Optimized Rapid Prototyping for Real-Time Embedded Heterogeneous Multiprocessors

Thierry Grandpierre, Christophe Lavarenne, Yves Sorel

▶ **To cite this version:**

Thierry Grandpierre, Christophe Lavarenne, Yves Sorel. Optimized Rapid Prototyping for Real-Time Embedded Heterogeneous Multiprocessors. the seventh international workshop, 1999, Rome, France. 10.1145/301177.301489 . hal-01800625

HAL Id: hal-01800625
https://hal.science/hal-01800625

Submitted on 27 May 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Optimized Rapid Prototyping for Real-Time Embedded Heterogeneous Multiprocessors

CODES'99 7th International Workshop on Hardware/Software Co-Design, Rome, May 1999

T. Grandpierre, C. Lavarenne and Y. Sorel
INRIA-Rocquencourt Domaine de Voluceau B.P. 105 - 78153 Le Chesnay Cedex - France
{Thierry.Grandpierre,Christophe.Lavarenne,Yves.Sorel}@inria.fr

Abstract

This paper presents an enhancement of our "Algorithm Architecture Adequation" (AAA) prototyping methodology which allows to rapidly develop and optimize the implementation of a reactive real-time dataflow algorithm on a embedded heterogeneous multiprocessor architecture, predict its real-time behavior and automatically generate the corresponding distributed and optimized static executive. It describes a new optimization heuristic able to support heterogeneous architectures and takes into account accurately inter-processor communications, which are usually neglected but may reduce dramatically multiprocessor performances.

1 Introduction

The increasing complexity of signal, image and control processing algorithms in embedded applications, requires high computational power to satisfy real-time constraints. This power can be achieved by parallel multiprocessors which are often heterogeneous in embedded system: they are made of different types of processors interconnected by different types of communication media. In these systems, communications are too often neglected although they may decrease tremendously the actual performances of the aforementioned applications. In order to help the designer obtain rapidly an efficient implementation (i.e. which satisfies real-time constraints and minimizes the architecture size) of these complex algorithms, and to simplify the implementation task from the specification to the final prototype, we have developed the AAA[1] rapid prototyping methodology[21].

The implementation of an algorithm on a architecture corresponds to a resource allocation problem. As classified in [5] there are two possible resource allocation policy : dynamic or static.

The dynamic policy is more efficient when the execution duration depend on the processed data. The algorithm implementation is also apparently simplified because the dynamic executive provides numerous services (task allocation, communication etc.). But the price to be paid is the induced overhead [3, 1, 17] both in program size and in execution time (mainly caused by expensive context switching and dynamic communication routing). The difficulty to predict run-time execution durations, and run time behavior, forces the designer to insert (large and empirical) safety margins which lead to resources waste, which must be added to the resources consumed by the dynamic operating system itself.

On the other hand, static scheduling minimizes the overall execution time by drastically reducing overheads [3, 7], but it does not allow to implement as various application as the dynamic policy does, because all the properties of the application, including its environment, must be known at compile time. Hence, the static policy is appropriate [17] for the implementation of real-time reactive [4] control, signal and image processing algorithms on embedded machines, where resources and time are hardly limited, and where the algorithm and its environment are well known. An important part of researches on static scheduling focus on minimizing the number of inter-processor communications [12, 25, 16] but rarely on their optimization (scheduling, routing [24], parallelism) which can be done statically but requires to be able to allocate the communication sequencers usually ignored. This lack of control induces inaccurate estimated communication delays and can cause unpredictable performance degradations [1]. This paper addresses this crucial issue, which has drawn too little attention in RTOS and real-time executive researches [7].

