Multiprocessor Scheduling using the DÉJÀ VU Scheduling Class Library*

Jürgen Dorn  Anna Prianichnikova
Markus Stumptner  Helmut Veith

Institut für Informationssysteme
Technische Universität Wien
{dorn, annap, mst, veith} @dbai.tuwien.ac.at

Johannes Reisinger  Ralf Schlatterbeck

System Design, FREQUENTIS
Vienna, Austria
{jreising, rschlatt} @frequentis.com

Abstract: The V4 system marketed by FREQUENTIS is a dependable voice and data
communication system for a wide area of applications, most notably for air traffic control
and digital trunked radio systems. Internally, V4 is based on a time triggered operating
system. To obtain predictable reaction and performance under arbitrary circumstances, V4
uses an offline task and message scheduler. In this paper, we describe work in process
concerning the development of a scheduler for V4 using the DÉJÀ VU application
framework for scheduling software in a joint project of the Institut für
Informationssysteme and FREQUENTIS.

1. The V4 System

V4 has been developed by the Viennese company FREQUENTIS which has five decades of
experience in supplying communication systems for Air Traffic Control. V4 supports voice, data and
multimedia transmission.

Such critical communications applications require highly reliable, fault-tolerant switching systems,
which guarantee that communication links between air traffic controllers, aircrafts, and adjacent
ground facilities are established without delay. To this aim, the V4 switching platform has a
distributed internal architecture containing a triple redundant internal communication ring where
application critical processors are triplicated in synchronous active redundancy, cf. Fig. 1. The V4
operating system is time-triggered. In the time-triggered approach, all communication and processing
activities are initiated at pre-determined points in time already scheduled before system start. An event-
triggered system, in contrast, would require a dynamic scheduling strategy to activate the appropriate
software tasks that service the event. This would not satisfy the high reliability and availability
requirements.

*This work is partly supported by the Austrian Science Fund. Project No. P12902-INF
Fig. 1: V4-Ring

In the real time multi-tasking operating system of V4, processes are executed according to a pre-calculated static schedule. The processes read and produce statically scheduled messages. Due to these properties, redundant copies of processes are (in the fault-free case) started at the same point in global time, with exactly the same input values.

2. Problem Description

2.1. The V4 System

From the scheduling point of view, a V4 system can be abstractly described as follows:

- **Hardware:** A number of processing nodes equipped with modules (CPUs and interfaces); every CPU has its own memory. The nodes are connected by a communication ring; the capacity of the ring is given by the number of bytes (timeslots/packets) per time unit (frame cycle). Every module contains a special part of the memory (called “message base”) where ingoing and outgoing messages are stored. A specific aspect of the V4 scheduler is that the modules are only partially assigned to the nodes in advance; thus the scheduler needs to assign processors to modules.

- **Software:** A number of processes each consisting of threads. A thread is a piece of software with a known maximum execution time that usually requires some input messages to be read and produces some output messages to be accessed by other threads. Every thread is executed periodically and can have an individual period. Thus, we can speak of different instances of a single thread or message. Threads and messages may be preempted by other threads and messages respectively. The memory of the preempted thread is stored on a stack. Consequently, when the preemption thread terminates, the preempted thread has to continue on the same module.
• **Periodicity of Time:** The communication (message) and thread scheduling is carried out in units of frame cycle FC, which amount to 125μs. The schedule decisions happen only at definite time points – frame cycles and dispatcher cycles (8FC). The schedule itself is periodic; its period is called application cycle (up to 1000 FC). The application cycle AC is the least common multiple of all periods of threads. There may be up to a few thousand threads in the system. An important issue connected to periodicity is to keep consistency between message instances. Concerning the latter, just note that the periodicity may result in a situation where one instance of a message is being read while the next instance already is being written; consequently several instances of a message may have to be kept in the message base (and distinguished) simultaneously. The message consistency has to be guaranteed by the scheduler.

