Application of Genetic Algorithms for LET-based Communication Scheduling

Johannes Pletzer, Josef Templ

C. Doppler Laboratory
Embedded Software Systems
University of Salzburg
Austria

Technical Report T025
22 October 2008

Application of Genetic Algorithms for LET-based Communication Scheduling[1]

Johannes Pletzer, Josef Templ

ABSTRACT

We propose a genetic algorithm for communication scheduling for distributed, time-triggered real-time systems. It is designed for systems based on the Logical Execution Time (LET) programming paradigm which enables the platform independent description of the timing behavior of such systems. We already developed a code generation framework that deals with communication and task scheduling and their interdependence. The framework is extensible by the use of plug-ins, which are used to support specific node platforms and communication protocols and therefore guarantee a clear separation of platform independent from platform dependent concerns. We extended this framework so that the platform-independent communication scheduling can also be customized by the use of a plug-in. On basis of that we implemented an additional scheduling strategy which bases on a genetic algorithm. This work is still work-in-progress but our first results indicate the feasibility of the application a genetic algorithm for our specific communication scheduling problem. Further work will focus on various improvements which we think will lead to a scheduling solution for production use.

1. Framework extensions

We extended our code generation and scheduling framework [1] so that the platform-independent communication scheduling strategy can also be customized by use of a plug-in. For that purpose we restructured the frame generator and updated the interface of the frame scheduler plug-in.

1.1. Frame generator

The frame generator is platform-independent and essentially analyzes the communication required between all LET-based modules in the system according to their node assignments. As described in [1], for every task that needs to transmit a message on the bus there is a communication window for doing so inside its LET period. The frame generator computes all those communication windows and generates a list of messages that need to be transferred between nodes and determines for every message a release time and a deadline.

Communication frames. Out of the list of messages the frame generator computes a list of frames that need to be transferred between nodes. Frames can be described as a container for messages and have a size, a release and deadline constraint, a sender and a list of receiver nodes. The frame window is the period between the release time and the deadline of a frame. Motivated by the application of a genetic algorithm for communication scheduling we extended our

[1] This technical report is based on the paper "A Code Generation Framework for Time-Triggered Real-Time Systems" [1] which was submitted to the Design, Automation & Test in Europe (DATE09) conference and is currently under review. It is included in this report as appendix.

framework so that the strategy by which messages are assigned to frames is easily exchangeable. Originally the strategy was fixed and relied on heuristics that assigned messages with the same sender and similar release time and deadlines to the same frame while trying to minimize the number of frames that are generated. The algorithm is explained in detail in [2, 3, 4]. We had to restructure the algorithm as the generation of the list of messages and the creation of frames were intertwined. Now we first generate a complete list of messages of the whole system and then assign them to frames which specify frame windows. This allowed us to specify a clean interface for the frame window generation:

```
public interface MessageToFrameWindowsStrategy {

  public List<Frame> assignMessagesToFrames(List<Message> messages,
         CommSchedulerPlugin plugin, BuildProgress buildProgress)
         throws Exception;

}
```

The function `assignMessagesToFrames` returns a list of frames on basis of a list of messages. Furthermore it is supplied with a `CommSchedulerPlugin` which is the frame scheduler plug-in which we discuss in detail below. It is used to assign concrete timings for frames and also to get information on if a set of frames are schedulable at all on a concrete communication protocol and how good the solution is via a metric in the range between 0 and 1. An instance of the class `BuildProgress` is provided to allow the strategy to output information on its progress. This is especially useful when it can be expected that the algorithm takes a considerable amount of time as it is typically the case with genetic algorithms.

We changed the existing algorithm so that it works with the new interface and also improved it so that it iterates over multiple set of frames by varying the threshold that controls whether to create a new frame for a message or to assign it to an already existing frame. During the algorithm a metric is applied which measures the compatibility of a message to an already created frame. This leads to different sets of frames which we all pass to the frame scheduler plug-in and use the one which gets the best metric which the plug-in provides as measurement on how good the solution is. This iterative approach also finds solutions which were not found before using a fixed threshold value.

1.2. Frame scheduler plug-in

The frame scheduler is platform specific and must be implemented for a concrete communication protocol. As outlined above, the frame scheduler determines the frame start and end times so that they respect the properties of the underlying bus protocol.

