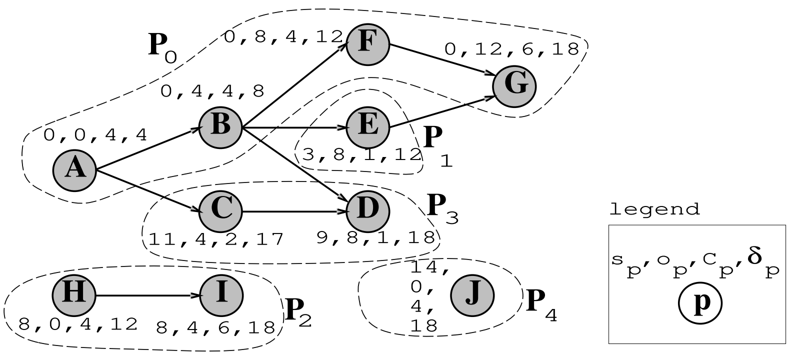

Figure 3: Critical paths for the example in Fig. 1

The *path slack* is the minimum slack of all vertices of the path, i.e.

Definition 5 (path slack): *Let P =(p_1, p_2, ..., $p_{|P|}$) be a path. The* slack *of P, S_P, is defined as:*

$$S_P = MIN\{s_{p_i} \mid 1 \leq i \leq |P|\}$$

Definition 6 (critical path): *The* critical path *P in a graph $G = (V, E)$ is the path with the smallest slack, and for which, for all edges $(p \to q) \in E_P$ task q has the smallest slack of any successor of p, and task p has the smallest slack of any predecessor of q, i.e.*

$$\forall (p \to u) \in E : s_q \leq s_u, \text{ and } \forall (v \to q) \in E : s_p \leq s_v$$

The following algorithm generates a list of paths in the graph G ordered by criticality.

Algorithm 7 (Graph Reduction 1):

Input: *task graph $G = (V, E)$ representing one transaction*
Output: *sorted list of paths $(P_0, P_1, ...)$ with the most critical first*
Goal: *compilation of task graph to CP-graph*
Method: *critical path clustering*

1. $i := 0$;
2. **while** $V \neq \emptyset$
2.1. **do** *use CPM to determine critical path P_i in G;*
2.2. *remove all vertices $p \in V_{P_i}$ and edges $(p \to q), (q \to p) \in E$ from G;*
2.3. *$i := i + 1$;*
2.4. **done***;*

The cost for finding a critical path grows linear with the size of the graph, i.e. $O(|V| + |E|)$. Hence the total cost for Algorithm 7 is $O(|V_{CP}| * (|V| + |E|))$. The critical paths P_0, ..., P_4 of the example in Fig. 1 are shown in Fig. 3.

Next, the vertices of each critical path are clustered in order to form a new graph, the *Critical-Path (CP) graph.* In the CP-graph each critical path represents one vertex, using *undirected* edges to represent the original connectivity, i.e.

Definition 8 (CP-graph): *The* CP-graph *is the* undirected *graph*

$$G_{CP} = (V_{CP}, E_{CP}) \text{ with } V_{CP} = \{P_0, P_1, ...\}, \text{ and}$$
$$E_{CP} = \{(P_i, P_j) \mid ((u \to v) \in E \text{ or } (v \to u) \in E), u \in P_i, v \in P_j, i \neq j\}$$

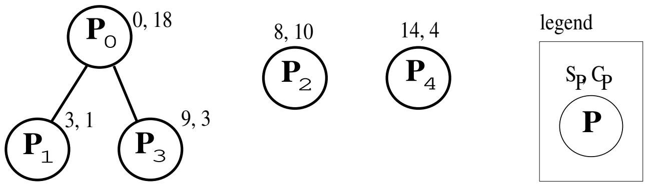

Figure 4: The corresponding CP-graph obtained from the paths in Fig. 3

The resulting CP-graph of the example in Fig. 3 is shown in Fig. 4.

Each CP-graph vertex is labelled with the computation time of the corresponding path, i.e.

Definition 9 (CP computation time): *Let P be a CP-graph vertex. The computation associated time, C_P, is the sum of the computation times of all tasks in the critical path P.*

Fig. 4 also shows all corresponding slack values and computation times. The deadlines are no longer of interest, since they are captured by the corresponding slack value (because we only deal with tasks with the same period).

THEOREM 10: *If each CP-graph vertex (representing a number of tasks) is exclusively allocated to a single processor, the resulting assignment is feasible, i.e. no deadlines will be missed (provided that there is a feasible assignment at all, i.e.* $\forall p \in V : s_p \geq 0$*).*

Proof: Each CP-graph vertex represents a non-interfered sequence of computations. Hence the *worst-case response time* of each $p \in V$ is $R_p = o_p + C_p$. From Definition 4 follows:

$$\begin{aligned} \delta_p &= s_p + o_p + C_p \geq \quad (\text{since } s_p \geq 0) \\ &\geq o_p + C_p = R_p \end{aligned}$$

The goal of the second graph reduction step is to further reduce CP-graphs by joining some of the paths (i.e. CP-graph vertices), while still providing a feasible assignment. This has to be done with regard to two objectives: Firstly, and most importantly, keeping communication load low is of interest (since communication interference and overhead is not considered during graph reduction), i.e. keeping the number of CP-graph edges low. And secondly, it is the aim to minimise the number of processors. For these reasons we introduce a measurement for the *size* of a CP-graph which is used by the second graph reduction algorithm (Algorithm 12):

DEFINITION 11 (GRAPH SIZE): *Let* $G_{CP} = (V_{CP}, E_{CP})$ *be a CP-graph. The* size *of* G_{CP}, $SZ(G_{CP})$*, is defined as:*

$$SZ(G_{CP}) = c_e * |E_{CP}| + c_v * |V_{CP}|$$

The constants c_e and c_v in Definition 11 capture the relation between the importance of communication load and use of processors. High loaded communication networks can be taken into regard by increasing c_e. Values $c_e = 2$ and $c_v = 1$ turned out to be a reasonable trade-off during our test-case evaluation.

