Optimized static real-time scheduling of communications on a broadcast bus

Dumitru Potop-Butucaru, Yves Sorel

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Optimized static real-time scheduling of communications on a broadcast bus

Dumitru Potop-Butucaru — Yves Sorel

N° 6655

Septembre 2008

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6655--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

centre de recherche
PARIS - ROCQUENCOURT

Optimized static real-time scheduling of communications on a broadcast bus

Dumitru Potop-Butucaru , Yves Sorel

Thème COM — Systèmes communicants
Équipe-Projet AOSTE

Rapport de recherche n° 6655 — Septembre 2008 — 18 pages

Abstract: We consider the problem of minimizing bus usage for static real-time scheduling of hierarchical dataflow specifications involving conditional execution. Statically scheduling conditional communications over an asynchronous broadcast bus involves the sending of the activation conditions themselves, which allow all processors to know which messages they must throw away or use. As the communication of an activation condition may be hierarchically conditioned itself, this results in a complex calculus of activation conditions (also called logical clocks in some settings). We provide a technique that uses this calculus to ensure that no piece of information is sent twice over the bus. Our technique can be used to reduce a given static schedule to a normal form with no redundant communication. It can also be incorporated into existing scheduling algorithms to ensure by construction the absence of redundancy. The technique can also be used to reduce communication when some form of time synchronization is used (e.g. on time-triggered buses), but some optimality properties may be lost.

Key-words: real-time scheduling, optimization, SynDEx, hierarchical dataflow, conditional execution, bus-based architecture, clock calculus

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Optimisation d'ordonnancements statiques temps-reel de communications sur un bus broadcast

Résumé : On s'interesse au problème d'optimisation de l'utilisation du bus pendant l'ordonnancement de spécifications dataflow hiérarchiques contenant de l'exécution conditionnelle.

Mots-clés : ordonnancement temps-reel, optimisation, SynDEx, flot de donnees hierarchique, execution conditionnelle, architecture base de bus, calcul d'horloges

1 Introduction

In fields such as avionics, automotive, and robotics, *hard real-time embedded systems* are often designed as automatic control systems. By consequence, their functional specification is often done in a *conditioned dataflow formalism* such as Simulink [2] or Scade [1, 8]. One main characteristic of these formalisms is that they go beyond the classical dataflow model [3] by introducing a form of *conditional execution* allowing the hierarchical description of execution modes. Thus, each dataflow block is hierarchically assigned an *activation condition* which governs its execution.

Upon distributed implementation, the information transmitted through the communication media must allow not only the computation of the activation condition of each dataflow block, but also the reliable communication between blocks. To complicate things, communications on the bus must be statically scheduled (modulo conditional communication) to ensure the temporal predictability of the system. At the same time, communication time must not explode, so that the real-time implementation can meet its deadlines. Balancing between these two objectives – functional correctness and communication efficiency – is necessarily based on the *fine manipulation of both time and activation conditions*, which is our objective in this paper.

Contribution. Our paper adresses these issues in the case of implementation architectures formed of several processors (of various types) connected to a unique broadcast bus. On this type of architecture, we consider a less typical problem: That of *optimizing given schedules*. Instead of proposing a new distributed scheduling heuristic, we seek to define schedule optimality properties that should be ensured by large classes of heuristics, but are easily missed due to the complexity of dealing with activation conditions. We are most interested in properties that allow the definition of low-complexity techniques ensuring the optimality property by transformation of given schedules. Such transformations can be used in conjunction with existing heuristics to improve their result.

Our first step in this direction is the definition of a *formal model of real-time schedule of dataflow specification with conditional execution*. The formalism includes the definition of a calculus allowing the *manipulation of activation conditions* through objects named *clocks*. Our new formalism is general enough to allow the representation of the output of real-life scheduling techniques such as AAA/SynDEx [5].

The new formalism allows us to state the two optimality properties, whose expression is difficult in the presence of complex activation conditions:

- The absence of redundant communications (a value is never sent twice on a bus during the execution).
- The worst-case execution time of the real-time schedule is the lowest possible with respect to the chosen schedule representation allowing implementation under the tightest deadlines.

For both properties, we provide low-complexity algorithms allowing the transformation of a given schedule to satisfy the optimality properties. These transformations preserve the correctness of the schedule and can be combined to ensure both properties at a time. The transformations can also be integrated into existing scheduling algorithms.

We thus provide a non-trivial optimization algorithm that can be readily used in embedded system development, and a model that can serve as base for the development of further optimizations.

Outline. The remainder of the paper is organized as follows. Section 2 intuitively presents our bus-based architecture model, and Section 3 gives our model of hierarchic conditioned dataflow. Section 4 is about activation conditions. It introduces the fundamental notions of clocks, signals, sampling, and computable clock. Clocks are used in Section 5 to bring the hierarchic dataflow to a flattened form that is more common as input of scheduling algorithms. Section 6 defines static schedules, and the correctness properties ensuring correct implementation of a dataflow by a schedule. The actual optimization results arrive in Section 7, which formally defines the desired optimality property, defines local optimization steps, and proves that any maximal sequence of such steps leads to a schedule that is optimal in some sense. We review related work in Section 8, and conclude in Section 9 with an emphasis on future work.