Frame scheduling. The frame generator supplies for every frame a window in which it must be scheduled. Note that however it does make a difference where exactly in this window a frame is scheduled. The start of this window is only a best-case estimate and it is not guaranteed that the whole system, including tasks on nodes, is schedulable if frames are actually scheduled at the beginning of this window as this might constrain the task scheduler too much so that no valid task schedule can be found. Therefore we require the frame scheduler plug-in to schedule all frames as late as possible. The transmission time of a frame of course depends on bus specifics such as the bus speed and timing properties such as inter-frame gaps. All these requirements must be taken into account for calculating correct frame start and end times. The frame scheduler also can merge

frames if their timing requirements are compatible and they have the same sender.

The frame scheduler plug-in is provided with an interface to check whether the calculated start and end times for frames lead to a feasible schedule on node level. This is a key feature of our framework as it takes into account the interdependence of communication and node task schedules and thus prevents the whole code generation process from running into a dead end by creating a communication schedule for which not all nodes are able to come up with an appropriate task schedule.

As a recent extension we added a floating point return value to the scheduling method which the frame scheduler plug-in must implement:

```
public double scheduleFrameWindows(List<Frame> frameWindows)
       throws UnfeasibleCommScheduleException, Exception;
```

This value represents a metric on how good a solution is on a specific communication protocol. It is negative when the set of frames is not schedulable at all. This might for example occur when the bandwidth of the protocol simply is too small to accommodate all frames that must be transferred or also when the resulting frame schedule leads to nodes where no task schedule can be found.

2. Genetic algorithm

Scheduling problems are a common field of application for genetic algorithms and so we decided to evaluate whether they are also suitable to tackle our specific scheduling problems concerning LET-based systems.

We found that genetic algorithms have already been successfully applied in the field of communication scheduling, for example for the FlexRay protocol [6]. The approach described in [6] allocates a set of tasks to nodes which are connected via a FlexRay bus [5] so that various constraints concerning task deadlines and message response and freshness times are met. Our application of the genetic algorithm differs as the task (or module) to node mapping is supplied by the user and our all constraints are directly derived from the LETs of the tasks. Also note that as long as those LET-imposed constraints are met the observable behavior of the system does not depend on when exactly a message is sent. Furthermore our approach is not restricted to a specific communication protocol and consequently is located in the platform independent part of our scheduling framework.

A typical genetic algorithm requires two things to be defined:

- a *genetic representation* (DNA) of the solution domain
- a *fitness function* to evaluate the solution domain

In our case the solution domain are all possible message to frame assignments where the number of messages is fixed for a given system and the number of frames can vary from 1 up to the number of messages. Consequently we use an array of integer values with length equal to the number of messages as DNA encoding. We assign each message to a frame by assigning an integer frame number to each message, i.e. DNA[i] = j associates frame j to message i. An advantage of this genetic representation is that every message is assigned to a frame and that the number of frames can vary between 1 up to the total number of messages. One major disadvantage however is that it is possible to create invalid solutions as there are restrictions on what messages can be assigned to the same frame. In order to improve the result when performing crossover of two DNAs, we apply a normalization algorithm every time the DNA is changed, which is after initialization and when applying crossover and mutation. The

normalization purges unused frame numbers and sorts frames by how many messages are assigned to it while assigned the number 0 to the frame with the most messages assigned to it and so on. Consider the following DNA example:

```
5 5 5 3 2 3
```

Our normalization step would transform this sequence to

```
0 0 0 1 2 1.
```

The fitness function calculation is actually divided into two steps. As first step we apply a fast check on a solution by checking two basic criteria every valid solution must fulfill. Those are that a frame only is allowed to contain messages from the same sender node and no messages with conflicting timing requirements are assigned to the same frame. Only when this check passes we apply our more sophisticated and also much more expensive fitness calculation function. We decided that a reasonable fitness function depends on what communication protocol is actually used to take into account its specific properties. Consequently the fitness calculation is done by the communication scheduling plug-in via the return value of the schedule function `scheduleFrameWindows` as described above. It also takes into account whether the assigned frame timings lead to a feasible task schedule on all nodes involved. In both steps negative fitness values indicate solutions that are unschedulable.

