Solving Embedded System Scheduling Problems using Constraint Programming

Cecilia Ekelin and Jan Jonsson

Department of Computer Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
{*cekelin,janjo*}*@ce.chalmers.se*

Abstract

Static scheduling of tasks in embedded distributed real-time systems often implies a tediuos iterative design process. The reason for this is the lack of flexibility and expressive power in existing scheduling frameworks, which makes it difficult to both model the system accurately and provide correct optimization guidelines in a first attempt. The most detrimental effect this has on the scheduling process is that the real-time tasks are over-constrained due to a careless use of rules-of-thumb.

We have developed a scheduling framework based on constraint programming which attempts to tackle these deficiencies. First, the framework allows for expressing the scheduling problem in terms much closer to the actual system requirements. This is because of the support provided by the framework for modeling constraints that are usually not handled by existing scheduling tools, for example, different types of task allocation constraints such as clustering and proximity to resources. A second strong feature of the framework is that it supports different strategies for single- and multi-objective optimization, something that is very important in the design of embedded real-time systems. An evaluation study encompassing a set of applications with typical embedded system constraints suggests that our approach does in fact provide the salient features necessary.

1 Introduction

Real-time system design is becoming increasingly dependent on flexible scheduling frameworks to cope with the size and complexity of contemporary applications. This is most apparent in the design of embedded distributed real-time systems where the scheduling framework must possess two salient features, namely *(a) support for a large variety of application constraints* to handle scheduling of distributed tasks and dedicated resources, and *(b) support for advanced optimization capabilities* to concurrently account for system requirements on predictability, reliability, and hardware performance, as well as various aspects of total system cost. As a basis for constructing a scheduling framework, real-time literature proposes several scheduling algorithms that present solutions to different aspects of the scheduling problem. Unfortunately, most existing scheduling algorithms only solve stand-alone problems (for example, focusing only on uniprocessor systems, one type of task model, and a single optimization criterion). This makes construction of a scheduling framework particularly difficult because there may exist a discrepancy between the theoretical scheduling problem and the practical aspects of the real-time system being designed, which gives rise to two major problems.

First, there is the problem of using constraints that are as close as possible to the original system requirements. Since many scheduling algorithms are defined with "hard-coded" assumptions regarding task and system models, it is easy to believe that it suffices to devise constraints that fit the scheduling algorithm. However, such an approach would modify the semantics of the original system requirements, and typically lead to the introduction of *artificial* constraints without any direct connection to the original scheduling problem [1, 2]. Such design approaches are very error-prone since the *ad hoc* constraint construction often leads to an over-constrained and infeasible system. This turns scheduling into a tedious design process since the designer must suggest suitable changes in the specification, implementation and/or system models.

Hence, instead of adapting the problem to the scheduling algorithm, the scheduling algorithm should be adapted to the problem in the sense that it should be capable of accounting for as many *natural* and *implementation-based* constraints as possible.

Second, even if a feasible schedule is found by the scheduling framework, the suggested solution may still be considered invalid by the system designer since requirements like cost and hardware resources usually cannot be accounted for in most scheduling algorithms. This makes it necessary for the designer to manually verify a schedule with respect to these requirements, and suggest suitable design changes. Hence, a scheduling framework with a notion of multi-objective optimization would shorten the process cycle. Moreover, it would increase the ability for real-time system developers to provide cost-effective solutions. This is particularly useful as the trend in embedded real-time system design is moving towards open systems [3]. Apart from making the scheduling process more effective, we also believe that by using constraints that are semantically closer to the requirements, the scheduling search space can be reduced. The rationale for this is that, when constraints are more close to the original requirements, the scheduling algorithm can operate in a more intelligent way since it has more knowledge of the problem (see, for example, recent results for branch-and-bound algorithms [4, 5]).

In this paper, we propose a scheduling framework that possesses all the salient features discussed above. While many other proposed scheduling approaches for distributed real-time systems (for example, simulated annealing or branch-and-bound) have similar features, it is well-known that they require a large amount of fine-tuning to perform well. In contrast, our framework is based on the concept of *constraint programming*, which means that constraints can be introduced in the framework in a flexible and natural way, and that the optimization features do not require tedious manual interaction.

Organization of this paper: This paper addresses how to build a scheduling framework based on constraint programming. The rest of our presentation is organized as follows. In Section 2, we describe typical requirements for embedded distributed real-time systems, and recollect methods for automated constraint derivation and scheduling of distributed systems. In Section 3, we summarize a set of constraints typically found in embedded distributed real-time system design. We present the underlying practical motivation for these constraints, and identify problems in using the constraints with existing scheduling algorithms. In Section 4, we present different generic strategies for single- and multi-objective optimization. We demonstrate how these techniques can be applied to scheduling of embedded distributed real-time systems, and how different strategies for constraining the scheduling problem can lead to improved scheduling performance. In Section 5, we introduce the constraint-programming paradigm and show how it can be used for building a scheduling framework that supports all identified constraints (from Section 3) as well as single- and multi-objective optimization. In Section 6, we demonstrate the practical usefulness of the framework by performing several case studies and evaluating different aspects of the framework's scheduling performance. Finally, in Section 7, we discuss our results and suggest future directions, while our findings are summarized in Section 8.

2 Background

The design of real-time systems encompasses three important phases. From the specification of the system, it must be possible to identify the *requirements* on the system in terms of functionality, performance, and cost. Based on the system specification, an implementation phase then follows wherein the designer makes a choice of programming environment, run-time system and hardware architecture. From the choice of implementation, it is possible to identify the concrete tasks that the application is required to perform, and the resource that are available for its execution. Finally, the designer schedules the application tasks on the available resources. This is done by using a scheduling framework that automates the scheduling and allocation of the task, and also verifies that all requirements of the application are met. In order to do this, however, it is necessary to use models of the application tasks and the system architecture.

