Hyperperiod Bus Scheduling and Optimizations for TDL Components

Emilia Farcas
Calit2
University of California, San Diego
efarcas@soe.ucsd.edu

Wolfgang Pree
C. Doppler Lab Embedded Software Systems
University of Salzburg, Austria
wolfgang.pree@cs.uni-salzburg.at

Abstract

The Timing Definition Language (TDL) provides a component model and a tool chain as a solution for building time-safe components that can be developed independently and integrated in a distributed platform without changing the observable behavior and the code of existing components. TDL is based on the Logical Execution Time abstraction and supports the decomposition of hard real-time applications into modules that are executed logically in parallel. This paper presents the algorithms for automatic schedule generation of TDL communications over the hyperperiod. As modules may switch modes independently, we combine the messages from all modes in the schedule. Hence, we introduce several optimizations to save bandwidth and improve the schedule's feasibility.

1. Introduction

Embedded software is often platform dependent and not compositional, especially from the timing perspective, leading to high costs for validation, integration, and maintenance. The behavior of a component depends on the overall system load and configuration, and timing properties result from the implementation without being explicitly specified at the design level. Even in model-based development approaches, the code generated from high-level specifications is later fine-tuned manually to meet the timing requirements for the target platform. Hence, the code and platform are tightly coupled, and the correspondence between the model and the final code is lost.

Therefore, embedded systems would benefit from a paradigm shift that ensures determinism, compositionality, and platform independence. AUTOSAR [1] aims at application-centric development of automotive software by decoupling functions from the underlying platform. We emphasize that not only software development, but also timing behavior should be platform independent. The explicit description of the timing behavior must be an integral part of the design and semantics of the software.

The Timing Definition Language (*TDL*) [16] is based on the *Logical Execution Time* (LET) abstraction, introduced in the realm of Giotto [7]. LET abstracts from the physical execution time and, thereby, from both the execution platform and the communication topology. *TDL* is a high-level description language for specifying the explicit timing requirements of a time-triggered application, which may be constructed out of several components. *TDL* allows modularization of applications, ECU consolidation, and the transparent distribution [3] of multi-mode real-time components.

With transparent distribution, a *TDL* application behaves exactly the same at run-time, no matter if all components are executed on a single node or if they are distributed across multiple nodes. The logical timing is always preserved; only the physical timing, which is not observable from the outside, may be changed. *TDL* components can be developed without having the execution on a potentially distributed platform in mind, as the distribution is visible only for the system integrator who specifies the mapping of components to computation nodes. Note that transparency applies not only to the functional but also to the temporal behavior of an application.

TDL transparent distribution allows independent component development and makes system integration easier, as the platform is considered after the components have been developed. The *TDL* tool chain frees the developer from the burden of explicitly specifying the communication requirements of components. Thus, *TDL* components can be moved to other nodes (if there are enough resources) without modifying their source code and without changing their behavior. *TDL* provides a complete tool chain for transparent distribution with a run-time system that enforces LET semantics and automatic generation of communications schedule and glue code.

This paper presents the automatic bus-schedule generation as an end-to-end process, starting with identifying the required messages for a set of components and obtaining in the end a table with the bus schedule for the target network topology. As we combine messages from different modes of execution, we use optimizations to allocate bandwidth efficiently. We introduce the algorithms for hyperperiod *TDL* scheduling with producer-consumer optimizations, dynamic multiplexing, and merging.

In the following, we start with an overview of the *TDL* component model. Then, we present each step of the bus-schedule generation. Section 4 presents measurements on the effects of various optimizations. An overview of related work and conclusions round out the paper.

1-4244-0826-1/07/$20.00 © 2007 IEEE

2. TDL Component Model

Our real-time component is the *TDL module*, which communicates with the physical environment through sensors and actuators, performs computation in tasks, and defines different operational modes that can be changed at run-time. A module is the unit of compilation and provides a namespace environment for declaring the language constructs (see [16] for the EBNF grammar).

LET defines the exact moments when a task exchanges data with the environment and other tasks. From the logical point of view, a task reads its inputs at the *release* event and runs continuously until its *termination* event; only then, its newly computed results are made available to the rest of the system. Thus, the observable temporal behavior of a task is independent from its physical execution (platform performance, communication topology, and scheduling policy). LET provides value and time determinism, compositionality, platform abstraction, and well-defined interaction semantics between parallel activities.

A module M can be only in one mode at a time. A mode $m \in \mathsf{Modes}[M]$ is a set of periodically executed activities, which can be task invocations, actuator updates, and mode switches: $\mathsf{Activities}[m] = \mathsf{Invokes}[m] \cup \mathsf{Updates}[m] \cup \mathsf{Switches}[m]$. A mode has a fixed period $T(m)$, and each activity a has its own rate of execution $\omega(a)$ within the mode. All activities are time triggered and can be executed conditionally (i.e., guarded by a Boolean function). *TDL* expresses only the timing behavior with LET semantics: when tasks read inputs and provide outputs, when actuators are updated, and when mode-switch conditions are checked. Data is communicated between entities through ports, and the functionality is specified as external functions in any imperative language.