In addition, many technical details have to be exactly modeled in the scheduler; they are crucial to determine the exact times and the fine structure of the scheduler’s dispatcher table.

### 2.2. Hard Constraints

All threads of a process have to be allocated to the same CPU. Threads and messages have to be allocated to the CPU and across the communication ring in such a way that every thread periodically receives and sends its messages in time. Thus, there are precedence constraints between threads and messages. In addition, deadlines (i.e., time constraints), and mutual exclusion constraints for threads have to be satisfied. Messages within the same CPU can be treated as pure precedence constraints, and need not be allocated on the ring. Due to the triple redundant internal communication mentioned above, the constraints of the replication mechanism also have to be satisfied. All constraints are hard, i.e. they have to be **exactly** satisfied (without any tolerance) to obtain a schedule. The precedence constraints are visualized in a graph called extended precedence graph, cf. fig. 2.

### 2.3. Criteria for Good Schedules

Unlike in usual scheduling problems, the maximum completion time is fixed and cannot be improved by the scheduler. Thus, the main quality criterion are the hardware costs; in addition, secondary criteria which improve the readability and simplicity of the schedule are defined.

#### Hardware Costs:

The hardware costs are influenced by the number of modules (CPUs), and the memory requirements of the modules. These two criteria are mutually dependent: reducing the number of modules by one may even double the memory requirements for the remaining modules due to new preemptions and larger message bases. On the other hand, less memory means less freedom for preemptions and communication, and may thus increase the number of necessary processors.

**Fig. 2:** An extended precedence graph, and a simplified schedule on three modules; in the schedule, only the messages to later instances are indicated by arrows.
The memory can be minimized by reducing the values of the following parameters:

1. **Communication on the ring.** This has two effects on the memory: first, more copies in the message base memory are needed if the receivers reside on different modules, cf. (4), and second, local messages are not included in the dispatcher table, cf. (2).

2. **Size of the schedule.** This means the length of the dispatcher tables. Since the dispatcher tables have to be kept in main memory, and may become very large, they have a strong influence on the memory requirements.

3. **Number of Preemptions.** Since for every preemption, the memories of the preempted and the preempting thread have to be kept in parallel, this has also a direct effect on the memory requirements.

4. **Message Base.** If one instance of a message is written while a reader has not yet read a previous instance of this message, both messages have to be kept in the message base, and thus more memory is needed on the module. In the case of several active readers the message base has to keep even more than two copies, if the message goes to different modules, cf. (1).

**Secondary Criteria:**

Several additional criteria make the schedule better readable (for possible manual modifications) and better adaptable, if new processes and threads are added. To this aim, the scheduler follows the following guidelines, in particular as heuristics in situations, where several choices are possible:

5. **Keep Overheads Low.** Overheads occur because of preemptions, handling of non-local messages, and switching of processes. All these factors are not only time-consuming, but also make the schedule hard to read. Therefore, their occurrence should be avoided, if possible.

6. **Distribute CPU load uniformly.** When the CPUs have equal load, adding new processes is simplified; moreover, the system will be more flexible if in practical applications, unexpected events occur.

The last criteria is the following:

7. **Latency of communication.** The overall transmission time for all messages in the system (time from the deadline of a sending thread to the deadline of a receiving thread) has to be minimised.

### 3. The DÉJÀ VU Application Framework

The DÉJÀ VU scheduling class library is an object-oriented application framework [FS98] supporting developers in the construction of scheduling systems for industrial production processes. DÉJÀ VU has been developed by the Industrial Information System group with a focus on the reusability of an abstract core [DGV98]. Currently, the core consists of about 150 classes, approximately 300,000 lines of code, and 15 person years have been invested. A public accessible documentation is available on the Internet [DÉJÀVU]. DÉJÀ VU has been applied successfully to the steel plant Böhler [DGGMV96].
The DÉJÀ VU architecture is structured in three layers:

- an abstract core layer defining the default behavior of scheduling systems,
- a layer specializing the behaviour for certain sub-domains such as steelmaking applications, and finally
- a layer with classes specialized for individual applications.