In the scheduling phase, it is possible to identify two potential caveats in the modeling of the tasks and the architecture. First, it is necessary (for practical reasons) to abstract away some implementation details and replace them with quantitative measures; for example, program code stretches and processor hardware mechanisms are typically analyzed separately in order to derive a worst-case execution time of

each task. These abstractions suffer from a potential risk of being overly pessimistic since embedded real-time systems often contain strict timing constraints. Second, in order to convey the system requirements to the scheduling framework, it is necessary to introduce a set of *constraints* that models system requirements into a manageable form for the scheduling algorithm. Since the scheduling framework is used to validate that the system requirements are actually met by the implementation, the methodology used for constructing constraints must be cognizant of the practical application domain as well as the theoretical scheduling domain. In our methodology, the constraints must *(a) reflect the intended behaviour of the system*, *(b) be derived in natural way without overconstraining the system*, and *(c) be fully supported by the scheduling framework*. In the following subsections, we describe the state-of-the art in practical real-time system design with respect to these three features.

2.1 Requirements

The system requirements for an embedded distributed real-time system can be divided into different groups which each impose a number of constraints on the scheduling problem.

The *functional* behavior of a real-time system is determined by the tasks and resources that constitute the system. Typical functional behavior requirements are those that control task execution order or task allocation, that is, *how* a task should execute. A real-time system also have *temporal* behavior requirements in addition to the functional ones. The temporal behavior of a task depends mainly on the environment (sensors, actuators or other tasks) that the task interacts with, that is, *when* a task should execute. These requirements directly affect the modeling of the application tasks and consequently the construction of the scheduling constraints.

Most real-time systems are also safety-critical in the sense that a failure in a processor could be catastrophic. To avoid such failures the embedded system must often also be *fault tolerant*. Fault tolerance is achieved by introducing redundancy in the system, for example, using additional processors and/or copies of tasks. Hence, it is clear that the requirement of making a system fault tolerant affects the implementation of the system, and consequently also the construction of scheduling constraints.

Apart from mere software requirements, it is also a practical consideration that the development of embedded real-time systems is made *cost-effective* so as to allow for mass-production of the system. That is why development using off-the-shelf hardware components has become a viable alternative in modern designs. Other practical aspects of embedded system design encompass the introduction of *weight* and *power-consumption* requirements. This means that cost, performance and various physical characteristics of the hardware components (processors, memory and busses) need to be conveyed to the constraint construction process.

2.2 Constraint derivation

Recent analyses [1, 2] have indicated that *most constraints are artifacts of the design*. This does not come as a surprise since they are a necessary consequence of the models used for tasks, run-time system and architecture. However, since the modeling process often has to be done manually, constraints should be constructed according to suitable guidelines or otherwise serious drawbacks will result. For example, an *ad hoc* constraint derivation is likely to lead to an over-constrained system which may prevent the scheduling algorithm from finding the best (or any) solution. On the other hand, if the system is too loosely constrained, finding the optimal solution might by too computationally intractable in practice. Furthermore, the semantics of the original requirements may be lost in the process, which from a practical point of view means that it will be hard to detect performance bottlenecks in the case of a failed scheduling attempt [6].

To overcome these problems, many researchers have recently proposed automated methods for constraint derivation. An example (taken from [1]) of a design flow for constraint derivation is illustrated in Figure 1. As indicated in the figure, there are basically two types of constraint derivation that can be performed: (a) derivation of system-level end-to-end constraints from performance requirements, and (b) derivation of task-level constraints from the end-to-end constraints.

The first type of constraint derivation means translating performance requirements, such as maximum steady-state error or maximum transient overshoot, into system-level end-to-end constraints that describe,

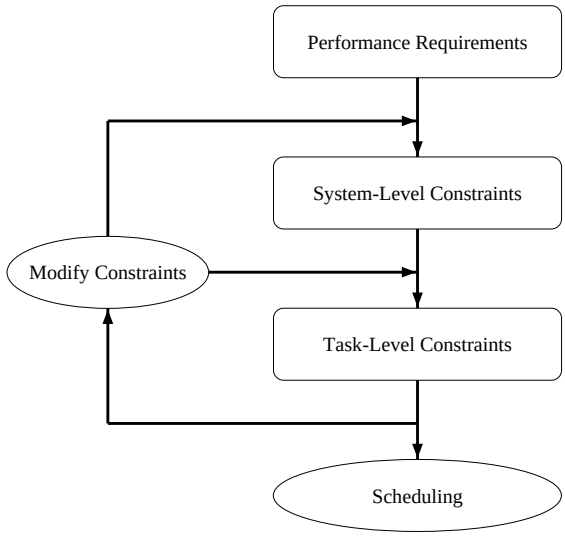

Figure 1: Constraint-derivation flow.

for example, maximum sensor-to-actuator latency, minimum sampling periods, or maximum output jitter. To that end, several techniques for system-level constraint derivation have been proposed in the context of control systems (see, for example, [7, 8, 9]).

The second type of constraint derivation means translating the end-to-end constraints into task-level timing and execution constraints. One of the most refined techniques for this type of constraint derivation is the period calibration method (PCM) proposed by Gerber, Hong and Saksena [10, 11]. The PCM works in a two-stage fashion. First, task periods are derived from given end-to-end system constraints. Deadlines and release times are then assigned to individual tasks using the derived periods[1]. While several other techniques exist for deriving task-level deadlines and release times (see, for example, [13, 14, 15, 16, 17, 18]), few techniques actually exist for deriving other types of task-level constraints. However, Ramamritham [19] and Abdelzaher & Shin [20] propose techniques that generate task clustering recommendations based on heuristic analyses of inter-task communication needs and task periods, respectively.

All of these constraint derivation techniques assume a subsequent scheduling stage which means that constraint derivation is primarily performed with the objective of increasing the likelihood of succeeding with the scheduling attempt. In fact, most techniques are tailored for a specific scheduling policy in order to generate good results[2]. It should also be noted that, with few exceptions, all techniques for derivation of task-level *timing* constraints work under the assumption that tasks have been assigned to processors in advance, which clearly limits their applicability to larger distributed systems. As we will see later in this paper, the constraint-programming paradigm does not suffer from this limitation.

2.3 Scheduling approaches

Recall that a scheduling framework for embedded system design should include a scheduling algorithm that is capable of accounting for various types of constraints. It is also desired that the algorithm should find a "good" or even optimal schedule using advanced single- or multi-objective optimization strategies. Unfortunately, most scheduling algorithms proposed in the real-time research literature fail to satisfy both of these requirements. In fact, there are very few scheduling approaches that even claim to solve this complex problem. However, there are three scheduling approaches that do have the potential to provide the capabilities necessary.