A module $M \in \mathsf{Modules}$ may *import* other modules $M' \in \mathsf{Imports}[M] \subset \mathsf{Modules} \setminus \{M\}$ and may *export* some of its own program entities to other client modules by declaring them as publicly visible. Figure 1 sketches two sample *TDL* modules: *MPrd* with two modes containing two tasks each, and *MCns* with one mode and a task *Task2* that uses the output from *Task1*. The arrows depict the import relationship and the LET data-flow semantics, which are defined per mode.

Parallel tasks within a mode may depend on each other, i.e., the output of one task may be used as the input of another task. Moreover, a service provider module may export a task's outputs, which in turn may be imported by a client module and used as input for the client's computations. However, this data-flow dependency is defined by LET and it does not introduce precedence constraints for the release times of tasks. In the current *TDL* semantics, the LET of a task is equal to its period within the mode; the tasks invoked in a mode are a subset of the set $\mathsf{Tasks}[M]$ declared in this module.

When a module starts executing a mode m, all tasks invoked in m are released synchronously. *TDL* restricts mode switches to be *harmonic*: mode switches from mode m to mode m' may not interrupt task invocations (i.e., LET) in the old mode m. Thus, the period for evaluating the condition of any mode switch in m is a multiple of the LCM from the periods of all tasks invoked in m. A mode switch in a module is executed instantaneously.

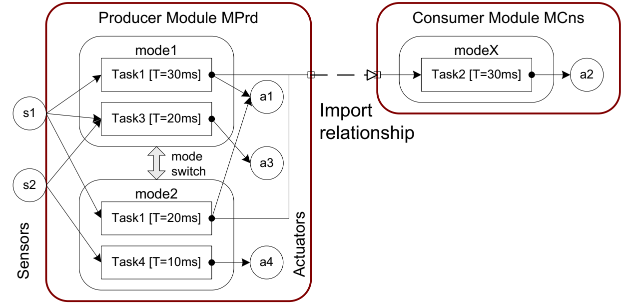

Figure 1. Visual representation of a module

The *TDL* component model solves the problem of ECU consolidation in the automotive domain by allowing multiple modules to run on the same ECU if there are enough resources available. All modules start the execution synchronously in their own start mode. Each module defines its own modes of operation and performs mode changes independent of other modules; this remains valid also when distributing modules. Thus, all modules execute logically in parallel: $M_1 \parallel M_2 \parallel \cdots \parallel M_{|\mathsf{Modules}|}$. *TDL* achieves parallel composition, as LET is always preserved: adding a new module will never affect the observable temporal behavior of other modules. It is the responsibility of the scheduler to guarantee conformance to LET.

3. Bus-Schedule Generation

Figure 2 shows the *TDL* tool chain. The compiler processes *TDL* source code and generates an abstract syntax tree (AST) representation of the *TDL* modules and the so-called ecode, which describes the LET semantics. The *TDL* run-time system [3] consists of a virtual machine for ecode execution, a scheduler, and a communication layer.

The plug-in architecture of the compiler allows its extension with any number of tools that rely on the AST. The Bus-Scheduler Plugin generates the bus schedule, based on a configuration file containing the list of computing nodes, the allocation of modules to nodes, and the physical properties of the network. For each target platform and module allocation, a Platform Plugin generates the glue code between the functionality, ecode, and run-time system and interfaces with a tool for WCET estimation.

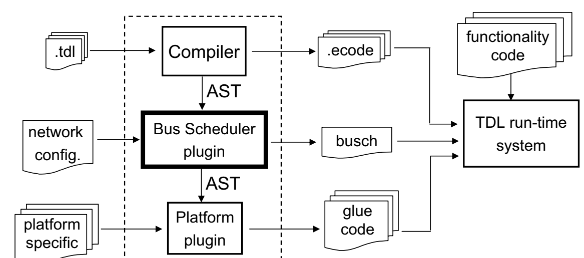

Figure 2. Overview of TDL tool chain

For transparent distribution [3], if a client module imports a producer module, the *TDL* run-time system creates an image of the producer module (i.e., a stub module) on the client's node. The client module will interact with this stub without knowing that the producer is actually on another node. The *TDL* run-time system synchronizes the stub with the producer itself based on the bus schedule, which must maintain the LET semantics.

3.1. Overview of Bus-Schedule Generation

We separate task and communication scheduling in two steps. The bus schedule is generated off-line as a description of the network activities within a bus period that is executed repeatedly. Tasks are scheduled on-line using EDF with timing constraints derived from the bus schedule.

We use broadcast communication on the bus. The access to the shared communication medium is collision free via a Time Division Multiple Access (TDMA) [11] approach. To support TDMA, we need a mechanism for clock synchronization over the network, which is implemented in the *TDL* communication layer if it is not available a priori on the platform (e.g., in our case studies [3] for CAN and RT-Ethernet). Our distribution approach is implemented on top of existing network protocols.

Furthermore, we use the Producer-Consumer model. The nodes that produce information trigger the sending of data over the network. The nodes that need the information do not send any requests, but just retrieve the data from the network. We use the terms *producer* and *consumer* only in respect to the data flow over the network.