The basic idea of the second reduction step is to fill slack intervals of paths (see Definition 5) with other paths, i.e. joining paths to allow them to later be allocated to the same processor. The following algorithm performs this reduction:

ALGORITHM 12 (GRAPH REDUCTION 2):

Input: *CP-graph* $G_{CP} = (V_{CP}, E_{CP})$ *representing one transaction,*
$V_{CP} = \{P_0, P_1, ...\}$ *with* $\forall i : 0 \leq i < |V_{CP}| - 1 : S_{P_i} \leq S_{P_{i+1}}$

Output: reduced *CP-graph* $G_{CP} = (V_{CP}, E_{CP})$

Goal: *minimise* $SZ(G_{CP})$

Method: *deterministic Steepest Descent*

1. $min := SZ(G_{CP})$;
2. **for** $i := 0$ **to** $|V_{CP}| - 1$ **for** $j := i + 1$ **to** $|V_{CP}| - 1$ **if** $S_{P_i} \geq C_{P_j}$
2.1. **do** (* let $H = (V_H, E_H)$ be the result of joining P_i with P_j, i.e. *)
2.2. $V_H := V_{CP} - \{P_j\}$;
2.3. $E_H := E_{CP} \cup \{(Q, P_i) \mid (Q, P_j) \in E_{CP}\} - \{(Q, P_j) \mid (Q, P_j) \in E_{CP}\}$;
(* insert additional edges into $P_i \in V_H$: *)
2.4. **for all** $(u \rightarrow v) \in E_{P_i}$ *and* $w \in V_{P_j}$ **with** $\delta_u \leq \delta_w \leq \delta_v$
2.4.1. $E_{P_i} := E_{P_i} \cup \{(u \rightarrow w), (w \rightarrow v)\}$
2.5. *update slack values in H;*
2.6. **if** $SZ(H) < min$ (* new minimum found *)

2.6.1. **do** $min := SZ(H);$
2.6.2. $G_{CP} := H;$
2.6.3. **done**
2.7. **done**

Algorithm 12 joins two CP-graph vertices (paths), P_i and P_j with $i < j$, whenever the slack of P_i is greater or equal to the computation time of P_j, i.e. $S_{P_i} \geq C_{P_j}$ (see line 2). The resulting vertex (path) is P_i (see lines 2.1.–2.3.). To capture the ordering and interference induced by deadline monotonic scheduling when executing the tasks in P_i and P_j on the same processor, additional edges are included into E_{P_i} (see lines 2.4.–2.4.1.), i.e. $P_i = (..., u, w, v, ...)$. After each pairing of paths the static offset values and static deadlines are re-computed to update available slack taking into account the new edges (see line 2.5.). The run-time complexity of Algorithm 12 is $O(|V_{CP}|^2)$.

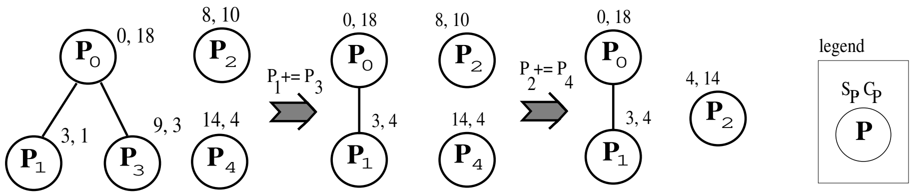

Figure 5: Reducing the CP-graph of Fig. 4

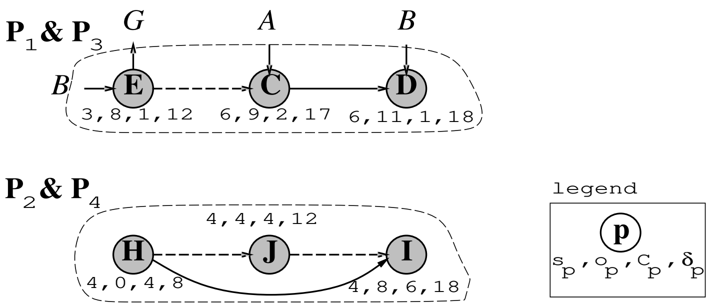

Figure 6: The union of P_1 & P_3, and P_2 & P_4

The CP-graph of the example in Fig. 4 is reduced twice as shown by Fig. 5. Starting with a size value of 9 (provided that $c_e = 2$ and $c_v = 1$), the vertices P_1 and P_3 are joined first (size value 6). Secondly, vertices P_2 and P_4 are joined resulting in a size value of 5. The unions of P_1 and P_3, and of P_2 and P_4 are shown in more detail in Fig. 6 (the additional edges are represented by dashed arrows) where all resulting deadlines, offsets, and slack values are given.