Simulated annealing is a local search technique that has been applied to real-time allocation and scheduling by Tindell, Burns and Wellings [22]. The method attempts to minimize a so-called energy function which describes the quality of a schedule. The algorithm offers great flexibility but since the

[1] In a recent paper, Nilsson *et al.* [12] presented an application of PCM on an avionics control software using constraint programming.

[2] An integrated constraint-derivation and scheduling approach is proposed in the holistic approach by Tindell *et al.* [21].

energy function contains both the constraints and the scheduling objective it can be troublesome to balance these factors correctly. The stochastic nature of simulated annealing reduces the computational complexity since only portions of the search space are examined but this also means that there is no guarantee that the resulting schedule is optimal.

Branch-and-bound (B&B) systematically explores the search space in a tree-like fashion (branching). The search space is reduced by pruning branches in the tree that can not improve the solution found so far (bounding). For this method to be effective the search-tree should be subject to a lot of pruning. However, the computation of an accurate pruning rule is about as complex as solving the problem in the first place. The use of B&B in the context of scheduling for distributed real-time systems have been investigated by several researchers (see, for example, [23, 24, 25, 26, 4]).

Constraint programming is a technique which has been used with great success to solve scheduling problems within operations research and artifical intelligence. The scheduling problem is expressed as variables and constraints which are handled by a constraint solver. The method has been shown to be promising also for real-time scheduling by Schild and Würtz [27]. We extend this work by considering additional real-time constraints and also adding support for different scheduling objectives. This is discussed in more detail in Section 5.

3 Identification of Constraints

An important issue in the design of embedded real-time systems is what task-level constraints exist for that particular domain. In this section, we look closer at these constraints and attempt to justify the presence of each constraint construct by examining its origin. We also discuss how constraints relate to each other, and, based on this information, attempt to identify a minimal set of necessary constraints to be implemented in a scheduling framework.

The system model assumed in this paper has been chosen to reflect a typical embedded system. The hardware architecture consists of a number of *nodes* which are connected via a *bus*, and each node contains one or more *processors*. The application includes a set of *tasks* that execute on the processors and possibly communicate by message passing on the bus. Each node has a number of *resources* that can be used locally by tasks at that node, or globally by all tasks in the system.

3.1 System constraints

System constraints are imposed by the selected hardware architecture of the system. How the hardware is composed depends on functional or economical reasons, which means that the system configuration is typically fixed. The configuration includes information on the number of processors and other resources in the system (that is, a model) as well as their performance and capacities (that is, properties). Hence, the system configuration can be expressed using *model* and *property* constraints.

Processors have different speed which means that the worst-case *execution time* of a task depends on which processor it is scheduled to execute on. Hence, in a distributed system the actual execution time for a task has to be dynamically calculated during the scheduling. Each processor is also subject to a *context-switch cost* which usually is assumed to be small enough to be neglected or, in the case of non-preemptive scheduling, assumed to be included in the task's worst-case execution time. However, for preemptive scheduling, where it is not known beforehand how many times a task will be preempted, this approximation is not desirable. In this case, worst-case execution time and context-switch cost have to be handled separately during scheduling.

Resources other than processors have different limited *capacities* which restrict the number of tasks that can simultaneously access a resource. For example, a processor might have a RAM of 16 kbytes. Assume there are three tasks T_1, T_2 and T_3 which each use 4, 8 and 12 kbytes, respectively, during execution. Then T_1 and T_2 are allowed to execute concurrently as well as T_1 and T_3, whereas T_2 and T_3 are not. A special case is a critical section which can be modelled as a resource with capacity 1. Tasks which share a particular critical section are then defined to require usage of 1 entity from this resource during their execution.

The communication on the bus can be modeled using a *transmission delay* which reflects the time to transmit messages. The transmission delay $c_{transmit}$ is expressed as:

$$c_{transmit} = c_{speed} \cdot c_{size} + c_{overhead} + c_{delay}$$

where c_{speed} is the data rate of the bus, c_{size} the size of a message to transmit, and $c_{overhead}$ and c_{delay} are overheads pertaining to the message queuing/retrieval and bus scheduling strategy, respectively. There are a number of bus scheduling strategies which each gives different c_{delay} in the formula above. A *linear* model causes no delay since it is assumed that the message always can be delivered whenever requested because there is no contention from other messages. In a *contention-based* model (such as the one used by Mecel's BASEMENT architecture [28]) the bus is regarded as an additional processor and the messages as tasks be to scheduled on the bus. The delay then depends on what other messages that simultaneously contend for the bus. If the bus uses a *time-triggered* protocol, for example, DACAPO [29] or TTP [30], messages can only be sent in dedicated time slots. The delay then depends on how long time it is to the next available time slot. In a *priority-based* communication protocol, such as CAN, the delay is caused by higher-priority messages being sent [31]. It is clear from this discussion that the transmission delay for each message has to be dynamically calculated during the scheduling.

3.2 Task constraints

Task constraints are the restrictions that control the *timing* and *execution* behavior of the tasks and concern both the behavior of a single task (*intra-task*) as well as interaction between tasks (*inter-task*).

3.2.1 Intra-task timing constraints

In this group, we find constraints that restrict the time limits within which a single task should execute.

The *period* determines how frequently a task should execute and is linked to how accurate the system needs to be in the interaction with its environment. From a pure functionality point of view it is likely that the required period not is completely fixed, but rather is expressed as an interval (sometimes called a *separation* constraint). To guarantee the responsiveness required by the system, a task is also subject to a *deadline* that defines an upper limit on the finish time of the task. On the other hand, a task must not start too soon. For example, a task might require a sensor value which is not immediately available at the beginning of the task's period. Hence, the *release time* of the task also has to be constrained. Note that periods are related to both deadlines and release times in that the k^{th} invocation of a task must typically be completed before invocation $k+1$, thus imposing deadlines and release times for each invocation.

Besides release time and deadline, the start and finish times of a task can also be constrained by input or output *jitter* constraints. Input jitter is the difference between the release time and the actual start time of the task. Similarly, output jitter is the difference between the deadline and the actual finish time of the task. The main reason to limit the amount of jitter is when a task is required to execute at regular intervals, which is the case in control applications.