The module is our unit of distribution. Modules are statically allocated to nodes, and more modules can be allocated to the same node. As a first step to fault tolerance, *TDL* supports module replication. The replicas are identified from the module-to-node assignment in the configuration file. We send the messages produced in all producer replicas and we process them in all consumer replicas.

The Bus-Schedule Generation Tool performs several steps (see Figure 3). It receives as input the *TDL* modules from the compiler and a configuration file for the target network topology and protocol. It provides as output the bus schedule with start and stop times for all frames.

Based on the allocation of modules to nodes, the Connection Identifier scans the *TDL* modules and identifies the producers, consumers, and data connections between them. Based on these connections, a *TDL* Distribution Protocol computes the bus period, identifies the required messages with their timing constraints, and binds messages to frames. A *message* represents the values of the task output ports produced by a task invocation. A *frame* represents the entire unit of information to be sent on the network: payload and control bits. The protocol maps n messages to $m \leq n$ frames, and currently does not support large messages, which should be split in several frames.

As each module switches its mode independently of other modules, we can create a global table with messages from all modes of all modules or a table for each possible

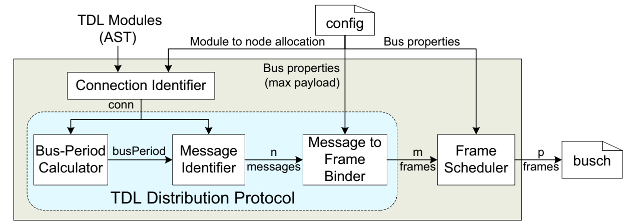

Figure 3. TDL Bus-Schedule Generation

combination of parallel modes - $\prod_M |\mathsf{Modes}[M]|$. As the latter leads to an explosion of tables, we use the first option. In this paper, we present a *TDL* scheduling approach over the *hyperperiod of task invocations*, where the schedule combines the messages from all modes. As modes in one module are exclusive, such a schedule wastes bandwidth. Thus, we introduce optimizations in the Producer-Consumer model (Section 3.4) and in the Frame Scheduler (Section 3.5), to reduce the number of frames. In TDMA, avoiding redundant messages at run-time is not helpful, as the time slots can not be reassigned to nodes. But we perform these optimizations off-line, when generating the schedule, as to improve the feasibility of the system.

The Frame Scheduler receives a set of frames with timing constraints and schedules the frames within the bus period, depending on the properties of the underlying network (e.g., protocol overhead, minimum and maximum payload size, gap bits, rate, and clock resolution). The Frame Scheduler performs multiplexing or merging between frames, leading to a reduced number of frames and to changes in the mapping between messages and frames. In the end, the bus schedule will contain $p \leq m$ data frames, plus the control frames. To maintain the LET semantics, the producer stub must know in which mode the producer is running. Thus, we allocate at the beginning of the bus period a control frame per each producer node containing the current modes of the producer modules.

Mode switches impose restrictions on the length of the bus period. As the execution of any mode must start synchronized with the bus cycle, a mode switch may not occur inside the bus cycle. We define $mspGCD_M$ for a module M as the GCD of mode periods and *mode-switch periods* in all modes of M. The mode-switch instants of M are positive multiples of $mspGCD_M$. Thus, for every module that communicates on the bus, $mspGCD_M$ must be a multiple of the bus period; this implies that $mspGCD_S$, the global GCD of $mspGCD_M$, is a multiple of the bus period. Thus, *TDL* hyperperiod scheduling can be used when the *TDL* modules define periods such as the LCM of task periods is a divisor of $mspGCD_S$. This is a common scenario, as modules typically have related mode periods and could even have the same mode-switch periods (recall that mode switches are harmonic within a module). In [4] we presented a Microperiod *TDL* Distribution Protocol that defines the bus period as equal to $mspGCD_S$ for arbitrary *TDL* modules. In this paper, we focus on the end-to-end process and the optimizations in all steps of hyperperiod scheduling.

3.2. Connection Identifier

For bus scheduling, we are interested only in the data connections that require communication on the network. Thus, the Connection Identifier reads the allocation of modules to nodes from the configuration file and scans the *TDL* modules to identify producers, consumers, and data connections between them. Each mode activity $(a, \omega, guard, \mathsf{SrcPorts}, \mathsf{DstPorts})$ defines a relative frequency, a guard function, a set of source ports, and a set of destination ports. By investigating the declarations of mode activities in all modes from all modules, we can determine which activities use remote source ports, i.e., ports from imported modules located on other nodes.

Consumers of data can be any mode activities: task invocations, actuator updates, and mode switches. Task invocations need source values for all task input ports, whereas actuator updates need one source value that will be used to set the physical actuator. Mode switches can be consumers because they may initialize task output ports in the target mode with remote values. Moreover, the guard of any activity may also use remote ports in its arguments.

Producers of data can be only task output ports. Sensors can also be used as input to other entities; however, reading a local sensor takes logical zero time, but sending over the network a message with its value takes a non-negligible time. Thus, to maintain the transparent distribution, we do not allow direct references to remote sensors. Sensors can be accessed via task wrappers, which define the LET and reading rate for the sensors.