2 The architecture

The architectures we consider in this paper are formed of a unique broadcast bus denoted $\mathcal{B}$ that connects a set of processors $\mathcal{P} = \{P_i \mid i = \overline{1..n}\}$. The architecture is connected with the outside world through input and output FIFO *channels*. We denote with $\mathcal{I}(P)$ the finite set of input lines arriving on processor $P \in \mathcal{P}$ and with $\mathcal{O}(P)$ the finite set of output lines leaving P. We assume that each input arrives on exactly one processor, and that each output leaves exactly one processor.

All communication lines are assumed reliable (no message losses, no duplications, no data corruption). In this paper, we assume all communication and synchronization is only done through asynchronous message passing.[1] No form of global or local time is used to control execution (*e.g.*, through timeouts).

The formal model of our execution mechanism will be given under the form of constraints on the possible schedules in Section 6. We give in Appendix A.1 the intuition behind these constraints, under the form of one complying execution mechanism.

3 Dataflow with conditional execution

Our technique has evolved from needs in the development of the SynDEx optimized distributed real-time implementation tool [5]. Hence, our presentation is based on the dataflow formalism of SynDEx, which we define next. However, the results of the paper can be generalized to other hierarchical conditioned dataflow models used in embedded systems design, such as Lustre/Scade or Simulink.

[1]CAN [10] is a classical bus satisfying these hypotheses. However, the optimization technique we develop can be extended (with weaker results) to time-triggered buses such as TTA [10] and FlexRay.

3.1 Syntax

In SynDEx, functional specification is realized using a *synchronous* formalism very similar to Scade/Lustre [8]. A specification, also called *algorithm* in SynDEx jargon, is a *synchronous hierarchic dataflow diagram with conditional execution*. It is formed of a hierarchy of dataflow *nodes*, also called *operators* in SynDEx jargon. Each node has a finite set of named and typed input and output *ports*. We respectively denote with $\mathcal{I}(n)$ and $\mathcal{O}(n)$ the sets of input and output ports of node n. To each input or output port p we associate its type (or domain) $\mathcal{D}_p$.

An algorithm is a hierarchic description, where hierarchic decomposition is defined by means of *dataflow graphs*. A dataflow graph is a pair $\mathcal{G} = (\mathcal{N}_\mathcal{G}, \mathcal{A}_\mathcal{G})$ where $\mathcal{N}_\mathcal{G}$ is a finite set of nodes and $\mathcal{A}_\mathcal{G}$ is a subset of $(\bigcup_{n\in\mathcal{N}_\mathcal{G}} \mathcal{O}(n)) \times (\bigcup_{n\in\mathcal{N}_\mathcal{G}} \mathcal{I}(n))$ satisfying two consistency properties:

Domain consistency: For all $(o, i) \in \mathcal{A}_\mathcal{G}$, we have $\mathcal{D}_o = \mathcal{D}_i$

Static single assignment: For all $i \in \bigcup_{n\in\mathcal{N}_\mathcal{G}} \mathcal{I}(n)$ there exists a unique o such that $(o, i) \in \mathcal{A}_\mathcal{G}$.

The nodes are divided into *basic nodes*, which are the leaves of the hierarchy tree (the elementary dataflow operators), and *composed nodes*, which are formed by composition of other nodes (basic and composed). There are two types of basic nodes: *dataflow functions*, which represent elementary dataflow computations, and *delays*, which are the state elements. Each delay d has exactly one input and one output port of the same type, denoted $\mathcal{D}_d$, and also has an initializing value $d_0 \in \mathcal{D}_d$.

Each composed node has one or more *expansions*, which are dataflow graphs. We denote with $\mathcal{E}(n)$ the set of expansions of node n. The expansion(s) of a composed node define its possible behaviors. Composed nodes with more than one expansion are called *conditioned nodes* and they need to choose between the possible dataflow expansions at execution time. This choice is done based on the values of certain input ports called *condition input ports*. We denote with $\mathcal{C}(n) \subseteq \mathcal{I}(n)$ the set of condition ports of a conditioned node n. The association of expansions to condition port valuations is formally described using a partial function:

$$cond_n : \prod_{i\in\mathcal{C}(n)} \mathcal{D}_i \to \mathcal{E}(n)$$

The function needs not be complete, because all input valuations are not always feasible. However, the function must be defined for all feasible combinations of conditioning inputs. We assume this has already been checked.

An expansion $\mathcal{G} = (\mathcal{N}_\mathcal{G}, \mathcal{A}_\mathcal{G})$ of a node n must satisfy the following consistency properties:

- There exists an injective function from $\mathcal{I}(n)$ to $\mathcal{N}_\mathcal{G}$ associating to each port i the basic node $i_\mathcal{G}$ having no input ports and one output port of domain $\mathcal{D}_i$. We call $i_\mathcal{G}$ an *input node*.

- There exists an injective function from $\mathcal{O}(n)$ to $\mathcal{N}_\mathcal{G}$ associating to each port o the basic node $o_\mathcal{G}$ having no output ports and one input port of domain $\mathcal{D}_o$. We call $o_\mathcal{G}$ an *output node*.

We assume the hierarchy of nodes is complete, the algorithm being the topmost node. To simplify notations, we assume the algorithm is used in no expansion, and that each other node is used in exactly one expansion. We denote with $\mathcal{N}_n$ the set of nodes used in the hierarchic description of n.

An example of SynDEx specification in an intuitive graphical form is given in Appendix A.2.

