An Algorithm for Process Partitioning and Deadline Assignment of a Dataflow Application*

Cesare Bartolini, ISTI, CNR, Pisa, Italy
Giuseppe Lipari, Scuola Superiore Sant'Anna, Pisa, Italy

Abstract

The design of an embedded real-time application can be divided in three phases. In the first phase, the functional aspects of the application are specified, usually by means of a dataflow-like formalism. In the second phase, the architecture of the system is specified in terms of hardware capability, RTOS to be used, etc. In the last phase, the functional specification is mapped on to the system architecture and the performance of the system is evaluated. This operation is usually done manually. Moreover, there is no automatic tool to support that permits to evaluate the correctness of the mapping and the resulting performance. In this paper, we present a general dataflow model that allows the designer to assign real-time constraints in the early phase of the design. Then we present a simple algorithm that automatically maps a dataflow specification onto a set of real-time tasks to be executed in a RTOS. The algorithm is based on the assumption that the RTOS provides EDF scheduling. A schedulability analysis is presented that permits to evaluate if the temporal constraints can be met.

1 Introduction

In the design of an embedded system, it is important to separate the functional aspects of the application from the architectural aspects. For this reason, the design process for embedded systems can be logically divided in three distinct phases. In the first phase, the functional aspects of the application are addressed. They are related to the functionality of the application and abstract away from the implementation details. Usually, the designer uses a dataflow-like formalism, assuming maximum parallelism among the different activities to be carried out. Examples of such formalism are Simulink [1], Ascet SD [2], Ptolemy [5], etc.. In this phase, the functional and non-functional constraints are specified.

In a second phase, the architectural aspects of the system are specified. In this phase, and hardware platform is chosen, in terms of processors, amount of memory, external devices, networking technology, etc. Also, the SW architecture is specified, in terms of real-time operating system (RTOS), network protocol, etc..

Finally, the functional specification is mapped onto the architecture specification. The different activities to be carried out are assigned to real-time tasks to be executed by the RTOS. Communication mechanisms between tasks are selected. This assignment is done so that the temporal constraints specified in the initial phase are respected. This phase includes an evaluation of the performance of the application on the selected architecture. If the evaluation is satisfactory, the system can be actually implemented. Otherwise, the designers must go back to the first or to the second phase and change some specification. For example, the designer may decide that a faster processor is needed, or more memory is necessary, or that some activity needs a dedicated processor to be carried out. The designer can also decide to use some simpler algorithm to reduce the computational overhead.

The mapping phase is very important because it permits the evaluation of the "quality" of the adopted solution, in terms of cost/benefits. However, currently not many design tools support a mapping phase. In the current practice, the mapping is done manually, using an experienced engineer to select which task will execute which functional activity. Some technique exists that assigns each functional block to a different real-time task, and then selects the most appropriate task parameters so that the temporal constraints are verified. However, this approach may potentially lead to a great number of real-time tasks. In embedded system instead, it is very important to reduce as much as it is possible the resources used by the application like memory, RTOS overhead, and so on.

In this paper, we first present a dataflow model for an application that permits to specify the temporal constraints of an application. Then, we present an algorithm that automatically generates the task set, with all the task parameters. This algorithm is based on the assumption that the RTOS supports an EDF scheduler, though current efforts show some progress in using a fixed priority scheduler. Fi-

*This work has been supported by Ericsson Lab Italy in the framework of the Pisatel initiative (http://www.iei.pi.cnr.it/ERI/frame.htm).

nally, we present the schedulability analysis for the generated task set. If the task set is schedulable, then the original temporal constraints on the dataflow specification will be respected.

In the aims of this research, we assume the words "task" and "process" as referring to the same entity.

2 Related work

Most of the terminology used in this paper is based on some works by Lee et al. [8, 4], with some significative difference.

Gerber, Saksena et al. [6, 7, 9] presented a methodology for automatically deriving the parameters of a task set that describes an embedded application. In particular, in [6], they describe a method for assigning intermediate constraints on the basis of the end-to-end constraints. In their methodology, they start from the task set. The underlying assumption is that the initial mapping of the dataflow application on the task set is already done. The approach described in this paper presents several differences with the one in [6]. First, we automatically derive both the set of tasks and their parameters. Second, we only assume one kind of temporal constraint, that we call "path deadline" (see Section 3.1) and corresponds to the "freshness constraint" of Gerber and Saksena. The work presented here can easily be used in a system where some part of the computation is left to dedicated circuits (ASICs).