Algorithm 12 exploits the merging condition (*"$S_{P_i} \geq C_{P_j}$ with $i < j$"* in line 2.) in order to speed up computation. Without this condition each merge of CP-graph vertices would have to be explicitely checked for feasibility, with a resulting total complexity of $O(|V_{CP}|^3)$ for Algorithm 12. By the following theorem we prove the merging condition to be correct:

THEOREM 13: *If each vertex of the reduced CP-graph (i.e. the output of Algorithm 12) is exclusively allocated to a single processor, the resulting assignment is feasible, i.e. no deadlines will be missed (provided that there is a feasible assignment at all).*

Proof: It has to be shown that any merge of CP-graph vertices P_i, P_j will cause no deadline violation. The *conditions for merging* the vertices are:
$i < j$, $S_{P_i} \geq C_{P_j}$, $\forall k : 0 \leq k < |V_{CP}| - 1 : S_{P_k} \leq S_{P_{k+1}}$, and $\forall P \in V_{CP} : S_P \geq 0$. In the following I denotes P_i, and J denotes P_j, respectively. All tasks p in I and J are considered separately:

1. $p \in I$:

   In this case the maximum interference caused to p is: C_J, i.e. the worst-case response time of p is: $R_p = C_J + o_p + C_p$. It follows:

$$\begin{aligned} \delta_p &= s_p + o_p + C_p \geq && \text{(since } s_p \geq S_I \text{ (see Def. 5))} \\ &\geq S_I + o_p + C_p \geq && \text{(since } S_I \geq C_J \text{ (see merging conditions))} \\ &\geq C_J + o_p + C_p = R_p \end{aligned}$$

2. $p \in J$:

   Two further sub-cases are considered:

   (a) $C_I \leq S_J$:
   Analogously to the case $p \in I$.

   (b) $C_I > S_J$:
   The tasks $q \in I$ with deadlines greater than δ_p, or less or equal to δ_p are considered separately:

   i. $q \in I' = \{u \mid u \in I \text{ and } \delta_u > \delta_p\}$:
   By deadline monotonic priority assignment q will always be scheduled later than p (since $\delta_q > \delta_p$). Hence q does not interfere with p.

   ii. $q \in I' = \{u \mid u \in I \text{ and } \delta_u \leq \delta_p\}$:
   Let $q' \in I'$ be the task with the largest static deadline δ. The maximum interference caused to p is: $C_{q'} + o_{q'} - o_p$ (if positive, 0 otherwise). That means a deadline would only be missed under the *assumption*: $C_{q'} + o_{q'} - o_p > s_p$. It follows:

$$\begin{aligned} \delta_{q'} &= s_{q'} + o_{q'} + C_{q'} \geq && \text{(since } s_{q'} \geq S_I \geq C_J) \\ &\geq C_J + o_{q'} + C_{q'} \geq && \text{(since } C_J \geq C_p \text{ (see Def. 9))} \\ &\geq C_p + o_{q'} + C_{q'} = \\ &= C_{q'} + o_{q'} - o_p + C_p + o_p > && \text{(since } C_{q'} + o_{q'} - o_p > s_p) \\ &> s_p + C_p + o_p = \delta_p \end{aligned}$$

   It follows: $q' \in I'$: $\delta_{q'} > \delta_p$. This contradicts the definition of I'. By that the above assumption $(C_{q'} + o_{q'} - o_p > s_p)$ cannot be true. Hence, it follows: $C_q + o_q - o_p \leq s_p$.

   From (i) and (ii) follows: The worst-case response time of p is:
   $R_p \leq s_p + o_p + C_p = \delta_p$.

Due to the insertion of the additional edges by Algorithm 12 and the re-computation of all static offsets, static deadlines and slack values in each iteration step, all properties used by this proof will hold for each iteration step.

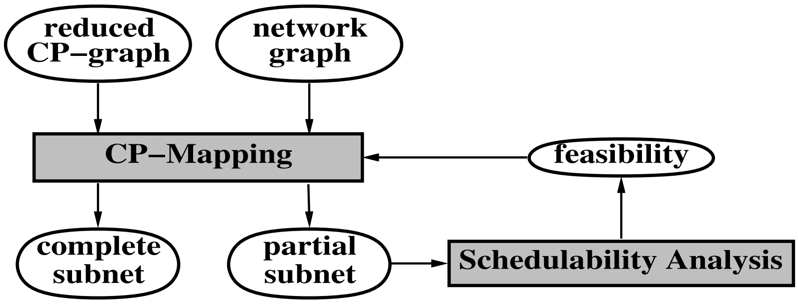

Figure 7: Overview of the transaction mapping

4.2 CP-Mapping

During this step all reduced CP-subgraphs[4] are mapped onto the target processor network.

The two reduction steps will in most cases drastically reduce the problem size (task graph size versus CP-graph size) even for large systems, as shown by our test-case evaluation in Section 5. Hence no serious complexity problems are likely to arise when using an optimal algorithm (i.e. backtracking) for mapping CP-graph vertices to the processor network. By "optimal" we mean that if there is any valid mapping, the algorithm will find it. Within each iteration step of the backtracking algorithm (i.e. when a CP-graph vertex is mapped) schedulability analysis is used to check if all deadlines are met (see Fig. 7). If that is not the case the algorithm backtracks to next possible partial assignment.

The CP-Mapping deals with communication (delays) in the following way: All tasks of adjacent CP-graph vertices will be placed either on the same processor, or on neighbouring processors in the processor network (referred to as *neighbourhood relation* in the following). By this assumption no message routing is necessary, which drastically reduces communication load[5]. Communication delay has also to be incorporated into the schedulability analysis of the communication network.

ALGORITHM 14 (CP-MAPPING):

Input: *list of CP-subgraphs* $G^1_{CP}, G^2_{CP}, ...$
with $G^i_{CP} = (V^i_{CP}, E^i_{CP})$, *and* $V^i_{CP} = \{P^i_0, P^i_1, ...\}$;
processor network graph $N = (X, L)$, $X = \{x_0, x_1, ...\}$

Output: *mapping* $(G^1_{CP} \cup G^2_{CP} \cup ...) \longrightarrow N$

Goal: *find a feasible mapping so that the neighbourhood relation is fulfilled*

Method: *backtracking*

(* recursive backtracking procedure: *)
1. **function** *try*(P, x): *boolean*
(* try to place CP-vertex P on x or next to x *)
1.2. **do** $Y := \{x\} \cup \{y \mid (x, y) \in L\}$; (* denoted as $\{y_0, y_1, ...\}$ *)
(* all y_i are in the neighbourhood *)
1.3. **for** $i := 0$ **to** $|Y| - 1$
1.3.1. **do** *place* P *on* y_i;

[4]Each subgraph (= set of dependent tasks) is less or equal to a transaction.