From an AI point of view, the abstract core defines a model-based behavior for scheduling problems. Here, algorithms are applied that are capable of solving a wide area of problems. For example, the temporal reasoning algorithms in the core are reusable for any scheduling problem. The drawback of the generality is the required effort for the algorithms. The intermediate layer defines restricted models where certain assumptions about the sub-domain lead to either more tractable or to more specialized algorithms. For example, if flow-shop schedules are sufficient for a sub-domain, scheduling theory says that many algorithms are applicable that are not applicable for general job-shop scheduling [Fre72]. The third layer enables the realization of application-dependent heuristics and thus to tune the problem solving process.

Besides the obvious goal of solving the multi-processor scheduling problem, another scientific aim of the project is more related to software engineering: to verify the reusability of the scheduling class library for a scheduling application outside its original scope.

A large part of the object-oriented design is inherited from the DÉJÀ VU system. On the other hand, it soon became clear that being developed primarily for industrial production processes, DÉJÀ VU lacks certain mechanisms which are specifically needed for the V4 domain. In the process of adapting DÉJÀ VU to the new application, the computer scheduling sub-domain was introduced. Care was taken to introduce new computer scheduling classes which are general enough to fit into the DÉJÀ VU design philosophy of easy reusability. These classes should be reusable if further applications in the sub-domain of computer scheduling shall be developed. The following UML-class diagram shows the most important parts of the scheduling framework and the extensions available for computer scheduling.
Typically, industrial scheduling systems continually obtain orders from a production planning system that describe which products have to be delivered until some due date. The scheduler must then find a process plan that describes how a product can be produced and allocates the required operations on existing resources. Orders are records of a certain format that are read into an order list in the scheduler. The read-method that reads an individual order is intended to be reused by all production process schedulers. The existing resources and technical constraints of these resources are given once and must not be reread whenever new orders arrive.

The input specification for the V4 scheduler is different. Since V4 requires an off-line scheduler all requirements are given at the same time. These requirements are given in a file using a defined input language. The required operations in V4 are threads and messages that are allocated to modules located on nodes. Although there are no products defined, we can interpret a computer process containing threads as an order to be scheduled. If we derive computer process from the existing order class, these processes may play the role of orders in the existing framework. This means, we may reuse the user interface behavior and the standard handling of orders.

### 3.2 Allocating operations

During scheduling allocation of processes predetermines allocation of their threads and messages. An important difference to production process scheduling is the need of allocating different instances of a thread. If a thread has a period of 1ms, then every millisecond an instance of this thread has to be allocated to a computer module. Since threads have different periods, the scheduling time horizon is the least common multiple of all periods of threads.
4. Problem Solution

4.1. Existing Methods and Algorithms for Computer Scheduling

In the literature, only very simplified cases of multiprocessor scheduling with precedence constraints have been studied. As stated in [BEPSW96], “Very little is known about the design of efficient approximation algorithms for the scheduling of tasks on a real multiprocessor topology. Of particular practical interest are also problems with constraints on communication channel capacities ...”.

The restrictions of previous published work include that messages are treated either as pure precedence constraints, or as precedence constraints with time, instead of separate entities which have to be scheduled on a resource. Moreover, the usual notion of preemption in the scheduling literature allows to continue a preempted task on a different resource, in contrast to our specification. In addition, since the allocation of the threads to modules is determined by the allocation of the processes, an additional level of abstraction is introduced. Nevertheless, several important algorithms can be used as a basis for collecting experience and devising more adequate algorithms, in particular the work of Coffman and Graham [CG72], Giffler and Thomson [GT60], as well as the more recent Earliest Task First algorithm by Hwang et al [HCAL89].