We introduce Algorithm 1 for identifying the producer-consumer data connections from a set of *TDL* modules. For each producer task τ, we select only its output ports that are required by at least one consumer, obtaining the set of producer ports $\mathsf{ProdOutPorts}[\tau] \subset \mathsf{OutPorts}[\tau]$. We also create a *data connection* from each producer task to all its consumers. Data may be produced, respectively read, with different frequencies in different modes. Thus, a connection $dc = (\tau, PrdPeriods, CnsPeriods)$ has associated the set of periods $PrdPeriods$ of invoking τ in all modes and the set of periods $CnsPeriods$ of reading the producer ports in all consumers in all modes. We use the notation $X\cup \leftarrow x$ for adding an element to a set, and $x \mid p$ for identifying elements that satisfy a property.

Algorithm 1. Identify data connections

```
identifyConnections(Set Modules) returns Set
  dataConnections ← ∅
  ∀M ∈ Modules :
    // identify connections from other modules to this module
    ∀m ∈ Modes[M] :
      //check all mode activities
      ∀(a, ω, guard, SrcPorts, DstPorts) ∈ Activities[m] :
        // check all source ports for this activity and its guard
        ∀s ∈ SrcPorts[a] ∪ SrcPorts[guard] :
          // identify all remote replicas of the producer module
          ∀PrdM ∈ Imports[M] | s ∈ PrdM :
            if node(M) ≠ node(PrdM) then
              select τ ∈ Tasks[PrdM] | s ∈ OutPorts[τ]
              ProdOutPorts[τ]∪ ← s
              if ∄dc(τ) ∈ dataConnections then
                dc ← new DataConnection(τ)
                // get all periods for updating s in all modes
                ∀pm ∈ Modes[PrdM] :
                  if τ ∈ Invokes[pm] then
                    PrdPeriods[dc]∪ ← T(pm)/ω(τ, pm)
                  endif
                dataConnections∪ ← dc
              endif
              // add the period of the current consumer
              CnsPeriods[dc]∪ ← T(m)/ω(a)
            endif
  return dataConnections
```

As *TDL* supports module replication, we check each replica separately to identify the connections. Depending on the allocation to nodes, the replicas of the same producer can have different consumers requiring communication on the network. Thus, we create an independent connection for each producer replica. For instance, module *M1* is replicated on three nodes *N1*, *N2*, and *N3*; module *M2* imports *M1* and runs on *N3*. We create connections only for producer tasks in the first two replicas of *M1*.

3.3. Basic Producer-Consumer Model

The straightforward approach to implement the Producer-Consumer model is to ignore the knowledge about the periods of consumers. We need the Connection Identifier only to filter task output ports to those that have at least one consumer. However, we communicate messages each time a new value is produced, regardless of whether a consumer instance uses that particular value. Thus, we send messages with the frequency of producers.

We compute the bus period as the LCM of producer task periods in all modes and all modules:

$$busPeriod = \operatorname*{lcm}_{\forall dc}\{T \mid T \in PrdPeriods[dc]\}$$

We introduce Algorithm 2 for identifying messages and frames from a set of connections. We create one message for each instance of a producer task τ within each mode of operation; the message gathers the values of all producer ports $\mathsf{ProdOutPorts}[\tau]$. Thus, the number of messages is given by the bus period divided with the period of the producer $TPrd$ in the analyzed mode. The release r and deadline d constraints for a message depend on the worst-case execution time $wcet$ and the LET instants for the producer-task instance. The size of the message is fixed, summing the size of the producer ports.

Algorithm 2. Basic Prod-Cons model

```
createMessagesFrames(Set connections) returns Set
  frames ← ∅
  ∀dc ∈ connections:
    ∀TPrd ∈ PrdPeriods[dc]:
      //identify all task invocations within the bus period
      ∀invPrd ∈ ℕ = 1, ··· , busPeriod/TPrd:
        msg ← createMessage(τ(dc), invPrd, TPrd)
        addMessage(msg, frames)
  return frames

//creates a message for an invocation of a producer task
createMessage(Task τ, inv ∈ ℕ, T ∈ ℕ) returns Message
  r ← (inv − 1) · T + wcet(τ)  //msg release
  d ← inv · T                  //msg deadline
```

return new Message(τ, r, d)

//adds a message to an existing frame of the same producer with
//the same deadline or creates a new frame for the message
addMessage(Message m, Set $frames$)
  if $\exists f \in frames \mid d(m) = d(f) \wedge (\forall m' \in f, \tau(m) = \tau(m'))$
  then
    $r(f) \leftarrow \max(r(f), r(m))$ *//keep the max release time*
    $f \cup \leftarrow m$ *//add the message to the frame*
  else
    $f \leftarrow$ new Frame(m) *//copy m constraints (r,d,size) to f*
    $frames \cup \leftarrow f$
  endif

When a producer task has invocations with the same deadline in different modes, it is redundant to schedule all messages because modes are exclusive. Thus, we allocate all messages from such producer instances to only one frame, whose release constraint is the maximum release among all messages. This allocation strategy is a simple case of multiplexing. The size of the frame is the size of *one* message.