3.2 Operational semantics

The operational semantics is the classical cycle-based synchronous execution model which is also used in Scade/Lustre. The execution of a node (the algorithm included) is an infinite sequence of *execution instants* where the node reads all its inputs and computes all its outputs. The behavior of a node depends on its type:

- The computation of a *dataflow function* is *atomic*: All inputs are waited for and read, then the non-interruptible computation is performed, and then the outputs are all produced.
- The *delays* are state elements. When executed in an instant, a delay delivers the value stored in its previous execution instant. In the first execution instant, no previous value is available, and the delay d delivers the initializing value d_0. Then, it waits and reads its input, and concludes the execution of the instant by storing this new value for the next instant.
- If a *composed node* has condition input ports, then its execution starts by waiting and reading the values of the conditioning inputs. These values are used to choose one expansion, and execution proceeds as though the node were replaced by its expansion. In particular, no atomicity is required and two nodes can be executed in parallel if no dependency exists between them.

As mentioned above, it is assumed that any possible combination of conditioning inputs of a node n is assigned an expansion through $cond_n$. This amounts to a partition of the possible configurations of the conditioning inputs among expansions. This requirement ensures an important correctness property: The fact that any every value that is read in an execution instant has been produced in that instant (conditional execution never leads to the use of uninitialized data).

4 Clocks

The execution of an algorithm being a discrete sequence of execution instants, the sequence of instants where a node n is executed is naturally described with a function from non-negative integers (the execution instant indices) to Booleans (1 for execution, 0 for non-execution) $clk(n) : \mathbb{N} \rightarrow \mathbb{B}$. More generally, we use such functions to represent the sequence of instants where some event takes place (computation of a signal, bus communication), or where some condition is met.

Following a convention long used in synchronous programming [6] (and inspired from synchronous circuit design) we call these functions *logical clocks*, or simply *clocks*. In our case, clocks are useful because they represent activation in a way that is independent from the actual computation of activation at runtime, thus simplifying the formal manipulation of activation conditions.

4.1 Basic clock operations

The Boolean operations $\vee$ (union), $\wedge$ (intersection), and $\neg$ (negation) are instant-wise extended onto clocks. Similarly for the difference operator defined as $a \setminus b = a \wedge \neg b$. We say that a clock c_1 is a sub-clock of clock c_2 if $c_1(i) \Rightarrow c_2(i)$ for all $i \in \mathbb{N}$.

By definition, if C is a finite set of clocks, $\bigwedge C$ is the conjunction of all the clocks in C, and $\bigvee C$ is their disjunction.

We denote with $\top$ the clock that is always true, which corresponds to the activation condition of the algorithm itself. We denote with $\perp$ the clock that is always false.

4.2 Signals

A signal s is a value that is computed and can be used in computations at precise execution instants given by a clock $clk(s)$. At every such execution instant, s is assigned a unique value. Inside each instant, the value of s can be read only after being assigned a value. A signal s has a type $\mathcal{D}_s$. Given a signal s and a clock $c \leq clk(s)$, we define the *sub-sampling of* s *on* c, denoted $s@c$, as the signal of clock c and domain $\mathcal{D}_s$ that has the same values as s at instants where c is true.

In a dataflow graph, each value is produced by exactly one output port in an execution instant (due to the single assignment rule). However, hierarchical decomposition and conditional execution mean that the value may be produced by different atomic dataflow functions at different execution instants. At scheduling time, it is useful to have a naming facility unifying such output values corresponding to the same high-level output port. We use signals to perform this naming function, and we associate to each algorithm a the set $Sig(a)$ of such signals (see Appendix A.2 for an example). We create a signal in $Sig(n)$ for each:

- Input and output port of the algorithm node. The input ports of the other nodes use the signal associated with output port feeding them.
- Output port of a dataflow node, whenever the port is not connected to the input port of an output node. The remaining output ports produce values of the signal associated with the output port they feed.

For a signal s, we denote with $node(s)$ the node whose input or output port defines s. By construction $clk(s) = clk(node(s))$. We denote with $sig(p)$ the unique signal associated to some port p of a node in the algorithm.

4.3 Conditioned clock. Clock tree

Given a set of signals S and $X \subseteq \prod_{s \in S} \mathcal{D}_s$ we shall write $S \in X$ to denote the condition that the valuation of the signals of S belongs to X.

Consider now a clock c and the finite set of signals S such that $clk(s) \geq c$ for all $s \in S$. Consider $X \subseteq \prod_{s\in S} \mathcal{D}_s$ a set of valuations of the signals in S. Then, we can define the sub-clock of c, denoted $c.[S \in X]$ which is true whenever c is true and $S \in X$. We call such a sub-clock a conditioned clock.

Using the conditioning operator, it is easy to define the clocks of all the nodes in an algorithm, in a top-down manner. The clock of the algorithm node is $\top$. Given the clock $clk(n)$ of a composed node n:

- If n is not conditioned, then the clock of all the nodes in its unique expansion is $clk(n)$.
- If n is conditioned, then the clock of all the nodes in expansion $\mathcal{G} \in \mathcal{E}(n)$ is $clk(n).[\mathcal{C}(n) \in {cond_n}^{-1}(\mathcal{G})]$, where ${cond_n}^{-1}(\mathcal{G}) = \{x \in \prod_{c\in\mathcal{C}(n)} \mathcal{D}_c \mid cond_n(x) = \mathcal{G}\}$.

The resulting clocks are all of the form $\top.[S^1 \in X^1]. \ldots .[S^k \in X^k]$, which are naturally organized in a tree with $\top$ as root.