All mentioned algorithms try to linearize the precedence graph by implementing a ready list of tasks which may be executed, and a heuristic selection function which determines the next executed task among them. A major difference to the current problem is that in our case, the precedence graph is cyclic (since a message may be sent to a later instance of a periodic thread), and thus by unwinding the precedence graph we obtain a large graph whose diameter is determined by the least common multiple of the periods of the threads, and thus by a potentially very large number.

An alternative approach using Iterative Deepening A* [F88], [F94] was tested, but the success of this algorithm very much depends on a good heuristic function. It is very difficult to find one, that is not itself NP complete, if mutex constraints and message scheduling are taken into account.

A first implementation at FREQUENTIS has used genetic algorithms to generate a schedule. While this method gives correct schedules, the run time of the heuristic evaluation function is roughly proportional to the length of the schedule. This may lead to problems with long schedules because the evaluation function has to be called for every generated schedule. In addition the schedules produced are highly unstable, in that a small change in the specification usually leads to a widely different solution.

Therefore, we describe a more goal-oriented search algorithm in the following section.

4.2. Algorithms for the DÉJÀ VU Computer Scheduler

The DÉJÀ VU Computer Scheduler is based on the following principles:

Partition: Initially, the scheduler will try to allocate the processes in such a way that possibly many messages between threads are within a CPU, i.e., we decompose the set of modules into components such that most of the messages are sent within components. This is achieved by heuristic evaluation of inter-process communication. This step is of particular importance, since it predetermines the allocation of all threads to a high extent.

Initial Plan Generation: The first plan is generated using a hybrid form of the Earliest Task First algorithm [HCAL89] and the Giffler-Thomson algorithm [GT60] mentioned in section 4.1.; for this initial schedule, only the main constraints are considered (precedence, time constraints, and mutual exclusion) to keep the size of the search space low.
An important modification is that both messages and threads are treated as different types of tasks. The algorithm **ResolvePrecedence** (fig. 4) unwinds the precedence graph step by step in contrast to general purpose solution strategies as genetic algorithms or simulated annealing. It uses a function \( \text{level}(t) \) which heuristically orders the priority of the tasks; using probabilistic dispatching it combines various strategies and easily extendible: for example, a task has high priority, if its earliest start time \([\text{HCAL89}]\) or its earliest finishing time \([\text{GT60}]\) are low, and if recursively, the tasks depending on it have high cumulated priority. In addition, precedence level functions like in \([\text{CG72}]\) can be used to refine the heuristics.

Program **ResolvePrecedence**;

\[
T := \text{tasks without predecessors}; \quad \text{time} := 0;
\]

while \( T \) not empty

\[
\begin{align*}
\text{MAX} & := \max\{\text{level}(t) : t \in T\}; \\
T' & := \{t \in T : \text{level}(t) = \text{MAX}\}; \\
\text{allocate one random thread } t \text{ of } T'; \\
T & := T - \{t\}; \\
\text{time} & := \text{next possible time point}; \\
T & := T \cup \{\text{all threads whose predecessors finished at } \text{time}\};
\end{align*}
\]

**Fig. 4:** Algorithm **ResolvePrecedence**

Note that the points where random choices are made (i.e., where \( |T'| > 1 \)) can be remembered as backtracking positions.

**Optimization:** Using the tabu search method implemented in Déjà Vu, the initial schedule is gradually improved in order to fulfill also the minor constraints while optimizing the above mentioned quality criteria, i.e., schedule size and costs.

**5. Conclusion**

This paper describes the application of the Déjà Vu applications class library to process and message scheduling for the V4 telecommunication architecture system. Use of the Déjà Vu framework enables the incremental development of the schedule by a limited extension and modification of the existing class library. We have described how new, problem specific classes are inherited from and use the general classes in the framework and have given an outline of the Déjà Vu computer scheduling algorithm implemented by the new classes.

**References**


[DÉJÀVU] *Documentation of the DéjàVu scheduling class library.*
http://www.dbai.tuwien.ac.at/proj/DejaVu/document/