As mentioned earlier, many scheduling algorithms require that deadlines and release times are defined for each task in order to solve a certain scheduling problem. If only end-to-end deadlines are given for the application, these deadlines must be split into shorter deadlines which are assigned to each task. In a similar fashion, release time constraints are typically imposed as a way to obtain mutual exclusion between tasks that access the same resource. However, the introduction of such forced artificial task-level constraints may put an unnecessary strain on the task set, and hence affect schedulability. To avoid this, the semantics of mutual exclusion and end-to-end deadlines should be modeled using more flexible constraints, such as resource-usage and precedence constraints.

3.2.2 Inter-task timing constraints

This group includes constraints that express time limits within which two tasks should execute in relation to each other.

Interacting tasks are subject to three types of constraints that originate from the application requirements, namely *distance* constraints (due to transmission delay or input/output delays in a control system),

freshness constraints (due to aging of data in, for example, database applications) and *correlation* constraints (due to limits on the allowed time-skew in concurrent operations, for example fault-tolerance voting). There is also another type of inter-task timing constraint, but which originates from the implementation of the system, namely *harmonicity* constraints. These constraints are mainly imposed to simplify the implementation of communicating tasks. It is desired that the period of the receiver task is exactly divisible by the period of the sender task since this simplifies the procedure of identifying received messages. Depending on the run-time scheduler used, this constraint may or may not be needed. For example, in a priority-driven scheduler the harmonicity constraints are necessary since the order of execution for two tasks of different periodicity is not predictable. On the other hand, if a time-driven scheduler is used, it is possible to guarantee (using precedence constraints) that the order of execution between two such tasks is always the same.

3.2.3 Intra-task execution constraints

This group includes constraints that are local to a single task and determine on what processor and with what resources the task should execute.

If a task *uses a resource*, the set of nodes that the task can be allocated to is automatically restricted since the resource must be present at the node and have enough capacity. We distinguish between the cases where the resource is always required (*static*) and where the resource is required only during execution (*dynamic*). Examples of these cases are ROM and RAM, respectively. Tasks can also be directly allocated to a certain node using *locality* constraints which often are implementation recommendations made by the designer, and less frequently given in the specification. Another implementation issue is whether all invocations of a task have to execute on the same processor (*affinity*), which is most likely desired in a distributed system because the cost of migrating the task is too high. It also has to be decided whether tasks are allowed to interrupt each other (*preemption*). By allowing preemption it might be possible to find schedules for designs that are otherwise infeasible.

3.2.4 Inter-task execution constraints

This group includes constraints that determine in what order, on what processor, and with what resources two or more tasks should execute in relation to each other.

A specific function in the system is often modeled as a sequence of operating tasks. Hence, tasks should execute in a certain order which is typically expressed by *precedence* constraints. In case data should be exchanged between tasks, *communication* constraints are used to model a precedence constraint as well as an amount of data to communicate[3]. It is worth noting that distance constraints can actually be used to model precedence (assume a "distance" equal to 0) as well as communication constraints (assume a "distance" equal to a fixed transmission delay).

In some distributed systems, *clustering* is used to assign communicating tasks onto the same node in order to limit the cost for the communication network or to increase the schedulability. Another rationale for clustering is when the system has different modes for which different task sets are active. Now, if only a few tasks are active in one mode, it could be a good idea to save power by turning off all nodes that do not contain any active tasks. Hence, tasks that are active in the same mode should be allocated to the same node. Such an example is the computer system of a car which operates differently depending on whether the car is turned on (driving) or off (parked) and where the system should not consume too much power while parked. The opposite of clustering, *anti-clustering*, refers to the case when a set of tasks *cannot* execute on the same node. The typical application for this constraint is when tasks have been replicated to achieve fault-tolerance and the replicas must be allocated to different nodes.

In systems where it is necessary to prevent simultaneous access to indivisible resources, such as critical sections and I/O-devices, *exclusion* constraints determine whether two tasks are allowed to execute concurrently. It should be noted that this is an artificial constraint, the existence of which is merely due to the lack of support for more natural constraints (such as resource-usage constraints) in existing scheduling algorithms.

[3]Note that the communication constraint becomes a pure precedence constraint if the tasks are located on the same node since the transmission delay will be zero.

Constraint	System		Task				Origin		
	Model	Property	Timing		Execution		Natural	Implementation	Artificial
			Inter	Intra	Inter	Intra			
Processors	X							X	
Resources	X							X	
Context switch		X						X	
Transmission delay		X						X	
Resource capacity		X						X	
Execution times		X						X	
Deadlines				X			X		x
Release times				X					X
Periods				X			X		x
Jitter				X			X		
Distance			X				X		
Freshness			X				X		
Correlation			X				X		
Harmonicity			X						X
Preemption						X		X	x
Affinity						X		x	X
Locality						X		X	x
Resource usage						X		X	
Precedence					X		X		
Communication					X		X		
Clustering					X		x		X
Anti-clustering					X			X	
Exclusion					X				X

Table 1: Constraint taxonomy showing which groups a constraint mostly (X) and sometimes (x) belongs to.

3.3 Constraint taxonomy

The constraints and the groups they are divided into constitute a constraint taxonomy which is summarized in Table 1. The table shows whether a constraint is *natural*, *implementation-based* or *artificial*. Natural constraints are directly derived from the system requirements while implementation-based constraints are imposed as a consequence of the choice of hardware architecture and run-time scheduling strategy. Artificial constraints are typically a result of adapting to limitations in existing scheduling algorithms or to control the scheduling of the tasks.

We believe that the constraint taxonomy can aid the system designer in the process of modeling the scheduling problem. With knowledge of common constraint definitions, it becomes easier to identify constraints from the system requirements. Furthermore, the identification becomes more exact which makes the model more accurate, thus increasing schedulability.

4 Definition of Optimality

Optimization of a schedule is possible and desirable since the specification does not completely determine the system. To be able to determine whether a schedule is optimal or not, we need to identify a measure for the quality of a schedule. This measure can involve one or more metrics which each provide information about the schedule's quality. We refer to these two cases as *single-objective* and *multi-objective* optimization problems.

4.1 Single-objective optimization

Single-objective optimization requires that we define an *object function* which computes the value of a schedule. That is, if x is a feasible schedule and f is the object function, then $f(x)$ is the value of the

schedule. In this section, we describe some object functions for single-objective optimization that are relevant in the design of embedded distributed real-time systems.