The schedulability analysis is based on the processor demand analysis, first presented by Baruah et al. of the schedulability condition has been described in [3].

3 Functional model

In our model, an application is described by a directed acyclic graph (DAG). More formally, a *functional model* is an ordered pair (V, E), where V is a set of vertices representing the *functional blocks* and E a set of edges, representing the *functional links*.

- $V = \{C_1, \ldots, C_n\}$ is the set of functional blocks. Functional blocks are the minimal independent operations composing the application. As we will see later, a process p_i may be composed of one or more functional blocks, but a functional block must belong entirely to a single process. We assume functional blocks to be non-reentrant.

- $E = \{l_1, \ldots, l_m\}$ is the set of functional links which connect and carry data from one functional block to another. Each functional link l_i must have a single source, called $src\,(l_i)$, and a single sink or $snk\,(l_i)$. We say that l_i is an input link for $snk\,(l_i)$and an output link for $src(l_i)$. A functional block can have more than one input link and more than one output link. A link with $src\,(l_i) = C_j$ and $snk\,(l_i) = C_k$ will be denoted by (C_j, C_k).

An *external event* e_i is a particular functional block with no input links. An *output* o_j is a functional block with no output link. Thus the link carrying an external event e_i to the functional block C_j will be (e_i, C_j), while one carrying the data from C_i to the output o_j will be (C_i, o_j).

A functional block will be activated, thus starting its execution, when one of its input links are fired. During or after its execution (depending on the algorithm used) it will fire some or all of its output links. The blocks are activated with an "or" activation semantics, meaning that a block is activated every time one of its input links is fired; if the blocks receives multiple inputs, they are buffered and will be executed as soon as possible. Ideally, there is no limit to the number of pending activations that a functional block (or a process) can sustain.

Definition 1 *A **functional chain** from C_i to C_j, or $P\,(i, j)$, is an ordered sequence $P = [l_1, \ldots, l_n]$ of links that, starting from $C_i = src\,(l_1)$, reach $C_j = snk\,(l_n)$ crossing $n+1$ functional blocks such that $snk\,(l_j) = src\,(l_{k+1})$. C_i will be the chain's source and C_j its sink.*

A functional chain, or simply chain, can also have an external event or an output (or both). Notations will be $P\,(e_i, j)$ for the former and $P\,(i, o_j)$ for the latter.

Definition 2 *In a chain $P\,(i, j)$, if a block $C_h \in P\,(i, j)$ is activated before $C_k \in P\,(i, j)$, the former is a **predecessor** in respect to the latter, which is **successor**; the notation used is $C_h \prec C_k$.*

Definition 3 *A **path** $P\,(e_i, o_j)$ is a chain with e_i as source and o_j as sink.*

Definition 4 *Two paths are **similar** if they have the same source an sink while being different paths, thus crossing different sets of functional blocks.*

Also, being the application described with a DAG, no chain must exist which has the same block as both source and sink; i.e., no cycles must exist in the graph.

3.1 Instances

An external event will be activated several times during the life of the application. Each activation is a different instance of the event and must be treated apart from others. Thus, $e_{i,j}$ is the j-th instance of event i. When an event arrives, one or more paths will be activated, meaning that the

blocks in the paths will start executing. Paths are activated only when events arrive. Thus $P_{i,j}$ is the j-th activation of path i.

Definition 5 *The **path deadline** for P_i, or Δ_i, is the end-to-end constraint for a path P_i, the equivalent for paths of process deadlines.*

The path deadline is a relative time constraint, independent of the actual path instance. In the scope of this research, the path deadline will be a constant value through all instances of a path. A good way of ordering paths is based on their deadlines, from lowest to highest.

In addition, for each path activation there will be an absolute deadline: $\delta_{i,j} = P_{i,j} + \Delta_i$.