4.4 Clock formulas

While abstract clocks facilitate the formal manipulation of activation conditions, at runtime we must specify a way to compute them. In our setting, the computation of all clocks starts from $\top$, which is the condition that is true at each execution cycle.[2] Starting from $\top$, the computation of a clock c proceeds as a sequence of signal tests (conditioning operators) and Boolean clock operations. We shall call such a sequence a *clock formula*. The definition of a clock formula f includes the definition of its support − the set of signals it depends on $supp(f) = \{s_1@c_1, \ldots, s_k@c_k\}$.

For instance, consider the signals of integer domain $s_1@\top$ and $s_2@\top$. Then, the clock $c_0 = \top.[s_1 \in \{1\}].[s_2 \in \{2\}]$ can be computed from the signals $\{s_1@\top, s_2@\top.[s_1 \in \{1\}]\}$ by the following pseudo-code running at each activation of the algorithm:

```
c0:= false; read s1; if s1=1 then if s2=2 then c0:=true end end
```

As mentioned above, a given clock can be computed in various fashions. For instance, the clock c_0 defined above can also be computed from the signal sets $\{A@\top, B@\top\}$ or $\{B@\top, A@\top.[B \in \{2\}]\}$. This remark is important, as inside a schedule, depending on what information is available on the bus, it may be interesting to use one or the other of the clock formulas (*e.g.* to minimize communication).

All the clocks that can be generated starting from $\top$ using conditioning and the Boolean clock operators have clock formulas computing them. The inductive proof of this result is given in Appendix A.3.

5 Flattened dataflow

The hierarchy of a SynDEx specification facilitates the specification of complex control patterns, ensuring through simple construction rules that every value

[2]This is due to the fact that (1) no global or local time reference is available to allow the definition of time-related clocks, and (2) all inputs and outputs are bound to clock $\top$, meaning that they can't serve as independent primitive clocks.

that is consumed has been produced. However, actual dataflow computations are only performed by the leaves of the hierarchy tree (dataflow functions and delays), while composed nodes only represent control (conditional execution). **We want to be able to represent schedules that specify the spatial and temporal allocation at the level of the hierarchy leaves**.[3]

To facilitate the description of such flat schedules, we identify in this section the set $Op(a)$ of *operations* that need to appear in a correct schedule of the algorithm a. Each $o \in Op(n)$ is associated a clock $clk(o)$ defining its activation, a set $IS(o)$ of signals it reads, and a set $OS(s)$ of signals it produces. The set $Op(n)$ is formed of:

- All the atomic dataflow functions that are not input or output nodes. For such a node n we set $IS(n) = \{sig(i) \mid i \in \mathcal{I}(n)\}$ and $OS(n) = \{sig(o) \mid o \in \mathcal{I}(n)\}$.
- For each delay node n:
  - One read operation $read(n)$ with $clk(read(n)) = clk(n)$, $IS(read(n)) = \emptyset$, and $OS(store(n)) = \{sig(o)\}$, where o is the output port of n.
  - One store operation $store(n)$ with $clk(store(n)) = clk(n)$, $OS(store(n)) = \emptyset$, and $IS(store(n)) = \{sig(i)\}$, where i is the input port of n.
- For each input port i of the algorithm node a, an input operation $read(i)$ with $clk(read(i)) = \top$, $IS(read(i)) = \emptyset$, and $OS(read(i)) = \{sig(i)\}$.
- For each output port o of the algorithm node a, an output operation $send(o)$ with $clk(send(o)) = \top$, $OS(send(o)) = \emptyset$, and $IS(send(o)) = \{sig(o)\}$.

We associate no operation to nodes that are not leaves, as (1) all control information is represented by clocks, and (2) we assume control to take no time, so that we don't need to attach its cost to some object. This timing abstraction is consistent with the example execution mechanism given in Appendix A.1.

5.1 Timing and placement information

To allow real-time scheduling, we assume each operation $op \in Op(a)$ is assigned a worst-case execution time $d_P(op)$, on each processor P. To represent the fact that a processor P cannot execute an operation op, we set $d_P(op) = \infty$. This last convention is particularly important for the operations corresponding to algorithm inputs and outputs. These operations must be placed on the processor having the corresponding input or output channel.

In addition to operation timings, we need communication timings. We associate a positive value $d_{\mathcal{B}}(\mathcal{D})$ to each domain $\mathcal{D}$. This value represents the worst-case duration of transmitting one message of type $\mathcal{D}$ over the bus (in the absence of all interference).

[3] Such a formalism can also represent coarser-grain allocation policies where allocation is done at some other hierarchy level.

6 Static schedules

In this section, we define our model of static real-time schedule of an algorithm over the bus-based architecture defined in Section 2. To simplify the developments, we assume that scheduling is done for one execution instant, the global scheduling being an infinite repetition of the scheduling of an instant. This corresponds to the case where the computations of the different execution instants do not overlap in time in the real-time implementation. The results of this paper can be extended to the case where instants can overlap.

The remainder of the section defines our formal model of static schedule. The definition includes some very general choices, such as the use of data messages for synchronization. These choices will be clearly identified in the text. The resulting formalism is general enough to allow the representation of schedules generated by several common static scheduling techniques, including the ones of SynDEx.