A traditionally-used optimization criterion for hard real-time systems is the *maximum lateness*, which is the same as the shortest slack in the final schedule. The rationale for using lateness as the optimization criterion is made clear by noting that the lateness will never exceed 0 for a feasible schedule.

In a system with a fixed number of processors, a commonly-used optimization criterion is the *load balance*. This optimization strategy means distributing the tasks between the processors such that they have approximately the same amount of load[4], which makes the most use of the available resources.

If the hardware architecture has not been fixed, on the other hand, it could instead be important to minimize the *number of processors* used in order to keep hardware costs down. To this end, the number of processors could be regarded as unknown and be subject to minimization. However, in reality parameters such as number of processors and other resources are not flexible enough to allow for optimization. Instead, an incremental approach, where various system configurations are manually evaluated, is likely to be more suitable.

For distributed systems, it is sometimes also necessary to optimize with respect to *communication*, that is, the amount of messages sent on the network. In embedded systems, the main arguments for minimizing the time required for message passing are that a low bus utilization may (a) enable the system designer to use a cheaper bus with less communication bandwidth, and (b) reduce the total amount of cabling in the system, thus reducing weight. If contention-based message scheduling is used it is typically desired to group the messages into frames, of a certain size, which are then used to transmit the messages. The objective of such a strategy is that it will minimize the overhead associated with sending/receiving messages.

In control systems, minimizing *jitter* is typically used as the optimization criterion. Note that jitter can be considered as a scheduling objective as well as a constraint. Minimizing jitter means minimizing the drift in the task invocations.

Since many embedded real-time systems are also dependable, it may be necessary to maximize the *reliability* of the system. That is, given the failure rate of each hardware component, the probability of a system failure is minimized.

4.2 Multi-objective optimization

Multi-objective optimization is necessary when we have several object functions which we want to optimize at the same time. In contrast to the single-objective case, we are now confronted with the problem of combining these functions such that a returned solution will be in line with what is considered optimal. We will now discuss some possible combinations.

The simplest approach is to make a single-objective function which is the sum of the participating functions, that is, $f(x) = \sum f_i(x)$. The major disadvantage with this approach is that, if the variation in the values of the different functions is large, some functions may dominate others. For example, assume $f_1(x) \in [0, 2]$ and $f_2(x) \in [0, 42]$. Then, if $f_1(x) = 2, f_2(x) = 21$ we obtain $f(x) = 23$ which should be considered to be better than the case where $f_1(x) = 1, f_2(x) = 28$ (which gives $f(x) = 29$) because $f_1(x)$ is maximized for the former case. However, the latter case has a greater single-objective value (assuming maximization) and will be considered the best solution for this optimization approach. A way to overcome this disadvantage is to assign weights to the functions [32]. It can be difficult, though, to come up with well-working weights.

Another way to solve the unbalance in the summarization approach is to make sure that the value ranges of the object functions are the same [33]. This can be achieved by mapping the original values into ranges of equal size, for example, $[0, 100]$. Our previous example would then give $f_1(x) = 2 \rightarrow 100, f_2(x) = 21 \rightarrow 50, f(x) = 150$ which is better than $f_1(x) = 1 \rightarrow 50, f_2(x) = 28 \rightarrow 67, f(x) = 117$. The size of the mapped range should correspond to the largest value range among the object functions. Otherwise, a (small) increase in its value might fail to increase its mapped value.

If the value ranges are equal yet another approach can be taken. Instead of a sum, the combined value is chosen to be the minimum of all object function values. This value is then to be maximized. This approach tries to balance the function values even further since a function far away from its optimum is more likely to be increased than one that is close.

[4]By 'load' of a processor, we mean the sum of the task execution times on that processor.

It could also be argued that the optimization should be performed in priority order. That is, first optimize on the most important function, and then optimize on the next second most important function under the assumption that value of the first function is maintained. This procedure is then repeated for all object functions. An example where this approach may be applicable is in communication scheduling. In this case the bus utilization is first minimized to determine which messages should be sent. Then, these messages should be formed in as few groups as possible in order to make the transmission less expensive.

5 Constraint Programming Framework

Most proposed real-time scheduling algorithms are "hard-coded" with respect to the constraints, task properties, resources and optimization criteria they can handle. Although a specific method in most cases outperforms a generic one, the former soon becomes quite complex and it becomes difficult to include new features. The constraint programming approach allows constraints to be specified in terms closer to the requirements. The included constraint solver then provides techniques to automatically perform the constraint derivation as well as the scheduling.

5.1 Introduction to constraint programming

The problems addressed using the constraint programming paradigm are called *constraint satisfaction problems (CSP)*. A CSP consists of variables with associated domains and constraints between the variables. A solution to a CSP is an assignment of values to the variables such that all constraints are satisfied. The programming of a CSP can be described in three steps:

(1) Declare the variables (and their domains)
(2) Post the problem constraints
(3) Search for a feasible or optimal solution

The constraint solver uses *propagation* in order to reduce the search space. That is, by considering the constraints the domains of the variables are narrowed by removing values that cannot be part of a solution. However, propagation alone is usually not enough to find a solution, but backtrack searching is also required.

The tool that we have based our framework on is SICStus Prolog [34] and its associated constraint solver for finite domains [35]. An interesting feature of this tool is the possibility to guide the search algorithm by varying the search parameters. By using suitable parameters the search time can be significantly reduced. The translation of the constraints into the code format internal to the solver is fairly straightforward and several examples of how constraints and applications are modeled can be found in [36].

5.2 Problem feasibility

In our case, a schedule is feasible if there exists a solution to the corresponding CSP. It is desired that a feasible schedule is found quickly. Not only from the users point of view, but also in the context of optimization. This schedule is then used as a starting point for further refined attempts to find an optimal solution. In order to speed up the search we aid the constraint solver by supplying heuristics. Intuitively, tasks should be balanced between the nodes in order to increase the chances of meeting their deadlines. However, this is not visible to the solver. Therefore, we pre-assign the tasks to different nodes before the actual search begins. This assignment is undone if it turns out to make the problem infeasible.