Based of these observations, AAA uses static scheduling (even communications are routed and scheduled statically) and rapidly leads to a debug-free optimized prototype which is consequently reusable as we will see along this paper. In section 2, we present the models used in AAA. In order to address the NP-complete [8, 19] resource allocation problem, and because we aim rapid prototyping, section 3 presents a fast greedy list scheduling heuristic which allows to rapidly predict and optimize the performances of different kinds of algorithms on different kinds of architectures. Section 4 gives rules to automate the generation of the static executive corresponding to the chosen implementation. Section 5 presents both the software tool SynDEx which implements AAA, and an application designed and realized with it. Finally section 6 gives a brief conclusion.

2 AAA Methodology

2.1 Architecture Model

The heterogeneous multiprocessor target architecture is specified as a non oriented hypergraph of operators (graph vertices), that may be of different types, connected through bidirectional communication media (non oriented graph edges), that may also be of different types. A communication medium may connect two operators or more. This graph (in the middle of figure 1) describes the available parallelism of the architecture. Each operator is a finite state machine (programmable, with instruction and data memories) which

[1] AAA stands for Adequation Algorithm Architecture, "adequation" is a french word meaning an efficient matching

executes sequentially a subset of the algorithm's *operations* (Cf. next section). Each communication medium executes sequentially *communication operations* (Cf. section 2.3). A medium includes not only the wires needed to move data spatially between operators memories, but also the *transformator* units (DMA or UART) that sequences memory accesses on each side of the wires. Each transformator is a finite state machine and consequently, a medium which is composed of several communicating transformators is equivalent to a single finite state machine. On each side, the communication medium is able to synchronize with the operator in order to access, in alternation (mutual exclusion), shared data buffers. Since in general, a *transformator* requires the operator sequencer (via short interrupts) for sequencing and executing operations that it is not able to do itself, a processor (CISC, DSP, etc) usually includes not only one operator, but also a part of each connected media.

2.2 Algorithm Model

An algorithm, as defined by Turing and Post [23], is a finite sequence (total order) of operations directly executable by a finite state machine. For a multiprocessor architecture, composed of several finite state machines, algorithms must be specified with at least as much parallelism as the architecture. Since we want to be able to compare the implementation of an algorithm on different architectures, the algorithm graph must be specified independently of any architecture graph. Thus, we extend the notion of algorithm to an oriented hypergraph of operations (graph vertices), which execution is partially ordered by their data dependences (oriented graph edges). We need an hypergraph model (an example is given in the top of figure 1) because each data dependence may have several extremities but only one origin (diffusion). This *dependence graph*, also called directed acyclic graph (DAG) [10, 20], exhibits a potential parallelism: two operations which are not in data dependence relation, may be executed in any order by the same operator or simultaneously by two different operators. We deal with reactive systems which are in constant interaction with the environment that they control. This is why the algorithm operations needed to compute the output event for the actuators, from the input event from the sensors, are indefinitely repeated, once for each sampling of the sensors. In other words, the algorithm is an infinitely wide, but periodic, acyclic dependence graph, reduced by factorization to its repetition pattern [15] (also called a dataflow graph), which is executed for each input event.

In order to detect design mistakes as soon as possible in the development cycle, algorithm graphs may be produced by the compilers of high level specification languages, such as the Synchronous Languages [9] (Esterel, Lustre, Signal), which perform formal verifications in terms of events ordering in order to prevent dead-locks.

2.3 Implementation Model

The prime assumption of our implementation model is that each operation of the algorithm graph does not require more than one operator for its execution, but there must be at least one operator of the architecture graph, able to execute it. When several operators are able to execute an operation, one of them must be chosen in order to execute it.