In addition, an external event e_i can be periodic or sporadic. In the former case it will repeat every T_i interval; in the latter case, T_i will represent the minimum interarrival time for event e_i, and will be the interval accounted for in the worst-case analysis.

3.2 Processes

The goal of our algorithm is to divide an application into processes. A process is denoted by p_i, while its j-th instance is $p_{i,j}$; $a_{i,j}$ denotes the time of the j-th activation. Relative process deadlines are denoted by $D_{i,j}$, while absolute ones are $d_{i,j}$ (referring to the j-th instance of process p_i). It might be necessary to keep track of more than one instance of each process, because it may be activated by more than one path.

As for the relative deadline, past research has often been based on the assumption that it is the same in every instance of a task, thus it would be D_i if this assumption were true; however, in the aims of this research, the relative deadline is often assigned dynamically, and may change every time the task is activated, so it is necessary to consider the instance index too.

The time when process p_i starts its execution for the j-th activation is $s_{i,j}$; while the time when the same execution is terminated is $f_{i,j}$. Obviously, the deadline is respected if $f_{i,j} \leq d_{i,j}$.

Definition 6 *The **waiting time** for the j-th instance of process p_i is the time difference between its start time and its activation time: $w_{i,j} = s_{i,j} - p_{i,j} \geq 0$.*

Definition 7 *The **response time** for the j-th instance of process p_i is the time difference between its finishing time and its start time: $r_{i,j} = f_{i,j} - s_{i,j}$.*

Definition 8 *The **computation time** for the j-th instance of process p_i is the time it would take executing if there were no other concurrent tasks, and is called $c_{i,j} \leq r_{i,j}$.*

Normally, the computation time is dependent on the process state and inputs, though initially the assumption that $\forall j, k,\ c_{i,j} = c_{i,k}$ will be used. Hopefully there will be an esteem over the length of the computation time; in most cases this will be the *Worst-Case Execution Time* (WCET).

Definition 9 *The **blocking time** for the j-th instance of process p_i is the time during which the process, though it has started executing, isn't running due to scheduling issues; that is, the time during which the process is suspended (after $s_{i,j}$) to allow execution of other tasks: $b_{i,j} = r_{i,j} - c_{i,j}$.*

4 Late activation

Late activation (or LA) is the name of the algorithm presented in this paper. It is a set of rules for partitioning the application into processes and scheduling the task set thus created. This algorithm can be used if there is little or no knowledge of the internal working of the application.

4.1 Process partitioning

The rules for process partitioning with LA are reported below.

1. It is essential that a process is part of a path, meaning that it must contain one or more contiguous functional blocks belonging to the same path. A block of the process might well belong to more than one path, but there must be a path to which all functional blocks in the process belong, plus they must be in sequence.

2. A functional block is non-reentrant. Therefore, to simplify the partitioning algorithm, we restrict the number of possible allocations by imposing that a functional block can be assigned to only one process. Thus, a block which is sink to more than one functional link must necessarily be the first block of a process; all subsequent blocks in the process must have one input link only. With this rule, a process can be activated by several other processes.

3. A block which fires more than one output link should be the end of a process. This rule keeps the algorithm simple and allows a higher degree of flexibility in assigning the process deadlines.

These rules partition the set of functional blocks in a set of processes. The precedence relation between functional blocks imposes a precedence relation between processes. In the following, we will use the notation $p_i \prec p_j$ to denote that process p_i is a predecessor of p_j.

By using these rules, there is the possibility that the application be split into too many processes, thus reducing efficiency and occupying too many resources. For example,

by changing Rule 3, it would be possible to group an entire path into a process. We are currently investigating other algorithms to further reduce the number of processes.

4.2 Scheduling algorithm

The key rule of the algorithm is that every task keeps track of all tasks it has to activate during its execution; when it finishes executing, all those tasks are activated and set ready. Deadlines are assigned at that time, based on the path deadline and the behavior of predecessors. If the activation of a task depends on a conditional expression, and the condition is not met, then the task will *not* be activated. Processes are activated only if required, and all output links of a block are fired at the end of the block's execution, so tasks are ready for execution as soon as they are activated by the scheduler. We assume that tasks are preemptable and that are scheduled by an EDF scheduler.