5.3 Problem optimality

The optimization algorithm (for minimization) used in the framework can be described as follows:

(1) Find a feasible schedule, x_i
(2) **If** a schedule is found **then** add the constraint $f(x) < f(x_i)$ **else** x_{i-1} is optimal
(3) Increase i and go to (1)

Solution	Loosely constrained	Tightly constrained
Feasible	Easy since many solutions exist	Harder since propagation usually can not reduce the search space enough
Optimal	Can be harder since many candidate solutions exist	Can be easier since the search space is smaller

Table 2: Complexity relationship (of original problem).

This algorithm applies directly to single-objective optimization. For the multi-objective case the object functions are mapped into new ones with range $[0, 100]$ which represents percentage of the optimal solution. We then use a combination of summarization and maximizing minimum. That is, for each function f_j the inequality $f_j(x) \geq \min\{f_j(x_i)\}$ should hold as well as the inequality $\sum f_j(x) > \sum f_j(x_i)$. The framework also supports priority-ordered optimization which is treated as consecutive single-objective optimizations.

Hence, once a feasible schedule is found the search is restarted but with additional constraints. These optimality constraints does not only state that a better solution is required but also prunes the search space due to constraint propagation.

5.4 Problem complexity

The complexity involved in finding a feasible solution versus finding an optimal solution to a CSP depends on the tightness of the problem, that is, the relation between the number of solutions and the size of the search space. The relationship is illustrated in Table 2 which was presented by Tsang in [37].

We can see that, since our constraint derivation approach does not make the problem tighter than necessary, we should be able to quickly locate of a feasible schedule. Furthermore, once a solution is found, our optimization algorithm increasingly tightens the problem and thus continuously decreases the search space.

6 Performance Evaluation

To validate the quality of our approach we will now evaluate the scheduling framework using a set of realistic examples. This will stress both the modeling and scheduling capabilities of the framework. To this end we have studied the following applications:

- A mobile base-station [38] which includes 3 processors and 17 tasks with precedence, communication, clustering, locality and release time constraints. There is one end-to-end deadline.
- A control application [16] which includes 4 processors and 22 tasks with precedence, communication, clustering and locality constraints. There are 5 end-to-end deadlines.
- A safety-critical application [19] which includes 3 processors and 12 tasks with communication and anti-clustering constraints. There are 2 end-to-end deadlines.

6.1 Framework setup

We configured the scheduling framework to model time-driven, non-preemptive task scheduling with affinity (no migration) and contention-based message scheduling.

Each application was tested for feasibility as well as for applicable aspects of optimality. For the single-objective optimization, we have used the objectives that were originally proposed in conjunction with the applications, namely lateness (for the base-station and control applications) and communication (for the safety-critical application). The multi-objective optimization was performed on lateness, load balance and communication. The estimated proximity to the optimum for lateness and communication were given by the constraint solver, while, for load balance, we optimistically divided the tasks between the processors such that the maximum load was minimized. The priority-order optimization was performed in the following order: communication, load balance and lateness. This choice of optimization order is used to reflect that, in a distributed embedded system, the communication may be considered to be the major bottle-neck. Once the communication has been determined, it could then be interesting to make the

Problem	Feasible	Single-objective			Multi-objective	Priority order
		objective	result	performance		
Base-station	0.11/6	lateness	-199	0.14/11	0.18/11	0.29/25
Control	0.10/0	lateness	-21	0.19/5	0.20/5	0.32/7
Safety-critical	0.11/7	communication	20	2.5/1463	1.8/345	4.0/2298

Table 3: Scheduling performance (secs/backtracks).

Solution	Function values			Estimated proximity to optimum			
	communication	load balance	lateness	communication	load balance	lateness	sum
1	32	32	-1	46	92	10	148
2	26	44	-4	56	69	40	165
3	22	44	-4	63	69	40	172
4	20	39	-9	66	78	90	234

Table 4: Multi-objective search trace for the safety-critical example.

best use of the available resources. Finally, lateness can be used to distinguish between otherwise equal solutions.

The examples were scheduled using an Ultra Sparc 10 with 128 M bytes of primary memory.

6.2 Results

The scheduling results are listed in Table 3. Performance is given as x/y where x is the time in seconds and y is the number of backtracks in the constraint solver.

Mainly, the number of backtracks gives information about the difficulty of the problem. The time, however, also depends on the number of constraints and variables which were created, that is, the size of the problem. The differences in the number of backtracks for an example reflect the tightness of the problem. For instance, the number of backtracks required to find an optimal solution for the control application using single-objective optimization (5) is only slightly higher than for a feasible solution (0). Hence, the application constraints almost completely determine the problem which results in a small search space. In contrast, the safety-critical application is subject to a large difference between the number of backtracks used to find the feasible (7) and single-objective (1463) solutions. Hence, this problem is less constrained which results in a large search space. These observations consequently corroborate the relationships listed in Table 2.

Table 4 shows how the search progresses in the multi-objective optimization of the safety-critical application. In the table it can be seen that in the first found solution, the load-balancing objective is estimated to have reached 92 percent of its optimal value while the lateness only has reached 10 percent. The constraint solver then tries to find a solution where all objectives have reached at least 10 percent and where the sum of the estimated proximity values is larger than the current sum (148 percent). In the next found solution, the lateness and the communication have increased their percentage at the cost of the load balance. However, since the total sum is greater (165 percent) than before this new solution is regarded as a better one. After four iterations optimum (of the estimated proximity) is obtained despite a rather low proximity of optimum for the individual functions. The main reason for this is that the estimation mechanisms used are too weak, something we will elaborate further on in Section 7.

6.3 Complexity comparison

To demonstrate how a problem is affected when additional constraints are introduced, we solve the scheduling problem for the control application using the DACAPO communication model [29]. The DACAPO model is more restrictive than the contention-based model used so far since communication must now be synchronized with time slots. The DACAPO model assumes that each processor is given dedicated time slots where tasks on that processor are allowed to send messages. The time slots are of fixed size and

Strategy	Slot size	Backtracks	Optimal result
Contention	-	5	-21
DACAPO	1	13	-20
DACAPO	2	1	-16
DACAPO	5	1	no solution

Table 5: Effects of over-constraining in the control application.

are divided equally between the processors. It is assumed that the least common period (length of the schedule) is a multiple of the number of processors. Furthermore, the total amount of transmission time for a processor should be a multiple of the size of the time slots. A message must fit into one time slot. The slots are assigned to the four processors in a round-robin order[5].