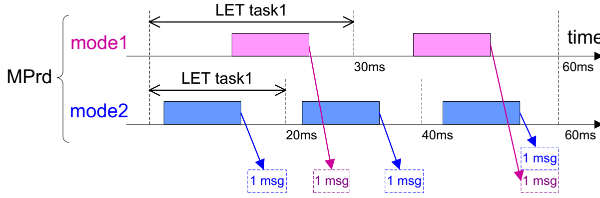

Figure 4. The messages and frames for a producer invoked in different modes

Figure 4 presents as example *task1* invoked in two modes of module *MPrd* with periods of $30ms$, respectively $20ms$. Within the bus period of $60ms$, there are five messages in total, but only four frames because the last producer invocation has the same deadline in both modes.

3.4. Optimized Producer-Consumer Model

We optimize the Producer-Consumer model by considering only the messages that are actually used by at least a consumer in at least one mode of operation. When the consumers have a lower frequency than the producer, an obvious optimization is to suppress unused messages. Therefore, this optimization saves network bandwidth.

The difference to the basic model from Section 3.3 is that we must consider the relation between the periods of producers and consumers. If there exists one consumer with a smaller period than the producer, it will consume all results produced; thus, we cannot reduce the number of messages for this producer in the given mode. In general, there can be data connections that can be optimized, whereas others can not.

The bus period becomes equal to the LCM of (1) producer periods for all data connections dc and (2) consumer periods only for connections that can be optimized:

$$busPeriod = \operatorname*{lcm}_{\forall dc}\{PrdPeriods[dc]\} \cup \{CnsPeriods[dc] \mid \min(CnsPeriods[dc]) > \min(PrdPeriods[dc])\}$$

We may obtain a larger bus schedule than in the basic model, because we consider also consumer periods. However, this schedule better utilizes the bandwidth.

We introduce Algorithm 3 for identifying the required messages and frames. When a consumer runs slower than the producer, for each consumer instance we identify the producer instance with the closest deadline to the release of the consumer. We create a corresponding message with release and deadline determined by this producer instance, only if the message was not already requested by other consumers. Although different consumers may require different output ports from tasks, we send all producer values to all consumers so that to maintain producer consistency. From this reason, we created a data connection per producer task and not per producer port.

Algorithm 3. Optimized Prod-Cons model

createMessagesFrames(Set $connections$) **returns** Set
  $frames \leftarrow \emptyset$
  $\forall dc \in connections$:
    $\forall TPrd \in PrdPeriods[dc]$:
      if $\min(CnsPeriods[dc]) \leq TPrd$ **then**
        //create messages for $TPrd$ as in basic $Prd-Cns$ model
        $\forall invPrd \in \mathbb{N} = 1, \cdots, busPeriod/TPrd$:
          $msg \leftarrow$ createMessage($\tau(dc), invPrd, TPrd$)
          addMessage($msg, frames$)
      else
        $\forall TCns \in CnsPeriods[dc]$:
          $\forall invCns \in \mathbb{N} = 1, \cdots, busPeriod/TCns$:
            //identify the last invocation of the producer
            $invPrd \leftarrow invCns \cdot TCns/TPrd$
            if $\nexists msg(\tau(dc), invPrd, TPrd)$ **then**
              $msg \leftarrow$ createMessage($\tau(dc), invPrd, TPrd$)
              addMessage($msg, frames$)
            endif
      endif
  return $frames$

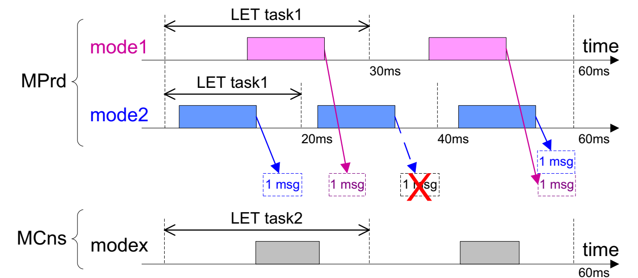

Figure 5. Optimizing messages

Figure 5 reviews the example from Section 3.3, with a consumer *task2* invoked in one mode of module *MCns* with period $30ms$. In the optimized model, we create four messages and three frames. When *task1* runs in *mode2*, the result of its second invocation is not used by *task2*, because the second invocation of *task2* reads the inputs at its release of $30ms$ as specified by LET semantics, thus, requiring the result of the first invocation of *task1*.

3.5. Frame Scheduler

The Frame Scheduler receives as input a set of frames from the basic or optimized Producer-Consumer model. Each frame has a payload size and timing constraints com-

puted from the messages bound to the frame. The worst-case transmission time is computed from the frame size and the network properties. The *release* and *deadline* of the frame define a communication window. As the communication windows of different frames could overlap, we schedule them with a variation of the Reversed EDF algorithm, or the Latest Release Time [12]. When scheduling a frame, we assign it the transmission *start* and *stop* times.