[5]Using routing there is no higher chance to find a feasible assignment due to increased communication interference and overhead.

1.3.2. $\quad feasible := SchedulabilityTest();$
1.3.3. $\quad$ **if** $feasible$
1.3.3.1. $\quad$ **for all** *"not yet assigned"* $Q \in V_{CP}$ **with** $(Q, P) \in E_{CP}$
1.3.3.1.1. $\quad feasible := try(Q, y_i);$
1.3.4. $\quad$ **if not** $feasible$
1.3.4.1. $\quad$ *remove* P *from* y_i*;*
1.3.5. $\quad$ **done**
1.4. $\quad$ **return** $feasible;$
1.5. $\quad$ **done**
(* main procedure: *)
2. $\quad$ **procedure** $CP\_Mapping()$
2.1. $\quad$ **do for all** G^i_{CP} **for** $j := 0$ **to** $|X| - 1$
2.1.1. $\quad try(P^i_0, x_j);$
2.2. $\quad$ **done**

The algorithm starts with placing the most critical path (CP-vertex) of each CP-subgraph (see line 2.2.1.) and then recursively iterates through all adjacent CP-vertices (see line 1.3.3.1.1.). In order not to violate the neighbourhood relation, each adjacent vertex is either placed on the same processor or on an adjacent processor (see line 1.2.). If there is a deadline violation (see lines 1.3.3. and 1.3.4.), the algorithm backtracks to the next possible entrance point (see lines 1.3.4.1. and 1.3.1.).

During test-case evaluation it turned out to be reasonable to always try free processors first (see line 2.1.), before backtracking to already occupied processors. Similarly, free processors (or processors hosting tasks of the same transaction) should be preferred to occupied processors in the neighbourhood (see line 1.3.). In general this will protect the schedulability analysis as much as possible from dealing with long LCMs.

4.3 Discussion of the Slack Method

The main weakness of the Slack Method is that it sometimes fails in finding any feasible assignment, although there is one, due to the fact that the graph reductions lead to loss of information, or due to communication interference which is intentionally ignored. This could be improved by using a more sophisticated method than Steepest Descent in Algorithm 12, or by using a better size function (see Definition 11). However, the experimental results in Section 5.3 show that for most of the considered larger applications a solution was found by the simple approach.

The use of an optimal algorithm for the CP-Mapping might in some cases lead to large computation times. Although this was not detected during test-case evaluation, the backtracking policy of Algorithm 14 could be replaced by a non-guided search or list-processing approach, in order to improve performance.

The CP-Mapping may also suffer from large computation times when dealing with multi-period systems, since then the involved schedulability analysis has to cope with long LCMs. For example, the application *tind* in (Section 5) consisting of 43 tasks with 4 different period lengths is blown up to $36,520$ task activations inside the LCM. However, when dealing with processor networks which have known properties, a special-purpose solution should be used in order to improve run-time performance by exploiting these properties. This can be done, for instance, in the case of point-to-point grid networks where the regular shape of this kind of processor network can be exploited to avoid that tasks with different periods are assigned to the same processor. This pointed out in more detail in [4].

For incorporation of replicated tasks, and tasks that are preallocated please refer to [4].

5 Test-Case Evaluation

This section presents an evaluation of a number of application examples. The properties of the considered controller applications are listed in Tab. 1.

id	$\|V\|$	$\|E\|$	#transactions	$\|V_{CP}\|$	$\|E_{CP}\|$
bsh21	48	66	1	3	1
bsh30	48	66	1	2	0
bsp11	738	2016	1	11	40
bsp20	738	2016	1	6	22
bus15	1710	2663	1	10	29
bus7	1710	2663	1	27	200
hyp	148	154	5	16	6
loadanr7	73	91	1	2	2
meg	100	88	4	13	4
pend3d10	346	813	1	5	11
pend3d9	346	813	1	5	9
tind	43	36	4	6	0
tinlarge	499	410	4	55	36

Table 1: Application examples

The examples[6] are mostly mechatronic applications: The examples *hyp*, and *meg* are constructed, example *tind* is from [49], and example *tinlarge* is constructed from a number of copies of *tind*. Examples with identical prefixes in their names are equal, apart from different periods, where smaller numbers indicate smaller periods and deadlines. In the following, the results of three different investigations are discussed. First, comparative results of the different purely non-guided search methods are presented in Section 5.1. Second, change of run-time behaviour and objective function values are considered, in a situation when each non-guided search is pre-processed by List Processing (see Section 5.2). And finally, the same is done with the Slack Method (see Section 5.3). Optimal algorithms are not considered due to their complexity as explained in Section 3.1.

All examples were assigned to grid networks consisting of both *schedule-driven (SD)* processors (i.e. running cyclic executives) as well as OS-driven (OD) processors (using an online deadline-monotonic dispatcher)[7]. In the corresponding objective function (to be minimised) the two processor types are incorporated as follows: For each SD processor the cost is 100%, since it is assumed that available slack cannot be used, and hence the whole processor is used. For each OD processor the cost is the sum over the task loads hosted there, including OS overhead.

[6]The examples can be accessed by `http://www.c-lab.de/~peter/benchmarks.html`.

[7]For a discussion of these two processor types refer to [4, 31].