6.1 Basic definitions

Consider an algorithm a. A static schedule $\mathcal{S}$ of a on the bus-based multiprocessor system is a set of *scheduled operations*. Each $so \in \mathcal{S}$ has (1) one execution resource, denoted $Res(so)$, which is either the bus or one of the processors, (2) a clock $clk(so)$ defining the activation condition of the operation, (3) a clock formula $form(so)$ for computing $clk(so)$, (4) a real-time date t_{so} giving the latest (worst-case) real-time date at which the execution of the operation will start, and (4) resource-specific information:

- if $Res(so) = \mathcal{B}$, the scheduled operation is the emission of a signal denoted $sig(so)$ by processor $emitter(so)$.
- if $Res(so) = P_i$, the scheduled operation is the execution of an operation $op(so) \in Op(a)$ with $clk(so) = clk(op(so))$.

For convenience, we define $\mathcal{S}_{\mathcal{B}} = \{so \in \mathcal{S} \mid Res(so) = \mathcal{B}\}$ and $\mathcal{S}_{P_i} = \{so \in \mathcal{S} \mid Res(so) = P_i\}$. We also define the duration of a scheduled operation as $d(so) = d_{\mathcal{B}}(\mathcal{D}_{sig(so)})$ if $Res(so) = \mathcal{B}$ and $d(so) = d_{Res(so)}(op(so))$ otherwise.

Note that the definition of $\mathcal{S}_{\mathcal{B}}$ implicitly assumes that we make no use of specialized synchronization messages, nor data encoding, only allowing the transmission of messages containing signal values.

Unicity. We make the hypothesis that each operation is scheduled exactly once in $\mathcal{S}$, meaning that no optimizing replication such as inlining is done. This assumption is often used in real-time scheduling. Formally, we require that for all $o \in Op(a)$, there exists a unique $so \in \mathcal{S}$ with $op(so) = o$.

6.1.1 Data availability

The clock defining the execution instants where the signal s is sent on the bus before date t is:

$$clk(s, t, \mathcal{B}) = \bigvee \{clk(so) \mid (so \in \mathcal{S}_{\mathcal{B}}) \wedge (sig(so) = s) \wedge (t_{so} + d_{\mathcal{B}}(\mathcal{D}_s) \leq t)\}$$

Recall that we assumed that each input arrives on a single processor, and that each non-input signal is computed on a single processor. Then, $clk(s,t,\mathcal{B})$ is the clock giving the instants where s is available system-wide at all dates $t' \geq t$.

The clock defining the instants where s is available on P at date t is:

$$clk(s,t,P) = clk(s,t,\mathcal{B}) \vee \bigvee \{clk(so) \mid (so \in \mathcal{S}_P) \wedge (s \in OS(op(so))) \wedge (t_{so} + d_P(op(so)) \leq t)\}$$

The second term of the conjunction corresponds to the local production of s.

Given a schedule $\mathcal{S}$, a signal v and a clock $c \leq clk(s)$, we denote with $ready\_date(P,s,c)$ the minimum t such that $clk(s,t,P) \geq c$, and with $ready\_date(\mathcal{B},s,c)$ the minimum date t such that $clk(s,t,\mathcal{B}) \geq c$.

6.2 Correctness properties

The following properties define the correctness of a static schedule $\mathcal{S}$ with respect to the initial algorithm a (viewed as a flattened graph), and the timing information that was provided. This concludes the definition of our model of real-time schedule, and allows us, in the next section, to state the aforementioned optimality properties, and define the corresponding optimization transformations.

Delay consistency. The read and store operations of a delay node must be scheduled on the same processor, to allow the use of local memory for storing the value. Formally, for all delay node n and $so_1, so_2 \in \mathcal{S}$, if $op(so_1) = op(so_2) = n$, then $Res(so_1) = Res(so_2)$.

Exclusive resource use. A processor or bus cannot be used by more than one operation at a time. Formally:

- *On processors:* If $so_1, so_2 \in \mathcal{S}_P$ such that $so_1 \neq so_2$ and $clk(so_1) \wedge clk(so_2)$, then either $t_{so_1} \geq t_{so_2} + d_P(op(so_2))$ or $t_{so_2} \geq t_{so_1} + d_P(op(so_1))$.
- *On the bus:* If $so_1, so_2 \in \mathcal{S}_\mathcal{B}$ such that $so_1 \neq so_2$ and $clk(so_1) \wedge clk(so_2)$, then either $t_{so_1} \geq t_{so_2} + d_\mathcal{B}(\mathcal{D}_{sig(so_2)})$ or $t_{so_2} \geq t_{so_1} + d_\mathcal{B}(\mathcal{D}_{sig(so_1)})$.

Causal correctness. Intuitively, to ensure causal correctness our schedule must ensure in static fashion that when a computation or communication is using a signal s at time t on clock c, the signal has been computed or transmitted on the bus at a previous time and on a greater clock.

- *On a processor:* For all $so \in \mathcal{S}_P$ of clock formula $form(so)$ we have:
  - If $s@c \in supp(form(so))$, then $clk(s, t_{so}, P) \geq c$
  - If $s \in IS(op(so))$, then $clk(s, t_{so}, P) \geq clk(so)$
- *On the bus:* For all $so \in \mathcal{S}_\mathcal{B}$ of clock formula $form(so)$ we have:
  - If $s@c \in supp(form(so))$, then $clk(s, t_{so}, \mathcal{B}) \geq c$
  - $clk(sig(so), t_{so}, emitter(so)) \geq clk(so)$.

7 Scheduling optimizations

In this section, we use the previously-defined model of static schedule to state some optimality properties which we expect of all schedule, yet which are not easy to state or ensure in the presence of complex activation conditions. We provide algorithms ensuring these properties.