As producer tasks are scheduled on-line with deadlines at the start of the frame containing the produced message, our strategy is to schedule frames as late as possible, to allow more flexibility for task scheduling. Thus, we sort the set of frames by deadlines and then by release times; we schedule the ordered frames non-preemptively starting from the end of the bus period and going backwards. We do not restrict the TDMA pattern inside a cycle, as frames sent by different nodes may alternate in the schedule and may have different sizes. We used this approach in our cases studies with a time-triggered adaptation of CAN [9] and RT-Ethernet.

The Frame Scheduler has constraints from the physical properties of the communication infrastructure. For example, it aligns the start time according to the inter-frame gap and the clock resolution on the nodes. It also generates control frames, for time synchronization and for communicating the active modes. Remember that a producer can change the mode only at the end of the bus cycle.

We introduce Algorithm 4 for scheduling a set of *TDL* frames with multiplexing and merging between frames. We define *multiplexing* as the allocation to the same frame of messages from different modes of operations of the same module; these messages will not be sent in the frame in the same bus cycle but in *different bus cycles* depending on the current mode. We define *merging* as the allocation to the same frame of messages that will be sent together in *the same bus cycle*. Both optimizations reduce the number of frames, but multiplexing also reduces the payload size.

Recall from Sections 3.3 and 3.4 that a frame had messages from the same producer task in different modes. After scheduling, a frame can have several multiplexed messages from different producer tasks, and it can also have merged messages, even from different modules. The bus schedule is static in the sense that it is predefined when a frame is sent and which messages are allocated to the frame. But *the structure of a frame changes depending on the active modes at run-time*. The schedule specifies statically the size of the frame such that, at run-time, the messages from any modes will fit in the frame.

Before scheduling a frame f, the Frame Scheduler tries first to multiplex it with a previously scheduled frame pf. As merging is not considered at this point, the Frame Scheduler identifies the messages from pf having the same producer module as the messages from f. If the two sets of messages correspond to producer tasks updated in different modes, they are exclusive at run-time. Thus, they can reuse the same frame in the schedule. However, we can reallocate the messages from f to pf, only if the timing constraints of all messages are met: $start(pf) \geq r(msg)$ and $stop(pf) \leq d(msg)$.

Algorithm 4. Multiplexing and merging

```
scheduleFrames(Set frames)
  sortRevEDF(frames); //sort backwards by deadline, then release
  endTime ← busPeriod
  ∀f ∈ frames:
    muxed ← merged ← false
    prevFrames ← previouslyScheduledFrames(frames, f)
    while pf ∈ prevFrames and not muxed:
      muxed ← tryMultiplexing(f, pf)
    if not muxed then
      while pf ∈ prevFrames and not merged:
        merged ← tryMerging(f, pf)
    endif
    if muxed or merged then
      delete f
    else
      scheduleFrame(f, endTime)
      endTime ← start(f) − gapTime
    endif
  frames∪ ← controlFrames   //add and schedule control frames

tryMultiplexing(Frame f, Frame pf) returns bool
  if node(f) ≠ node(pf) or d(f) < stop(pf)
    or module(f) ∉ pf then return false
  ∀msg ∈ f:
    if mode(msg) ∈ pf then return false
  newSize ← max(size(f), size(pf))
  if not checkTiming(pf, f, newSize) then return false
  reallocateMessages(msg(f), pf)
  return true

tryMerging(Frame f, Frame pf) returns bool
  if node(f) ≠ node(pf) or d(f) < stop(pf) then return false
  if we combine merging with multiplexing then
    //get the size needed by the producer module in the previous frame
    moduleSize ← sizePerModule(pf, module(f))
    ∀msg ∈ f:
      modeSize ← size(msg)+ sizePerMode(pf, mode(msg))
    overhead ← max{modeSize} − moduleSize
    newSize ← max(size(f), size(f) + overhead)
  else
    newSize ← size(f) + size(pf)
  endif
  if newSize > maxPayload return false
  if not checkTiming(pf, f, newSize) then return false
  reallocateMessages(msg(f), pf)
  return true
```

When we multiplex frames, we delete f and the new size of the frame pf is not the sum but the *maximum* between the sizes of pf and f. This could also lead to an enlargement of pf, which could change its transmission time in systems with high clock resolution. As pf could be any of the previously scheduled frames, this change could require the shifting of all frames between f and pf. We have chosen not to perform multiplexing (nor merging) if shifting is required.

If pure multiplexing fails, the Frame Scheduler tries to merge frames if they are sent by the same node, even from different modules. In pure merging, the size of f is simply added to the size of pf. In merging combined with multiplexing, each message from f is merged with the messages from pf from the same mode in the same producer module. Thus, we change the size required by some producer modes in pf, which may increase also the

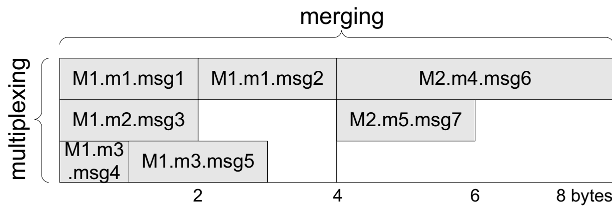

Figure 6. Multiplexing & merging in a frame

size required by the producer module.