Similarly, the cost for communication is the sum over the loads of links. This is expressed by the following definition:

DEFINITION 15 (OBJECTIVE FUNCTION): *Let SD be the set of SD processors, OD be the set of OD processors, and L be the set of communication links. The* objective function*, F, is defined as:*

$$F = 100 * |SD| + \sum_{P \in OD} load_P + \sum_{l \in L} load_l$$

The load of a processor P, $load_P$, is the sum of the loads of all tasks hosted there plus OS overhead. The load of a task p is $C_p/T_p * 100$ (see Section 2). Similarly, the load of a link l, $load_l$, is the sum of the communication load of all tasks using this link. The communication load of a task can also be derived from its communicational effort and its period length.

5.1 Evaluation of different non-guided search methods

Tab. 2 shows the results of the application of the different purely non-guided search methods with regard to both objective function values (upper rows) as well as CPU times (lower rows) on a Sparc Ultra Workstation.

It was very difficult (if not impossible) to find fair termination conditions for all methods. In general it is not easy to compare different non-guided search methods, since this would be more like a "beauty contest" [36], where the way decisions are made is not very well defined. There are many reasons why a certain technique could outperform another, such as the termination criteria, the cooling function of Simulated Annealing, the mutation probability of the Genetic Algorithm, the size of the tabu list of Tabu Search, etc.. However, the results in Tab. 2 show that all non-guided search methods have their strengths and weaknesses depending on the application. During the experiments represented by Tab. 2 Simulated Annealing reported the largest number of feasible assignments. Considering only the three examples (marked "*"), where all methods found a feasible assignment, Steepest Descent gave the best results (see row *"average *"* in Tab. 2), even though it needed more time than other methods (see row *"time *"* in Tab. 2). The Genetic Algorithm was unable to deal with *bus15* and *bus7* due to memory limitations.

The unusual solution space was a common problem for all these methods. Although they sometimes quickly report very good results (Steepest Descent and *tind*), they also sometimes suffer from poor assignments (Tabu Search and *tinlarge*), very long computing times (Steepest Descent and *bus15*), and inability to find feasible assignments (only 3 to 7 of 13).

5.2 Evaluation of List Processing

Tab. 3 shows the impact of List Processing[8] on the non-guided search methods, i.e. List Processing was used as a preprocessing step for different non-guided search methods. The improvement column "+/−" in Tab. 3 represents the average improvement of the combined method over the pure method, regarding only those cases where both methods could find a feasible assignment.

[8]For priority ordering, the task deadlines were used. For more details on the implementation of List Processing, please refer to [4].

name	SD	TS	SA	TA	GA
bsh21	–	–	3150	3150	–
	0.8	11.0	3.9	3.9	36.9
bsh30	–	–	2084	2077	–
	1.7	2.5	2.2	2.4	37.6
bsp11	–	–	–	–	–
	40:28.4	3:20.0	9:18.7	17:17.2	34:58.0
bsp20	–	–	2161	–	–
	4:42.8	2:47.0	1:19:57.0	36:10.5	34:33.2
bus15	–	–	–	–	–
	8:55:47.2	9:01.2	23:11.4	33:55.7	–.-
bus7	–	–	–	–	–
	3:38:55.2	9:00.3	23:20.6	28:37.8	–.-
hyp	–	–	–	–	–
	15.1	4.4	37.7	47.1	1:07.8
loadanr7	–	–	2124	2124	–
	2.8	6.5	11.8	11.6	42.9
meg	8996	22511	11024	10944	11352
*	12.3	3.9	31.3	40.8	1:46.7
pend3d10	–	–	–	–	–
	40.5	1:45.9	2:02.5	3:59.0	4:34.9
pend3d9	–	–	–	–	–
	40.4	21:17.9	2:05.5	5:15.1	4:35.0
tind	4101	9511	5341	5341	5159
*	1.6	0.6	5.3	6.3	20.3
tinlarge	48260	128511	60697	87977	62257
*	14:21.8	1:17.9	7:50.2	2:19.7	29:59.8
# feasible	3	3	7	6	3
average *	20452.3	53511.0	25687.3	34754.0	26256.0
time *	14:35.7	1:22.4	8:26.8	3:06.8	32:06.8

Table 2: Non-guided search results: Upper row = objective function value, lower row = CPU times. SD = Steepest Descent, TS = Tabu Search, SA = Simulated Annealing, TA = Threshold Accepting, GA = Genetic Algorithm, * = feasible

Just as a non-guided search, List Processing often fails in finding any feasible assignment. However, List Processing could detect a number of feasible assignments which were not found by the non-guided search methods (bold face numbers in Tab. 3). This increased the average number of feasible assignments (+95%). Unfortunately, the objective function values of the combined algorithms are still worse than those of the pure methods (see column *Improvement* in Tab. 3). Though the CPU times of List Processing are negligible compared to those of the non-guided search methods, the overall values of the combined approach are worse in some cases (see column *Improvement* in Tab. 3), due to the poor objective function values of List Processing.

Comparing the different non-guided search techniques, this time Threshold Accepting gave the best results with regard to both, the largest number of feasible assignments as well as the best quality among the six applications (marked "*"), where all methods succeeded in finding a feasible assignment (see row *"average *"* in Tab. 3), even though it needed more time than other methods (see row *"time *"* in Tab. 3).