The schedule optimizations we are interested in are based on the removal of redundant idle time and redundant communications. This is done only through manipulations of the clocks and clock functions of the communications on the bus, and through adjustments of the dates of the scheduled operations. In particular, our optimizations preserve the ordering of the computations on processors, as well as the ordering of communications on the bus (with the exception of the fact that only the first send of a given signal is performed at a given instant, the other being optimized out).

7.1 Static dependencies

To simplify notations, we define the sets of bus operations and processor operations that produce a given signal s. They are $\mathcal{S}_{\mathcal{B}}(s) = \{so \in \mathcal{S}_{\mathcal{B}} \mid sig(so) = s\}$ and $\mathcal{S}_P(s) = \{so \in \mathcal{S}_P \mid s \in OS(op(so))\}$.

To allow the optimization of a static schedule, we need to understand which data dependencies must not be broken through removal of communications. Our first remark is that a signal s can be used on all processors as soon as it has been emitted once on the bus. Subsequent emissions bring no new information, so that readers of s should not depend on them. We shall denote with $dep_{\mathcal{B}}(s@c)$ the subset of operations of $\mathcal{S}_{\mathcal{B}}$ that can (depending on activation conditions) be the first communication of signal s on the bus in execution instants where c is true.

$$dep_{\mathcal{B}}(s@c) = \{so \in \mathcal{S}_{\mathcal{B}}(s) \mid (clk(so) \wedge \Big(c \setminus \bigvee\{clk(so') \mid (so' \in \mathcal{S}_{\mathcal{B}}(s)) \wedge (t_{so'} < t_{so})\}\Big)) \neq \bot)\}$$

On a processor, things are more complicated, as information can be locally produced. However, local productions are always the source of the data, and cannot be redundant. We denote the set of local dependencies with $dep\_loc_P(s@c) = \{so \in \mathcal{S}_P(s) \mid clk(so) \wedge c \neq \bot\}$, and we set $c_{loc} = \bigvee\{clk(so) \mid so \in dep\_loc_P(s@c)\}$. With these notations,

$$dep_P(s@c) = dep\_loc_P(s@c) \cup dep_P(s@(c \setminus c_{loc}))$$

Now, we can define the static dependencies between the various scheduled operations. The set of scheduled operations upon which $so \in \mathcal{S}_{\mathcal{B}}$ depends is:

$$dep(so) = dep_{emitter(so)}(sig(so)) \cup \bigcup_{s@c \in supp(form(so))} dep_{\mathcal{B}}(s@c)$$

Similarly, for $so \in \mathcal{S}_P$:

$$dep(so) = \left(\bigcup_{s \in IS(op(so))} dep_P(s)clk(so) \right) \cup \bigcup_{s@c \in supp(form(so))} dep_P(s@c)$$

To these data dependencies, we must add the dependencies due to sequencing on the processors and on the bus, and we obtain the set of static dependencies that must be preserved by any optimization. We denote with $pred(so)$ the set of *predecessors* of a scheduled operation, which includes $dep(so)$ and the dependencies due to sequencing:

$$pred(so) = dep(so) \cup \{so' \in \mathcal{S} \mid (Res(so) = Res(so')) \wedge (clk(so) \wedge clk(so') \neq \bot) \wedge (t_{so} > t_{so'})\}$$

We call *scheduling DAG (directed acyclic graph)* the set $\mathcal{S}$ endowed with the order relation given by the $pred()$ relation.

Determining the exact static dependencies that need to be respected is appealing, but their computation involves, in the definitions of $dep_{\mathcal{B}}(s@c)$ and $dep\_loc_P(s@c)$, and $pred(so)$, the comparison between clocks. In our clock calculus based on conditioning and Boolean operators, the complexity of these tests is at least that of SAT. For this reason, we shall assume from this point on that the relation $pred()$ is not the exact one, but an over-approximation including all the dependencies of the exact relation. A variety of techniques exist allowing an approximate clock calculus, ranging from the low-complexity one implicitly used in SynDEx to BDD-based techniques like those of Wolinski [9].

7.2 ASAP scheduling

The first optimality property we want to obtain is the absence of idle time. In our static scheduling framework, this amounts to recomputing the dates of all the scheduled operations by performing a MAX+ computation on the scheduling DAG defined in the previous section. More precisely, starting on the scheduled operations without predecessors, we set:

$$t_{so} = \max_{so' \in pred(so)} (t_{so'} + d(so'))$$

This operation is of polynomial complexity, and it does not change other aspects of the schedule, meaning that the same implementation will function. The gain is a tighter worst-case bound for the execution of an instant.

7.3 Removal of redundant communication

A stronger result is the absence of redundant communications. We explore two complementary approaches to this problem here.

7.3.1 Removal of subsequent emissions of a same signal

The simplest of the two approaches is the one that seeks to reduce the clock of a signal communication based on previously-scheduled communications of the same signals. The minimization works by replacing for each $so \in \mathcal{S}_{\mathcal{B}}$ the clock $clk(so)$ with:

$$clk(so) \setminus \bigvee \{clk(so') \mid (so' \in \mathcal{S}_{\mathcal{B}}(sig(so))) \wedge (t_{so} > t_{so'})\}$$

The transformation of the schedule is a traversal of the bus scheduling DAG (of polynomial complexity). Due to the minimization of clocks (meaning that some communications are no longer realized at certain instants), the output of this transformation should be put in ASAP form. To do this, the scheduling DAG must be recomputed through a re-computation of the dependencies due to sequencing on the bus. The data dependencies are not changed, as this form of redundancy has been taken into account during the computation of the data dependencies.