A generic genetic algorithm works as follows:

1. Generate initial population
2. Evaluate the fitness of each individual in the population
3. Repeat until termination criteria is met:
    a) Select fittest individuals to reproduce
    b) Breed new generation through crossover and mutation (genetic operations) and give birth to offspring
    c) Evaluate the individual fitness of the offspring
    d) Replace worst ranked part of population with offspring

In step 1 we simply use a random initialization, i.e. we assign each message a random frame with a number from 0 to (number of messages - 1) and apply our DNA normalization as proposed above.

The fitness evaluation in step 2 and step 3c requires that the DNA of each individual is converted to a list of frames with messages assigned according to the DNA frame to message mapping. Only after that conversion the fitness calculation as detailed above can be applied.

As termination criteria in step 3 we currently use a fixed number of generations that can be specified by the user. As further useful termination criteria we see a timeout or also a termination if the mean or maximum fitness does not increase anymore during a certain number generations.

The crossover and mutation step 3c is performed by selecting a random crossover point and creating a new DNA by using two individuals out of the pool of fittest individuals as selected in step 3a. This is done by copying the DNA of one individual up to the crossover point and the DNA of the other individual's DNA from there on. Mutation is applied during this recombination process by replacing a DNA element by a random number between 0 and (number of messages - 1) at a certain probability. This probability is called the mutation rate in the context of genetic algorithms. It must be high enough to enable sufficient exploration of the solution space but must not be too high as this would lead to too much

destruction of good DNA sequences. We found a mutation rate of about 1 percent to be a feasible compromise.

3. Results & Further Work

We tested the proposed genetic algorithm based scheduling with a generator that produces a random set of LET-based components which exchange data and are distributed to a set of nodes. The algorithm was able to come up with a solution in most cases. With a pool of 100 individuals it typically took only a few generations until at least one valid solution was found. After about 50 generations the fitness of the best individual increased to the level of the old, heuristic approach. However our algorithm had problems finding a solution when we increased the number of messages in the system as then the number of generations necessary to obtain at least one valid solution increased considerably. For an example with 50 messages it took almost 100 generations to encounter the first valid solution. However it is important to note that for the fitness calculation during these 100 generations only the fast fitness check was needed which is significantly less expensive than the one used to evaluate valid solutions. In such cases the number of valid solutions that pass the fast scheduling check is very small compared to the whole solution domain. This is due to our current DNA encoding which needs to be improved so that it restricts the solution domain so that the number of invalid solutions is minimized.

When comparing results of test cases with less than 40 messages with the results of the old, heuristic approach, we observed that we get similar solutions after less than 100 generations using a population size of 100 individuals. We consider this as evidence that the heuristics can indeed be replaced by the use of the proposed genetic algorithm and we are confident that this also holds for cases with more than 40 messages with an improved DNA representation.

Future work will focus on finding a better genetic representation and also on additional and systematic benchmarks to evaluate the quality of the genetic algorithm. Furthermore we will investigate the possibility of also including the node task schedules in the genetic algorithm. We are convinced that use of a genetic algorithm is a promising approach concerning the specific topic of communication scheduling for LET-based systems.

References

[1] J. Pletzer, W. Pree, J. Templ. A Code Generation Framework for Time-Triggered Real-Time Systems. Submitted and under review for Design, Automation & Test in Europe (DATE09). Included as appendix.

[2] E. Farcas, C. Farcas, W. Pree, J. Templ. Transparent Distribution of Real-Time Components Based on Logical Execution Time. LCTES, Chicago, Illinois, 2005.

[3] E. Farcas, W. Pree, J. Templ. Bus Scheduling for TDL Components. Dagstuhl Conference on Architecting Systems with Trustworthy Components, May 2006.

[4] E. Farcas, W. Pree. Hyperperiod Bus Scheduling and Optimizations for TDL Components. ETFA07, Patras, Greece.

[5] FlexRay communication protocol. http://www.flexray.com.

[6] Shan Ding, Naohiko Murakami, Hiroyuki Tomiyama, Hiroaki Takada. A GA-based scheduling method for FlexRay systems. EMSOFT 2005.