name	LP	LP & SD	LP & TS	LP & SA	LP & TA	LP & GA	+/–
bsh21	–	–	–	3150	3150	–	0%
	< 0.1	0.8	11.0	3.9	3.9	36.9	0%
bsh30	6147	**4133**	**5147**	4488	3829	**6147**	-100%
*	< 0.1	1.4	0.7	2.6	4.7	5.6	-59%
bsp11	–	–	–	–	–	–	–
	1.2	40:29.6	3:21.2	9:19.9	17:18.4	34:59.2	–
bsp20	5611	**4605**	**4611**	3134	**2130**	**5093**	-45%
*	3.2	56:36.4	10:07.1	2:27:02.1	2:10:51.4	1:12.5	-84%
bus15	7614	**5015**	**6623**	**6153**	**6554**	–	–
	15.5	2:15:34.4	31:27.9	5:22:34.9	51:22.7	–.-	–
bus7	–	–	–	–	–	–	–
	6.4	3:39:01.6	9:06.7	23:27.0	28:44.2	–.-	–
hyp	–	–	–	–	–	–	–
	< 0.1	15.1	4.4	37.7	47.1	1:07.8	–
loadanr7	–	–	–	2124	2124	–	0%
	< 0.1	2.8	6.5	11.8	11.6	42.9	0%
meg	19110	17631	18110	18978	19080	11271	-31%
*	0.1	10.1	4.0	13.3	9.4	53.0	54%
pend3d10	9016	**6191**	**8928**	**4628**	**4860**	**9016**	–
*	0.9	37:40.4	3:04.0	10:12.7	3:46.8	34.6	–
pend3d9	9472	**7024**	**9376**	**6336**	**5216**	**9472**	–
*	0.9	52:50.3	3:03.4	4:19.5	4:12.0	36.0	–
tind	13082	4141	12082	5034	4988	5219	-7%
*	< 0.1	3.8	0.6	2.9	3.9	13.5	28%
tinlarge	–	48260	128511	60697	87977	62322	0%
	0.4	14:22.2	1:28.3	7:50.6	2:20.1	30:00.2	0%
# feasible	7	8	8	10	10	7	95%
average *	10406.3	7287.5	9709.0	7099.6	6683.8	7703.0	–
time *	5.1	2:27:22.4	16:19.8	2:41:53.1	2:19:08.2	33:35.2	–

Table 3: List Processing with non-guided search result: Upper row = objective function value, lower row = CPU times. LP = List Processing, SD = Steepest Descent , TS = Tabu Search, SA = Simulated Annealing, TA = Threshold Accepting, GA = Genetic Algorithm, * = feasible, +/- = improvement

5.3 Evaluation of the Slack Method

Tab. 4 shows the impact of the Slack Method on the non-guided search techniques. As in the List-Processing case the Slack Method served as a preprocessing step for different non-guided search techniques. The improvement columns are derived in the same manner as in Section 5.2.

As the data in Tab. 1 indicates, the problem size (the original graph versus the CP-graph) is very much reduced using the Slack Method. Hence, many more feasible assignments could be reported: +164% compared to purely non-guided search (bold face numbers), and +35% compared to List Processing with non-guided search, which is shown in Tab. 4. Only in one case (*pend3d9*) the Slack Method is unable to construct a feasible assignment. Furthermore, in most cases a combination of the Slack Method and non-guided search is better than purely non-guided search or List Processing with non-guided search with regard to both objective function values as well as CPU times (see column *Improvement* in Tab. 4).

Comparing the different non-guided search techniques, this time Steepest Descent gave the

name	SM	SM&SD	SM&TS	SM&SA	SM&TA	SM&GA	Improvement	
							NGS	LP
bsh21	3120	**3120**	**3120**	3120	3120	**3120**	1%	1%
*	< 0.1	1.0	1.1	1.5	1.5	2.9	57%	57%
bsh30	2063	**2063**	**2063**	2063	2063	**2063**	1%	57%
*	< 0.1	1.1	1.1	1.1	1.1	2.1	52%	56%
bsp11	4398	**4384**	**4384**	**4030**	**4378**	**4398**	–	–
*	1:00.2	1:04:59.9	41:33.2	56:27.4	27:44.1	2:54.6	–	–
bsp20	2171	**2171**	**2171**	2129	**2171**	**2171**	1%	45%
*	17.9	5:56.2	5:57.4	13:31.7	3:39.1	1:27.0	89%	91%
bus15	4670	**3667**	**3670**	**3432**	**3667**	–	–	41%
	2:08.5	1:07:10.3	44:28.4	6:22:08.1	42:04.6	–.-	–	1%
bus7	12805	**12703**	**12790**	**11824**	**12742**	–	–	–
	4:15.9	5:42:39.1	51:19.3	6:14:08.5	2:57:54.7	–.-	–	–
hyp	15227	**12139**	**14427**	**12636**	**12499**	**15227**	–	–
*	0.4	10.2	7.5	18.1	1:15.3	13.7	–	–
loadanr7	2155	**2124**	**2124**	2124	2124	**2155**	0%	0%
*	0.1	5.9	3.6	1.9	2.0	2.7	83%	83%
meg	12227	9317	11427	9794	9794	12227	19%	51%
*	0.2	4.5	3.0	10.7	11.3	18.3	75%	47%
pend3d10	6004	**5564**	**5938**	**5300**	**5454**	**6004**	–	16%
*	2.7	29:04.3	2:31.7	5:31.1	4:38.4	25.7	–	24%
pend3d9	–	–	–	–	–	–	–	–
	1.6	1.7	1.7	1.7	1.7	1.7	–	–
tind	6027	4225	5127	4554	4554	6027	16%	22%
*	0.1	0.7	0.6	2.0	2.0	5.5	68%	56%
tinlarge	56451	51961	56151	52603	55069	56451	30%	30%
*	7.0	1:52.5	1:08.3	2:25.1	56.5	58.3	87%	87%
# feasible	12	12	12	12	12	10	164%	35%
avg *	10984.3	9706.8	10693.2	9835.3	10122.6	10984.3	–	
time *	1:28.6	1:42:16.3	51:27.5	1:18:30.6	38:31.3	6:30.8	–	

Table 4: Slack Method with non-guided search results: Upper row = objective function value, lower row = CPU times. SM = Slack Method, SD = Steepest Descent, TS = Tabu Search, SA = Simulated Annealing, TA = Threshold Accepting, GA = Genetic Algorithm, * = feasible

best results with regard to the ten applications (marked "*"), where all methods succeeded in finding a feasible assignment (see row *"average *"* in Tab. 4), even though it needed more time than other methods (see row *"time *"* in Tab. 4). The Genetic Algorithm failed in all examples in improving the initial assignment from the Slack Method.