The modified schedule can be easily implemented by adding supplementary conditions that rule out a second emission of a signal on the bus.

7.3.2 Removal of useless communication operations

The removal of subsequent emissions reduces to 1 the maximal number of emissions of a signal in an instant. However, it does not try to restrict emissions to instants where the signal is needed. Doing this amounts to a reduction of the clock on which a signal s is sent through the bus in an instant $\bigvee_{so\in\mathcal{S}_{\mathcal{B}}(s)} clk(so)$.

We propose here the simplest such transformation, which consists in the removing of communication operations that are in none of the $pred(so)$ for some $so \in \mathcal{S}$. It is important here to note that the removal of one operation may cause the removal of another, a.s.o. The removal and propagation process can be done in polynomial complexity. The modified schedule is easily implemented by removing pieces of code from the initial implementation.

For a given schedule, the removal of subsequent emissions, followed by the removal of useless communication operation, and by a transformation into ASAP form leads to a schedule where none of the 3 techniques can result in further improvement.

7.4 Time-triggered buses

Our model of static schedule can also represent schedules for systems based on time-triggered buses such as TTA or FlexRay [10]. For such schedules, the removal of redundant communications still works. However, the computation of an ASAP scheduling can no longer be done using the MAX+ formula of Section 7.2 because the result may not be a date where the operation can be performed.

On the other hand, in a time-triggered system all clocks do not need to be computed starting from $\top$, because a notion of time and communication absence is defined. More work is needed to extend our model to cover such schedules.

8 Related work

To our knowledge, no work exists on the optimization of given real-time schedules for conditioned dataflow programs. Meanwhile, an important corpus of heuristic algorithms for real-time scheduling of conditioned dataflow programs (and similar formalisms) exists. We mention here only 3 approaches.

Our approach is closest related to work by Eles *et al.* [4] on the scheduling of *conditioned process graphs* onto multiprocessor systems. Like in our approach, the difficulty is the handling of activation conditions. However, the chosen approach is different. Eles *et al.* start from schedules corresponding to *each execution mode* of the system, and unify them into a global schedule covering all modes. The enumeration of execution modes allows a finer real-time analysis than possible in our model of static schedule (because a given operation has not one worst-case starting date, but one date per mode). The main drawback of the approach is the enumeration of all execution modes of the system, which can be intractable for complex systems. A better approach here could be an intermediate one between ours and Eles', where mode-dependent execution dates can be manipulated, but manipulation is not done on individual modes, but on sets of nodes identified by clocks. A second drawback of the approach, from a communication optimization point of view, is that for most communications the value that is transmitted is not specified.

Our work started from existing work on SynDEx [5]. The SynDEx tool allows the real-time scheduling of conditioned dataflow specifications onto hardware architectures involving multiple processors and communication lines (point-to-point and buses). When applied to architectures with a single bus, SynDEx generates schedules that can be represented using our formalism, but where each communication has either the clock of the sender, or the clock of the intended receiver. This simplifies clocks calculus, but potentially pessimizes communication, meaning that our techniques can be directly applied.

We also mention here the work by Kountouris and Wolinski [9] on the scheduling of *hierarchical conditional dependency graphs*, a formalism allowing the representation of data dependencies and activation conditions whose development is related to the implementation of the Signal/Polychrony language [7]. Kountouris and Woliski focus on the development of scheduling heuristics that use clock calculus to drive classical optimization techniques. We focus on the determination of optimality properties in a more constraint solution space, and on an schedule analysis based on a calculus involving both logical clocks and time.

9 Conclusion

We have defined a new model of real-time schedule of dataflow specification with conditional execution. The formalism includes the definition of a clock calculus for the manipulation of activation conditions.

Using the new formalism, we stated two optimality properties, whose expression is difficult in the presence of complex activation conditions. We provide schedule transformations that ensure these optimality properties.

We thus provide a non-trivial optimization algorithm that can be readily used in embedded system development, and a model that can serve as base for the development of further optimizations.

9.1 Future work

We envision several directions for extending the work of this paper. First of all, the schedule representation formalism should be improved to allow a finer characterization of worst-case execution dates, à la Eles *et al.* [4].

A second direction is the extension of the model to cover architectures with multiple buses. A natural question in this case is which of the optimality results still stand.

A third direction consists in extending the clock calculus, to cover specifications where the clocks are less hierarchized, or defined by external events, such as periodic input arrivals. A good starting point in this direction is the use of *endochronous* synchronous systems [11] specified in the Signal/Polychrony language [7].

References

[1] The Scade product page: http//www.esterel-technologies.com/products/scade-suite/.

[2] The Simulink product page: http://www.mathworks.com/products/simulink/.

[3] J. Dennis. First version of a dataflow procedure language. In *Lecture Notes in Computer Sci.*, volume 19, pages 362–376. Springer-Verlag, 1974.

[4] P. Eles, K. Kuchcinski, Z. Peng, P. Pop, and A. Doboli. Scheduling of conditional process graphs for the synthesis of embedded systems. In *Proceedings of DATE*, Paris, France, 1998.

[5] T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *Proceedings MEMOCODE'03*, Mont Saint-Michel, France, June 2003.

[6] P. L. Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal, a data-flow oriented language for signal processing. *Acoustics, Speech, and Signal Processing (see IEEE Transactions on Signal Processing)*, 34:362–374, 1986.

[7] P. L. Guernic, J.-P. Talpin, and J.-C. L. Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers*, April 2003. Special Issue on Application Specific Hardware Design.