As can be seen in Table 5, the use of the DACAPO model transforms the original problem into a non-optimal (or even infeasible) one. The table also demonstrates that, for a slot size of 2, the complexity was reduced, while, for a slot size of 1, the complexity was increased. Whether a constraint decreases or increases the complexity depends on its strength, that is, how much the constraint solver can reduce the search space due to constraint propagation in relation to the number of solutions. For example, a DACAPO slot size of 1 probably only restricts the number of solutions, while a slot size of 2 also restricts the search space as indicated in the table. A slot size of 5 seems to restrict the search space as well, but unfortunately also over-constrains the problem so that no solution is found at all.

7 Discussion

Constraint programming is a declarative approach where it suffices to express what constitutes a solution. The constraint solver then handles the actual search. However, this search can be significantly improved by using knowledge about the problem as guidance. The SICStus Prolog constraint solver offers the possibility to supply search heuristics. So far, we have not looked much into this, but we believe that by tailoring the search algorithm to the specific problem, complexity can be significantly reduced. Previous work for the B&B algorithm has shown that this is a viable approach [4, 5].

A CSP can usually be modeled in many different ways. How a problem is modeled affects the complexity of solving it. By remodeling or supplying redundant constraints the constraint propagation can be more effective which reduces the search space. It would be interesting to see if such reformulation is possible.

As can be seen in Table 4, the multi-objective optimization strategy obtains an optimal value (20) of the communication. However, the estimated proximity of the communication in the optimal solution is only 66 percent. Since this is the lowest value among all estimated values in the fourth iteration, no solution with a lower estimated proximity value for the communication will be accepted. On the other hand, if that value had been more accurate (for example, close to 100 percent), a decrease in the estimated proximity for the communication would be tolerable if the other proximity values increased, resulting in a better solution. Hence, it would be of great value to have a tight estimation of optimum since this is essential to make our multi-objective optimization approach work well and also reduce the problem search space. Thus, further investigation on how to obtain fast, but accurate, estimations of the optimal value is desired.

8 Conclusions

Scheduling of real-time tasks in an embedded distributed system is a difficult problem since such systems not only have standard task-level timing constraints (such as periods and deadlines) but also have more advanced constraints such as locality (what processor to execute on) and clustering (what other tasks to execute with). Unfortunately, few existing scheduling algorithms are capable of accounting for such diverse set of constraints. Moreover, scheduling of embedded systems often requires multi-objective optimization

[5]In our test run we used the slot-order $\{3, 4, 1, 2\}$. However, other slot orders were found to produce similar results.

in order to simultaneously account for requirements such as schedulability, reliability, power consumption and economic cost.

In this paper, we have proposed a scheduling framework based on constraint programming that is capable of providing these features. One main advantage of the constraint programming paradigm is that it suffices to specify what defines an acceptable solution, instead of having to provide guidelines on how to find it. We have argued that, by using this framework, it is easy to model relevant constraints as well as perform single- and multi-objective optimization for embedded distributed real-time systems. To this end, the applicability of the framework was evaluated using a set of realistic applications with non-trivial constraints.

References

[1] M. Saksena, "Real-Time System Design: A Temporal Perspective," *Proc. of IEEE Canadian Conference on Electrical and Computer Engineering*, Waterloo, Canada, May 1998, pp. 405–408.

[2] K. Ramamritham, "Where do Time Constraints Come From and Where do They Go?," *International Journal of Database Management*, vol. 7, no. 2, pp. 4–10, 1996.

[3] J. A. Stankovic, "Strategic Direction in Real-Time and Embedded Systems," *ACM Computing Surveys, 50th Anniversary Issue*, vol. 28, no. 4, pp. 751–763, Dec. 1996.

[4] J. Jonsson, "Effective Complexity Reduction for Optimal Scheduling of Distributed Real-Time Applications," *Proc. of the IEEE Int'l Conf. on Distributed Computing Systems*, Austin, Texas, May 31 –June 5, 1999, pp. 360–369.

[5] I. Ahmad and Y.-K. Kwok, "Optimal and Near-Optimal Allocation of Precedence-Constrained Tasks to Parallel Processors: Defying the High Complexity Using Effective Search Techniques," *Proc. of the Int'l Conf. on Parallel Processing*, Minneapolis, Minnesota, Aug. 10–14, 1998, pp. 424–431.

[6] G. Fohler, "Dynamic Timing Constraints — Relaxing Overconstraining Specifications of Real-Time Systems," *Proc. of the IEEE Real-Time Systems Symposium – Work-in-Progress Session*, San Francisco, California, Dec. 3–4, 1997, pp. 27–30.

[7] M. Ryu, S. Hong, and M. Saksena, "Streamlining Real-Time Controller Design: From Performance Specifications to End-to-End Timing Constraints," *Proc. of the IEEE Real-Time Systems Symposium*, San Francisco, California, Dec. 3–5, 1997, pp. 91–99.

[8] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin, "On Task Schedulability in Real-Time Control Systems," *Proc. of the IEEE Real-Time Systems Symposium*, Washington, D.C., Dec. 4–6, 1996, pp. 13–21.

[9] D. Seto, J. P. Lehoczky, and L. Sha, "Task Period Selection and Schedulability in Real-Time Systems," *Proc. of the IEEE Real-Time Systems Symposium*, Madrid, Spain, Dec. 2–4, 1998, pp. 188–198.

[10] R. Gerber, S. Hong, and M. Saksena, "Guaranteeing Real-Time Requirements with Resource-Based Calibration of Periodic Processes," *IEEE Trans. on Software Engineering*, vol. 21, no. 7, pp. 579–592, July 1995.

[11] M. Saksena and S. Hong, "Resource Conscious Design of Distributed Real-Time Systems: An End-to-End Approach," *Proc. of the IEEE Int'l Conf. on Engineering of Complex Computer Systems*, Montreal, Canada, Oct. 21–25, 1996, pp. 306–313.

[12] U. Nilsson, S. Streiffert, and A. Törne, "Detailed Design of Avionics Control Software," *Proc. of the IEEE Real-Time Systems Symposium*, Madrid, Spain, Dec. 2–4, 1998, pp. 82–91.