6 Conclusion

This article has introduced the new constructive heuristic *Slack Method* for static allocation of communicating periodic hard real-time tasks. As shown by the test-case evaluation, the Slack Method is superior to list-processing-based approaches with regard to both finding more feasible assignments as well as finding assignments with better objective function values.

As shown by the discussion of related work and by the experimental evaluation, all investigated non-guided search methods mainly suffer from the same disadvantages: high sensitivity to variations of parameters, long processing times, and problems of finding feasible assignments for large application examples. These problems can be overcome by combining the Slack Method with non-guided search, which is the main practical result of this article. As emphasised by the experimental evaluation, a combined heuristic delivers good values within short response times to allow practical use even for large applications.

Since there is no method that always finds a sufficiently good assignment for the hard real-time allocation problem, we encourage the use of an *allocation framework*, where the user is automatically provided with the appropriate algorithm for his specific application, or at least has the possibility to test different algorithms in the context of a specific application domain (as in [27]). Future work should also include research with regard to systems consisting of non-identical processors, and heterogeneous processor networks.

The Slack Method will be used as a preprocessing step for non-guided search within the software architecture *CHaRy* (**C**-LAB **Ha**rd **R**eal-Time S**y**stem) [5]. The goal of CHaRy is to automatically derive distributed hard real-time implementations from higher-lever descriptions, including timing analysis, allocation and schedulability analysis.

CHaRy was originally developed for grid architectures. Currently CHaRy is extended towards bus interconnected distributed target systems, such as the *Basement* system [22], or the CAN-test-bed presented in [23].

Acknowledgments

The authors would like to thank Mark Nicholson (University of York) for his help on non-guided search techniques, and Prof. P.C.P. Bhatt (Paderborn University), Mikael Sjödin (Uppsala University) and Jan Jonsson (Halmstad University) for helpful comments on an earlier version of this article.

References

[1] J. M. Adán, M. F. Magalhães, and K. Ramamritham. *Meeting Hard Real-Time Constraints Using a Client-Server Model of Interaction.* In *Proceedings of the 7th Euromicro Workshop on Real-time Systems*, pages 286–293, 1995.

[2] Peter Altenbernd. *Deadline-Monotonic Software Scheduling for the Co-Synthesis of Parallel Hard Real-Time Systems.* In *Proceedings of the European Design and Test Conference*, pages 190–195, 1995.

[3] Peter Altenbernd. *On the False Path Problem in Hard Real-Time Programs.* In *Proceedings of the 8th Euromicro Workshop on Real-time Systems*, pages 102–107, 1996.

[4] Peter Altenbernd. *Timing Analysis, Scheduling, and Allocation of Periodic Hard Real-Time Tasks.* Ph.D. Thesis, Computer Science Department, Paderborn University, Germany, 1996.

[5] Peter Altenbernd. *CHaRy: The C-LAB Hard Real-Time System to Support Mechatronical Design.* In *Proceedings of the International Conference and Workshop on Engineering of Computer Based Systems ECBS'97*, pages 271–278, 1997.

[6] L. Baccouche. *Efficient Static Allocation of Real-Time Tasks Using Genetic Algorithms.* Internal Report, Imag Institute, Laboratoire de génie informatique, 1995.

[7] J. Błażewicz and K. Ecker. *Multiprocessor Task Scheduling with Resource Requirements. Journal of Real-Time Systems*, 6(1), January 1994.

[8] C. Cardeira and Z. Mammeri. *Neural Network versus Max-Flow Algorithms for Multiprocessor Real-Time Scheduling.* In *Proceedings of the 8th Euromicro Workshop on Real-time Systems*, pages 175–180, 1996.

[9] R. Chapman. *Static Timing Analysis and Program Proof.* Ph.D. Thesis, Computer Science Department, University of York, 1995.

[10] B.-C. Cheng, A. D. Stoyenko, and T. J. Marlowe. *Least-Space-Time-First Scheduling Algorithm: A Policy for Complex Real-Time Tasks in Multiple Processor Systems.* In *Proceedings of the WRTP'94*, 1994.

[11] T. C. K. Chou and J. A. Abraham. *Load Balancing in Distributed Systems. IEEE Transactions on Software Engineering*, 8(4):401–412, July 1982.

[12] M. Coli and P. Palazzari. *A New Method for Optimisation of Allocation and Scheduling in Real-Time Applications.* In *Proceedings of the 7th Euromicro Workshop on Real-time Systems*, pages 262–269, 1995.

[13] M. DiNatale and J. A. Stankovic. *Dynamic End-to-End Guarantees in Distributed Real-Time Systems.* In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 216–227, 1994.

[14] M. DiNatale and J. A. Stankovic. *Applicability of Simulated Annealing Methods to Real-Time Scheduling and Jitter Control.* In *Proceedings of the IEEE Real-Time Systems Symposium*, 1995.

[15] G. Dück and T. Scheuer. *Threshold Accepting: A General Purpose Optimisation Algorithm Appearing Superior to Simulated Annealing. Journal of Computational Physics*, pages 161–175, 1990.

[16] K. Efe. *Heuristic Models of Task Assignment Scheduling in Distributed Systems. IEEE Computer*, 15(6):50–56, June 1982.

[17] L. Ford and D. Fulkerson. Flows in Networks. Princeton University Press, 1962.

[18] F. Glover, E. Taillard, and D. deWarra. *A User's Guide to Tabu Search. Annals of OR*, 7(x):3–28, 1993.

[19] D. E. Goldberg. Genetic Algorithms in Search, Optimisation, and Machine Learning. Addison-Wesley Pub. Company Inc., 1989.