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[9] A. Kountouris and C. Wolinski. Efficient scheduling of conditional behaviors for high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 7(3):380–412, July 2002.

[10] R. Obermeisser. *Event-Triggered and Time-Triggered Control Paradigms.* Springer, 2005.

[11] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, March 2006.

A Appendix

A.1 Execution mechanism

We give here the intuition behind the constraints imposed on schedules in Section 6, under the form of one complying implementation.

We assume the processors sequential, but we allow all input arrival and to be executed in parallel through a DMA-like mechanism. Each processor maintains an execution queue. Each time a new information is ready (either arriving on the bus or some input channel, or computed locally) an interrupt is generated. The interrupt handler executes light-weight code that (1) updates the local application state taking into account the new data, (2) possibly places some dataflow blocks in the execution queue (when all input they needed has arrived), and (3) possibly sends some local data onto the bus (when specified by the static schedule). Upon return from the interrupt handler, the processor resumes execution of the dataflow blocks (data treatments) placed into the execution queue, one after the other. The execution of the dataflow blocks can't block, meaning that all flow control mechanisms must be implemented by the interrupt handler to avoid message loss or duplication.

This execution mechanism amounts to separating control, implemented by the interrupt handler, from actual data computations, which are placed in the execution queue. We assume control is static, which makes for a static schedule despite the dynamic execution queue.

For scheduling, we assign *worst-case time costs* to computations of dataflow blocks on the processors and to communications on the bus and the input channels. The execution time of interrupts will be abstracted as 0 in our timing model, meaning that control costs nothing. This timing abstraction of the execution mechanism is conservative as soon as the worst-case of the interrupt execution time is added to the duration of each computation and bus communication.

A.2 A SynDEx example

Fig. 1 gives an example of SynDEx functional specification in an intuitive graphical form. Dataflow nodes are represented by boxes, on which the ports are placed. Square boxes are the input and output nodes. The remaining boxes are the other basic and composed nodes. The labels $C = true$ and $C = false$ identify the two expansions of a conditioned node with condition port C.

The specified behavior is the following: At each cycle, add A to the value of the accumulator, until the accumulated value is greater than the input T. Then, decrement A at each cycle until the accumulator is smaller than T. In instants where $C = true$, the output P is computed as $F(B)$. Otherwise, P is computed as $G(A)$.

Dataflow blocks having no dependency between them can be executed in parallel. For instance, if $C = true$ then P can be computed and output as soon as B arrives. On the contrary, the computation of $+$ must wait until both A and B have arrived.

The signals of the algorithm correspond to the inputs T and A, the outputs O and P, the output of the delay, and the output of the "$\leq$" operator.

A.3 Basic clock formulas

We prove here constructively that all the clocks that can be generated starting from $\top$ using conditioning and the Boolean clock operators have clock formulas computing them. The construction proceeds inductively:

- The clock $\top$ is computed by the formula that uses no signal and always returns *true*.
- Assume that c is computed by f with $supp(f) = \{s_1@c_1, \ldots, s_k@c_k\}$. Assume S is a set of signals such as $clk(s) \geq c$ for all $s \in S$. Then, $c.[S \in X]$ is computed by the formula denoted $f.[S \in X]$ that first determines c and then reads and tests the signals of S against X. The formula $f.[S \in X]$ uses the signals $\{s_i@c_i \mid 1 \geq i \geq k\} \cup \{s@c \mid s \in S\}$.
- Similarly, if we assume that $c_i = f_i(s_1^i@c_1^i, \ldots, s_{k_i}^i@c_{k_i}^i)$, $i = 1, 2$, then $c_1 \wedge c_2$ can be computed by the formula $f_1 \wedge f_2$ that computes c_1, c_2 and then computes their conjunction. We have $supp(f_1 \wedge f_2) = supp(f_1) \cup supp(f_2)$. Similar reasoning produce $f_1 \vee f_2$, $f_1 \setminus f_2$, and $\neg f_1$, with $supp(f_1 \vee f_2) = supp(f_1 \setminus f_2) = supp(f_1) \cup supp(f_2)$ and $supp(\neg f_1) = supp(f)$.

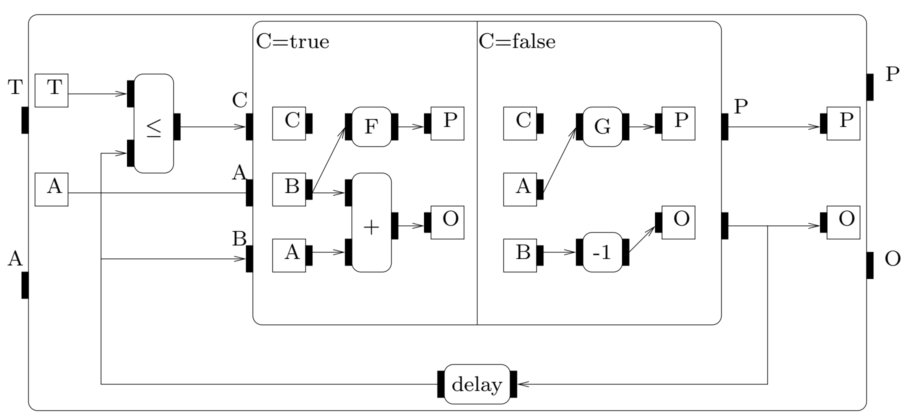

Figure 1: Example of SynDEx algorithm

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399