4.3 Deadline assignment

Relative and absolute deadlines can be easily derived from the end-to-end deadlines. A single firing of each external event will be considered during this study; this way, there will be a single instance for every path and for the first task of every path. The rules for assigning the deadlines are listed below:

- Every time a task is activated by an external event or by a functional link, it is assigned an absolute deadline;
- a process that is activated by an external event, be it p_1, has an absolute deadline of $d_1 = \min_i \{\delta_i \mid p_1 \in P_i\}$, while its relative deadline is

$$D_1 \quad = \quad \min_i \{\Delta_i \mid p_1 \in P_i\}\,; \tag{1}$$

- the absolute deadline of a process $p_{i,l} \in P$, which is activated by $p_{i-1,h} \in P$ at time $f_{i-1,h}$, is $d_{i,l} = \min_j \{\delta_j \mid p_{i,l} \in P_j\}$, while the relative one is

$$D_{i,l} = \min_j \{\Delta_j \mid p_{i,l} \in P_j\} - \sum_k (w_{k,h} + c_{k,h})\,,$$

$$p_{k,h} \in P \wedge p_k \prec p_i. \tag{2}$$

Since a task which is activated by an external event has no predecessors, rule (1) is a special case of rule (2). The late activation algorithm assigns the same absolute deadline to all processes in the same path. It is also evident that deadline computation is based exclusively on past events, and no estimates about future events are needed.

5 Examples

In this section, we present some examples of the late activation algorithm. Note that the applications presented as examples are already partitioned into processes as described in 4.1, thus the blocks appearing in the diagrams represent tasks and not just functional blocks.

5.1 Example 1

The application structure presented here is likely the simplest one for use with this protocol. A single external event activates every path in the graph, and no similar paths exist.

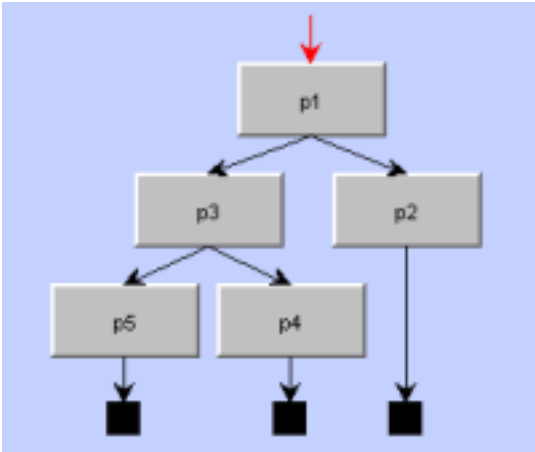

Figure 1. Example 1

The graph presented in figure 1 shows an application made up of three paths, ordered by deadline (assume that "Task i" is indicated with p_i):

- $P_1 = [p_1, p_2]$ has a relative deadline $\Delta_1 = 16$;
- $P_2 = [p_1, p_3, p_4]$ has a relative deadline $\Delta_2 = 20$;
- $P_3 = [p_1, p_3, p_5]$ has a relative deadline $\Delta_3 = 23$.

The only external event activates all paths at once. Every task has a computation time (the WCET will be used) and a relative deadline, which is assigned when the task is activated. Computation times are shown in table 1.

A Gantt chart showing the schedule is presented in figure 2.

Clearly, if there were other processes along one of the paths, their deadlines would have to take into account the waiting times af all tasks in that path. So, for instance, if p_5 activates other tasks, their deadlines are influenced by w_3 and w_5 (not mentioning w_1 since it is zero).

Task	Execution time
p_1	$c_1 = 5$
p_2	$c_2 = 4$
p_3	$c_3 = 6$
p_4	$c_4 = 2$
p_5	$c_5 = 3$

Table 1. Task computation times for example 1

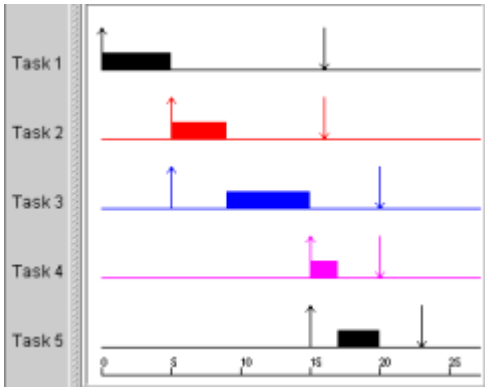

Figure 2. Scheduling chart for example 1

5.2 Example 2

The other example shows an application with two different inputs and as many outputs. The application graph is shown in figure 3, while the task WCETs are listed in table 2.