Figure 6 presents an example of seven messages allocated to *one frame* of 8 bytes: there are 4 bytes reserved for each merged module, $M1$ and $M2$. The rectangle of a message shows its size in bytes. In each module, there are messages multiplexed from different modes; $M1$ has also messages merged within mode $m1$ and $m3$. For instance, at run-time when $M1$ runs in $m3$ and $M2$ runs in $m5$, the frame will contain messages $msg4$, $msg5$, and $msg7$.

4. Evaluation

In the following, we present how Producer-Consumer optimizations, multiplexing, and merging frames affect the creation of the bus schedule. As we have automatic bus-schedule generation, we can perform numerical experiments on sets of *TDL* modules as input. Thus, we implemented a tool that creates random *TDL* modules.

We started with a single-node system and added one node in each iteration up to 25 nodes; we allocated two modules to each node. Each module had five modes, with each mode period as $24ms$ and the mode-switch frequencies as 1, resulting in a bus period of $24ms$. Each module has a random number of producer and consumer tasks less than five, where consumers use remote values uniformly from all other modules in the system. Each task within a mode has a random period among the divisors of the mode period. Tasks are assigned randomly to modes, but each task is invoked in at least two modes. Each task has a port of 4 bytes, to simplify the import relationships and avoid type mismatches.

We performed experiments for the CAN network, and the properties are provided in a configuration file. CAN has a maximum payload of 8 bytes and a minimum interframe gap of 3 bits. It has 44 envelope bits, but due to bit stuffing, the overhead of the frame varies with the message content and is maximum 68 bits. We assumed a bus rate of $1Mbit/s$ and a clock resolution of $200\mu s$ on the nodes, which reduces the bandwidth we can use. Within a bus cycle of $24ms$, we can send maximum 120 frames.

Figure 7 presents the total number of messages for the basic and optimized Producer-Consumer model. As the number of nodes grows, so does the number of producers and consumers in the system, which require a higher number of messages except when the new consumers use the same messages as the others (see the 15th node). The optimized model (Sect. 3.4) avoids creating messages for values produced but not required by any consumers; for random periods, it reduces the number of messages with

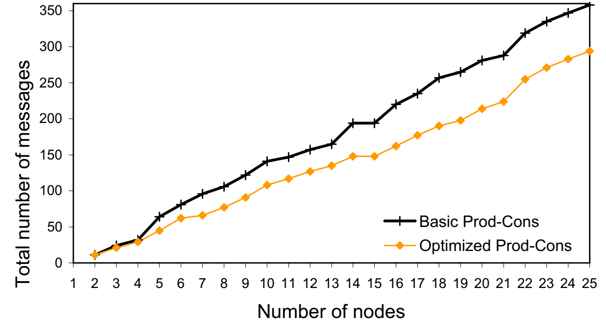

Figure 7. Producer-Consumer optimizations

23% in the average case.

Figure 8 presents the total number of scheduled frames and the data throughput when enabling or not multiplexing and merging in Frame Scheduler. The X axis shows the number of messages identified in the basic model. Recall that messages from the same producers are multiplexed from the beginning, thus the number of frames is reduced even if not optimizing in Frame Scheduler (see Basic PC line: 147 messages use 112 frames). Adding optimizations further reduces the number of frames and, more importantly, increases the schedulability of the system; each line stops when the schedule becomes unfeasible. We can schedule up to 320 messages within 112 frames, and the average gain of all optimizations is 50%.

Figure 8 also shows that merging reduces the frames more than multiplexing, as the message size is 4 bytes allowing merging, whereas tasks are invoked only in two modes limiting multiplexing. However, merging does not change the data throughput, but only the data efficiency by reducing the frame overhead. Instead, multiplexing reduces the data throughput as we reserve less space for messages sent in different cycles. Thus, multiplexing reduces the bandwidth consumption of a set of messages.

Furthermore, our strategy is to schedule frames as late

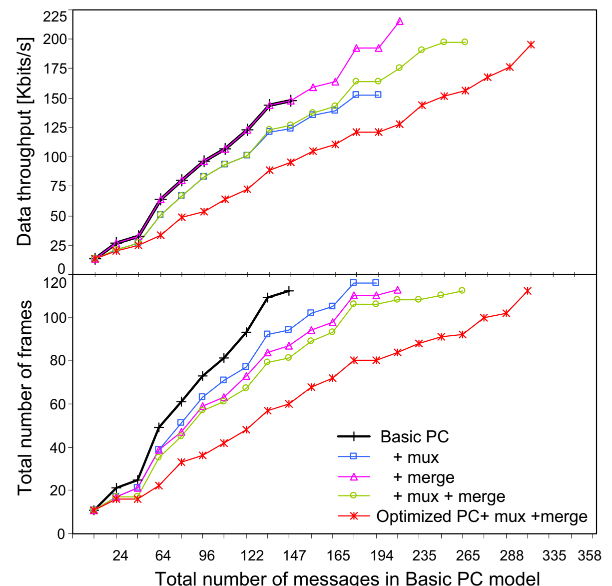

Figure 8. Frame Scheduler optimizations

as possible to allow more time for the task computation. Merging and multiplexing reduce the average slack of messages (message deadline − frame stop), as they are allocated to frames scheduled later in the bus period. The charts for the effect of the optimizations on the slack are in correspondence with the charts for the number of frames.