[20] G. Greenwood, C. Lang, and S. Hurley. *An Evolutionary Strategy for Scheduling Periodic Tasks in Real-Time Systems.* In *Applied Decision Technologies*, pages 171–188, 1995.

[21] J. Gustafsson and A. Ermedahl. *Deriving Annotations for Tight Calculation of Execution Time.* In *Proceedings of the Euro-Par'97 Conference*, 1997.

[22] H. Hansson, H. Lawson, M. Strömberg, and S. Larsson. *BASEMENT: a Distributed Real-Time Architecture for Vehicle Applications. Real-Time Systems*, 11(3):223–244, November 1996.

[23] H. Hansson and M. Sjödin. *CAN-based Real-Time Lab Environment. CAN Newsletter*, pages 48–49, September 1997.

[24] M. G. Harmon, T. P. Baker, and D. B. Whalley. *A Retargetable Technique for Predicting Execution Time of Code Segments. Journal of Real-Time Systems*, 7(2):159–182, 1994.

[25] R. B. Hitchcock. *Timing Verification and the Timing Analysis Program.* In *Proceedings of the 19th Design Automation Conference*, pages 594–603, 1982.

[26] C.-J. Hou and K. G. Shin. *Allocation of Periodic Task Modules with Precedence and Deadline Constraints in Distributed Real-Time Systems.* In *Proceedings of 13th Real-Time Systems Symposium*, 1992.

[27] J. Jonsson. *The Impact of Application and Architecture Properties on Real-Time Multiprocessor Scheduling.* Ph.D. Thesis, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1997.

[28] J. Jonsson, A. Olsson, and J. Vasell. *Predicting Real-Time Behaviour for Data-Flow Computations.* In *Proceedings of the 7th Euromicro Workshop on Real-time Systems*, pages 270–275, 1995.

[29] S. J. Kim and J. C. Browne. *A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures.* In *Proceedings of the International Conference on Parallel Processing*, 1988.

[30] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. *Optimisation by Simulated Annealing. Science*, 220(4589):671–680, 1983.

[31] H. Kopetz. *Event-triggered versus Time-triggerd Real-time Systems.* In *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*, pages 87–101, 1991.

[32] E. L. Lawler. Recent Results in the Theory of Machine Scheduling, pages 202–233. Springer Verlag, 1983.

[33] J. Y. T. Leung and J. Whitehead. *On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks. Performance Evaluation*, 2(4):237–250, December 1982.

[34] R. Naumann and C. Homburg. *LoDiT - Automatisches Partitionieren von mechatronischen Systemen für die verteilte Simulation.* In *10. Symposium der Simulationstechnik*, 1996.

[35] M. Nicholson. *Allocating and Scheduling Hard Real-Time Tasks on a Point-to-Point Distributed System.* In *Proceedings of Workshop on Parallel and Distributed Real-Time Systems*, 1993.

[36] M. Nicholson. *Optimisation Searches Inspired by Nature.* Qualifying Dissertation, Computer Science Department, University of York, 1995.

[37] M. A. Palis, J.-C. Liou, and D. S. L. Wei. *Task Clustering and Scheduling for Distributed Memory Parallel Architectures. IEEE Transactions on Parallel and Distributed Systems*, 7(1):46–55, January 1996.

[38] D.-T. Peng and K. G. Shin. *Static Allocation of Periodic Tasks with Precedence Constraints in Distributed Real-Time Systems.* In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 190–198, 1989.

[39] S.C.S. Porto and C.R. Celso. *A Tabu Search Approach to Task Scheduling on Heterogeneous Processors under Precedence Constraints.* Report PUCRioInf-MCC03/93, Pontificia Universidade Católica do Rio de Janeiro, 1993.

[40] P. Puschner and C. Koza. *Calculating the Maximum Execution Time of Real-Time Programs. Journal of Real-Time Systems*, 1(2):159–176, September 1989.

[41] K. Ramamritham. *Allocation and Scheduling of Complex Periodic Tasks.* In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 108–115, 1990.

[42] S. Rönngren and B. A. Shirazi. *Static Multiprocessor Scheduling of Periodic Real-Time Tasks with Precedence Constraints and Communication Costs.* In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, pages 143–152, 1995.

[43] F. E. Sandnes. *A Hybrid Genetic Algorithm Applied to Automatic Parallel Controller Code Generation.* In *Proceedings of the 8th Euromicro Workshop on Real-time Systems*, pages 70–75, 1996.

[44] J. Santos, E. Ferro, J. Orozco, and R. Cayssials. *A Heuristic Approach to the Multitask-Multiprocessor Assignment Problem Using the Empty-Slots Method and Rate Monotonic Scheduling. Journal of Real-Time Systems*, 13(2):167–199, September 1997.

[45] V. Sarkar. Partitioning and Scheduling Programs for Execution on Multiprocessors. The MIT Press, Cambridge, Massachusetts, 1989.

[46] G. C. Sih and E. A. Lee. *Declustering: A new Multiprocessor Scheduling Technique. IEEE Transactions on Parallel and Distributed Systems*, 4(6):625–637, June 1993.

[47] Friedhelm Stappert. *Predicting Pipelining and Caching Behaviour of Hard Real-Time Programs.* In *Proceedings of the 9th Euromicro Workshop on Real-time Systems*, 1997.

[48] K. Tindell. *Holistic Schedulability Analysis for Distributed Hard Real-Time Systems.* Report YCS197, Department of Computer Science, University of York, 1993.

[49] K. Tindell, A. Burns, and A. Wellings. *Allocating Real-Time Tasks (An NP-Hard Problem made Easy). Journal of Real-Time Systems*, 4(2):145–165, 1992.

[50] J. Xu. *Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations. IEEE Transactions on Software Engineering*, 19(2):139–154, February 1993.