Task	Execution time
p_1	$c_1 = 4$
p_2	$c_2 = 3$
p_3	$c_3 = 1$
p_4	$c_4 = 3$
p_5	$c_5 = 4$

Table 2. Task computation times for example 2

Again, three paths can be identified within this application:

- $P_1 = [p_1, p_2]$ has a relative deadline of $\Delta_1 = 18$;
- $P_2 = [p_1, p_3, p_4]$ has a relative deadline of $\Delta_2 = 22$;
- $P_3 = [p_5, p_3]$ has a relative deadline of $\Delta_3 = 25$.

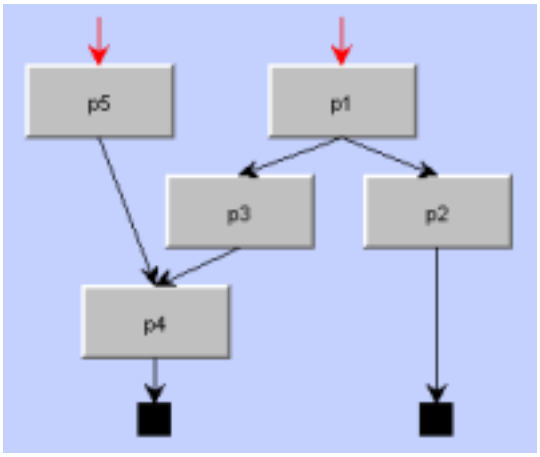

Figure 3. Example 2

This example is based on the application shown in figure 3. A Gantt chart showing the schedule is in figure 4.

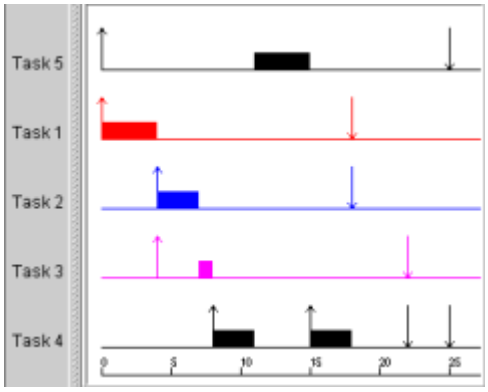

Figure 4. Scheduling chart for example 2

6 Feasibility test

To test the schedulability of the generated task set, we use the processor demand criterion, first proposed by Baruah et al. [3]. The standard test is the following:

$$\forall L \leq L^*, \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i \leq L, \qquad (3)$$

where n is the number of tasks, T_i, C_i and D_i are the minimum interarrival time for the i-th task, its WCET, and its relative deadline respectively, and $L^* = \frac{U \max_i (T_i - D_i)}{1-U}$, being $U = \sum_{i=1}^{n} \frac{C_i}{T_i} < 1$ the processor load. This is for a generic EDF-scheduled task set.

This formula can be used in our algorithm with the following assumption: when an external event e_i arrives, one or more paths are activated, and several tasks will be activated in sequence. Actually, there would be no difference if all tasks were activated at the same time (the event arrival) and executed in the right order. So we use external events in formula (3) in the place of tasks.

n_e is the number of events. The maximum load that an external event can generate on the processor is the sum of the WCETs of all tasks it activates; in addition, some tasks might be activated more than once by the same event, in the presence of similar paths, so this must be accounted for.

$$C_{e_i} = \sum_{j=1}^{m} \alpha_j C_j,\ p_j \in P_k,\ e_i = src\left(P_k\right), \qquad (4)$$

where α_j denotes the maximum number of times task p_j can be activated by a single arrival of event e_i. Pointedly, if $e_1 = src\left(l_1\right)$, and calling $p_1 = snk\left(l_1\right)$ the first task activated by e_1, then $\alpha_1 = 1$, while a task which has m input links has $\alpha_j = \sum_{h=1}^{m} \alpha_h$, $p_h = src\left(l\right)$, $p_j = snk\left(l\right)$. In other words, α for a generic task p_j is the sum of that of all its immediate predecessors (all tasks which can activate p_j).