5. Discussion and Related Work

TTTech [17] and its subsidiary TTAutomotive support the development of time-triggered automotive systems, based on TTP [18] and FlexRay [5] protocols. FlexRay is also supported by DECOMSYS's [2] Designer Pro and Vector's [19] DaVinci tool suites. Their tools for schedule generation require as input the set of messages and their timing constraints, whereas we automatically detect them from the *TDL* modules. *TDL* also provides tools that map *TDL* modules to the TTTech and Designer Pro tools.

Our Frame Scheduler can also be easily adapted for TTCAN [10], TTP, and FlexRay, by considering their restrictions on the structure of TDMA cycles. It can also incorporate multiplexing and merging within other scheduling algorithms than Reverse EDF (e.g., heuristic search), which could further optimize feasibility and bandwidth. [15] also addresses the problem of frame packing as to minimize the bandwidth consumption for CAN and to choose the priorities for frames. Frame packing is similar to what we call merging; their algorithms are based on greedy or bin packing and fixed-priority scheduling, whereas we address TDMA and combine merging with multiplexing for LET semantics. We also focus on minimizing the slack (by scheduling frames as late as possible) and then optimizing the bandwidth. Note that in our case, frames are not periodic inside the bus cycle.

We schedule tasks and messages in two steps, which is also the case of the [17, 2, 19] tools. This leads to suboptimal solutions, as there is a tradeoff between the two, but the bus-schedule generation could be tuned on the feedback from the schedulability analysis for tasks. Algorithms for scheduling clock-driven tasks and messages in one step (e.g., [6]) are impractical in our case because of independent mode switches.

Fixed-time partitioning between modules for processors and the network (e.g., virtual networks [14]) offers compositionality, but can waste system resources, leading to infeasibility. Server-based scheduling [13] is a more flexible and fair alternative for hierarchical scheduling, but it needs a timing analysis. These strategies need to be further evaluated for multi-mode LET systems like *TDL*.

Related work on mode changes addresses only global modes, which simplify the scheduling problem. [6] also proposes faster mode changes than at the end of the cycle, via a transition schedule. Giotto [7] saw the benefits of LET for distributed systems, but its prototype implementations have limited support for distribution, where the schedules are constructed mostly manually and do not support mode switches [8].

6. Conclusions

TDL supports parallel composition of real-time components in respect to both value and time determinism. It becomes possible to change the underlying platform and to distribute components without affecting the overall system behavior and without changing the code of components, because we identify and schedule automatically all required communications on the network.

We showed how producer-consumer optimizations, dynamic multiplexing, and merging improve the feasibility and reduce the bandwidth consumption. Producer-consumer optimizations and multiplexing reduce the payload, whereas merging reduces only the frame overhead. Merging is useful in systems with low clock resolution, when more messages have a small size and fit in a frame.

References

[1] *Automotive Open System Architecture*. www.autosar.org.

[2] DECOMSYS. www.decomsys.com.

[3] E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent distribution of real-time components based on logical execution time. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 31–39, June 2005.

[4] E. Farcas, W. Pree, and J. Templ. Bus scheduling for TDL components. In *Architecting Systems with Trustworthy Components*, LNCS 3938, pages 71–83. Springer, 2006.

[5] FlexRay Consortium. *FlexRay Communications System Protocol Specification, Version 2.0*, June 2004.

[6] G. Fohler. *Flexibility in Statically Scheduled Hard Real-Time Systems*. PhD thesis, Technische Univ. Wien, 1994.

[7] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proc. EMSOFT*, pages 166–184, 2001.

[8] T. A. Henzinger, C. M. Kirsch, and S. Matic. Composable code generation for distributed Giotto. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.

[9] ISO IS 11898. *Road vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High-Speed Communication*, 1993.

[10] ISO IS 11898-4. *Road vehicles - Controller area network (CAN) - Part 4: Time-triggered communication*, 2004.

[11] H. Kopetz. *Real-time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, 1997.

[12] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, 2000.

[13] T. Nolte. *Share-Driven Scheduling of Embedded Networks*. PhD thesis, Mälardalen University, May 2006.

[14] R. Obermaisser, P. Peti, and H. Kopetz. Virtual networks in an integrated time-triggered architecture. In *Proc. WORDS*, pages 241 – 253, Feb. 2005.

[15] R. Saket and N. Navet. Frame packing algorithms for automotive applications. *Journal of Embedded Computing*, 2(1):93–102, 2006.

[16] J. Templ. TDL Specification and Report. Technical report, University of Salzburg, Austria. Mar 2004, www.SoftwareResearch.net/site/publications/C059.pdf.

[17] *TTTech Computertechnik AG*. www.tttech.com.

[18] TTTech. *Time-Triggered Protocol TTP/C High-Level Specification Document, Edition 1.4.3*, Nov. 2003.

[19] *Vector Informatik GmbH*. www.vector-informatik.com.