[13] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic Scheduling of Real-Time Tasks under Precedence Constraints," *Real-Time Systems*, vol. 2, no. 3, pp. 181–194, Sept. 1990.

[14] B. Kao and H. Garcia-Molina, "Deadline Assignment in a Distributed Soft Real-Time System," *Proc. of the IEEE Int'l Conf. on Distributed Computing Systems*, Pittsburgh, Pennsylvania, May 25–28, 1993, pp. 428–437.

[15] R. Bettati and J. W.-S. Liu, "End-to-End Scheduling to Meet Deadlines in Distributed Systems," *Proc. of the IEEE Int'l Conf. on Distributed Computing Systems*, Yokohama, Japan, June 9–12, 1992, pp. 452–459.

[16] M. Di Natale and J. A. Stankovic, "Dynamic End-to-End Guarantees in Distributed Real-Time Systems," *Proc. of the IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, Dec. 7–9, 1994, pp. 216–227.

[17] J. J. Gutiérrez García and M. González Harbour, "Optimized Priority Assignment for Tasks and Messages in Distributed Hard Real-Time Systems," *Proc. of the IEEE Workshop on Parallel and Distributed Real-Time Systems*, Santa Barbara, California, Apr. 25, 1995, pp. 124–132.

[18] J. Jonsson and K. G. Shin, "Robust Adaptive Metrics for Deadline Assignment in Distributed Hard Real-Time Systems," *Real-Time Systems: The International Journal of Time-Critical Computing Systems*, 2000, (to appear).

[19] K. Ramamritham, "Allocation and Scheduling of Precedence-Related Periodic Tasks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, no. 4, pp. 412–420, Apr. 1995.

[20] T. F. Abdelzaher and K. G. Shin, "Period-Based Load Partitioning and Assignment for Large Real-Time Applications," *IEEE Trans. on Computers*, vol. 49, no. 1, pp. 81–87, Jan. 2000.

[21] K. Tindell and J. Clark, "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems," *Microprocessing and Microprogramming*, vol. 40, no. 2/3, pp. 117–134, Apr. 1994.

[22] K. W. Tindell, A. Burns, and A. J. Wellings, "Allocating Hard Real-Time Tasks: An NP-Hard Problem Made Easy," *Real-Time Systems*, vol. 4, no. 2, pp. 145–165, June 1992.

[23] T. Shepard and J. A. M. Gagné, "A Pre-Run-Time Scheduling Algorithm for Hard Real-Time Systems," *IEEE Trans. on Software Engineering*, vol. 17, no. 7, pp. 669–677, July 1991.

[24] J. Xu, "Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Trans. on Software Engineering*, vol. 19, no. 2, pp. 139–154, Feb. 1993.

[25] D. Peng, K. G. Shin, and T. Abdelzaher, "Assignment and Scheduling of Communicating Periodic Tasks in Distributed Real-Time Systems," *IEEE Trans. on Software Engineering*, vol. 23, no. 12, pp. 745–758, Dec. 1997.

[26] C.-J. Hou and K. G. Shin, "Allocation of Periodic Task Modules with Precedence and Deadline Constraints in Distributed Real-Time Systems," *IEEE Trans. on Computers*, vol. 46, no. 12, pp. 1338–1356, Dec. 1997.

[27] M. Schild and J. Würtz, "Off-Line Scheduling of a Real-Time System," *Proc. of CP97 Workshop on Industrial Constraint-Directed Scheduling*, Schloss Hagenberg, Austria, Oct. 1997.

[28] H. A. Hansson, H. W. Lawson, M. Strömberg, and S. Larsson, "BASEMENT: A Distributed Real-Time Architecture for Vehicle Applications," *Real-Time Systems*, vol. 11, no. 3, pp. 223–244, Nov. 1996.

[29] B. Rostamzadeh, H. Lönn, R. Snedsböl, and J. Torin, "DACAPO: A Distributed Computer Architecture for Safety-Critical Control Applications," *Proc. of the IEEE Int'l Symposium on Intelligent Vehicles*, Detroit, Michigan, Sept. 25–26 1995, pp. 376–381.

[30] H. Kopetz and G. Grünsteidl, "TTP – A Protocol for Fault-Tolerant Real-Time Systems," *IEEE Computer*, vol. 27, no. 1, pp. 14–23, Jan. 1994.

[31] K. W. Tindell, H. Hansson, and A. J. Wellings, "Analysing Real-Time Communications: Controller Area Network (CAN)," *Proc. of the IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, Dec. 7–9, 1994, pp. 259–263.

[32] C. C. Amaro, R. Nossal, and A. D. Stoyen, "On Cost Function Synthesis For Multi-Objective Design Decisions in Complex Real-Time Systems," *Proc. of Int'l Conference on Engineering of Complex Computer Systems*, Las Vegas, Nevada, Oct. 18–21, 1999, pp. 86–97.

[33] P. J. Bentley and J. P. Wakefield, "An Analysis of Multiobjective Optimization within Genetic Algorithms," Tech. Rep. ENGPJB96, Division of Computing and Control Systems Engineering, The University of Huddersfield, Huddersfield HD1 3DH, U. K., 1996.

[34] Intelligent Systems Laboratory, *SICStus Prolog User's Manual*, Swedish Institute of Computer Science, 1995.

[35] M. Carlsson, G. Ottosson, and B. Carlson, "An Open-Ended Finite Domain Constraint Solver," *Proc. of the Int'l Symposium on Programming Languages: Implementations, Logics, and Programs*, H. Glaser et al., Eds., Southampton, UK, Sept. 3–5, 1997, vol. 1292 of *Lecture Notes in Computer Science*, pp. 191–206, Springer Verlag.

[36] C. Ekelin and J. Jonsson, "A Modeling Framework for Constraints in Real-Time Systems," Tech. Rep. 00-9, Dept. of Computer Engineering, Chalmers University of Technology, S-412 96 Göteborg, Sweden, May 2000, Available at http://www.ce.chalmers.se/~cekelin/constraints.us.ps.gz.

[37] E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[38] Y. Zhao, "Derivation of Local Timing Constraints in Early Design Stages," Master's thesis, Dept. of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1999.