In addition, the equivalent of D_i can be assigned to the event; this will be the minimum deadline among all paths activated by e_i. $D_{e_j} = \min_j \left\{\Delta_j \mid e_j = src\left(P_j\right)\right\}$. T_{e_i} denotes the minimum interarrival time for the event.

Under these assumptions, external events can replace tasks in expression (3). In the end, the condition is:

$$\forall L \leq L_e^*, \sum_{i=1}^{n_e} \left(\left\lfloor \frac{L - D_{e_i}}{T_{e_i}} \right\rfloor + 1 \right) C_{e_i} \leq L, \qquad (5)$$

where $L_e^* = \frac{U_e \max_i \left(T_{e_i} - D_{e_i}\right)}{1 - U_e}$, and $U_e = \sum_{i=1}^{n_e} \frac{C_{e_i}}{T_{e_i}} < 1$. This condition is an upper bound, so it is sufficient.

7 Conclusion and future work

The late activation algorithm's strength is the fact that it requires no forward knowledge about the application. Absolute and relative deadlines can be assigned only on the basis of past events. In an application where execution times of processes are not known, the algorithm can be applied, since the only dynamic values that need to be known to calculate deadlines are the waiting times, which are known exactly when a task activates another.

This algorithm can be applied in a straightforward way when the RTOS is based on a message passing paradigm. In fact, in this paradigm all tasks are blocked waiting to be activated on a message queue. A task activates another tasks only upon completion. Moreover, a task can be activated several times by different tasks, and no activation must be lost. We are currently investigating the maximum length of a message queue given the application parameters and constraints. In fact, in an embedded system, it is also important to reduce the amount of memory needed by the application. Therefore, it becomes important to compute the maximum length of the queue and then provide an algorithm to reduce this length.

When the RTOS is based on a shared memory paradigm, it may be possible for different blocks (and hence for different tasks) to share data through mutex semaphores. However, in this case the task would experience a blocking time. We are currently extending equation (5) to include blocking times. Another interesting issue is to modify the algorithm and to try to allocate the functional blocks to the task so to minimise the number of possible conflicts on a shared resource.

Finally, we are considering the issue of applying our methodology in a HW/SW codesign environment. In such a context, a functional block might be mapped on a hardware component (i.e. a FPGA) instead of a software task. In this case, the FPGA can be considered as a coprocessor running on single task. One issue to consider is how to allocate the remaining functional blocks to the software tasks so to allow maximum parallelism and efficiency in the final implementation. This problem is very similar to the problem of scheduling a set of real-time tasks in a multi-processor environment, where tasks are statically allocated to processors.

References

[1] Simulink: User's guide. *IEEE Transactions on Systems, Man, and Cybernetics*, pages 811–824, 1992.

[2] ETAS GmbH. Whitepaper Ascet-SD, 1998.

[3] S. Baruah, A. Mok, and L. Rosier. Algorithms and complexity concerning the preemptively scheduling of periodic, real-time tasks on one processor. *Real-Time Systems Journal*, 2:301–324, 1990.

[4] S. Bhattacharyya, S. Sriram, and E. A. Lee. Minimizing synchronization overhead in statically scheduled multiprocessor systems, 1995.

[5] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2):0–, 1994.

[6] R. Gerber, S. Hong, and M. Saksena. Guaranteeing end-to-end timing constraints by calibrating intermediate processes. Technical Report CS-TR-3274, 1994.

[7] D.-I. Kang, R. Gerber, and M. Saksena. Parametric design synthesis of distributed embedded systems. *IEEE Transactions on Computers*, 49(11):1155–1169, 2000.

[8] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987.

[9] M. Saksena and P. Karvelas. Designing for schedulability: Integrating schedulability analsysis with object-oriented design.