The execution of each operation by an operator consists in reading the operation's input data from the operator memory, then in combining them to compute the output data which are finally written into the operator memory. Therefore, when two operations in data-dependence relation are executed by the same operator, the operation producing the data must be executed in sequence before the operation consuming the data. When two data-dependent operations are executed by two different operators, the data must be

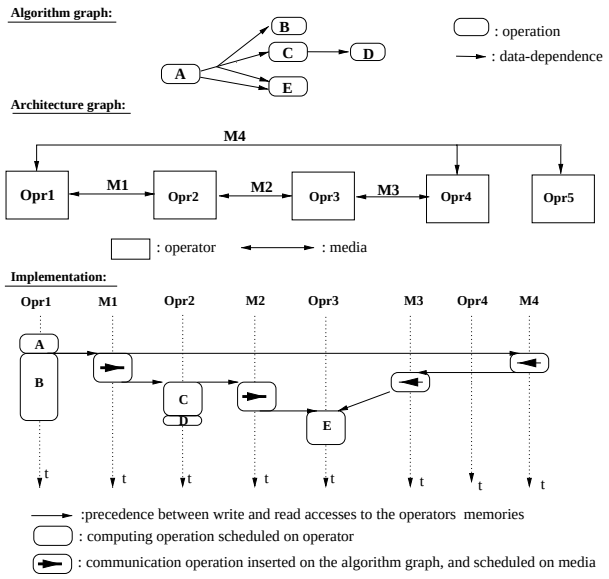

Figure 1: Example of implementation

transferred, from the memory of the operator executing the producing operation (after its execution), into the memory of the operator executing the consuming operation (before its execution). Such a data dependence is called an *inter-operator data dependence*. In order to support it, a *route* (path in the architecture graph) must be chosen between the two operators. For each communication medium composing that route, one communication operation (executed by the corresponding medium) must be inserted in the algorithm graph between the producing and the consuming operations. When the route is composed of more than one intermediate medium, the data must be temporarily stored in the memory of each intermediate operator. The implementation requires not only to choose, for each operation, the operator by which it will be executed, but also to choose, for each inter-operator data dependence, the media that will execute communication operations to route the data. These choices correspond to a spatial allocation of the architecture resources, usually called partitioning or placement, called hereafter *distribution*. Since operators (resp. media) are finite state machines, the implementation also requires to choose, for each operator (resp. medium), an execution order between the operations (resp. communication operations) assigned to it, compatible with the precedences required by the data dependences. These choices correspond to a temporal allocation of each architecture resource, called hereafter *scheduling*.

Distribution and scheduling of operations, as well as their optimization in the case of homogeneous architectures, have been formalized in terms of graph transformations in [21]. Here, we take into consideration heterogeneous architectures (see section 3), and we improve two routing aspects of the implementation. In [21], all data dependences between two operators are routed through the same path, arbitrarily chosen among the shortest ones. We reexamine the route choice for each inter-operator data dependence, such that two simultaneous ones may be parallelized on different routes instead of being sequenced on the same arbitrary chosen route as in [21]. Moreover in [21], data diffusion is implicitly allowed only in the operators at either extremities of a route. Whereas here, we allow diffusion in each intermediate operator on the route, so that, for example, diffused data, instead of being communicated once through each route, i.e. twice through each medium shared by the

two routes, will be transferred only once through each medium of either route.

Figure 1 shows a simple example of implementation with communication optimization based on parallel routing and diffusion: operations (A,B), (C,D), and E are assigned respectively to operators Opr1, Opr2, and Opr3. The two data dependences between A and E are communicated simultaneously on the two (same length) parallel routes M1-M2 and M4-M3. The data produced by A on Opr1 is first transferred to Opr2 (by the communication operation executed by M1), where it is diffused to C and to the communication operation executed by M2, by which it is transfered to Opr3 for E. In parallel, the second data produced by A, is transfered to C by another route made of M4, Opr4, and M3.

3 AAA Optimization Heuristic

For a given pair of algorithm and architecture graphs, there is a large but finite number of possible implementations, among which we need to select the most efficient one, i.e. which satisfies real-time constraints and minimizes architecture resources. This optimization problem, as most resource allocation problems [8, 19] is known to be NP-complete, and its size is usually huge for real applications. This is why we use heuristics. Since experience shows that the algorithm graph and the architecture graph are modified several times before a satisfying result is obtained, and because we aim rapid prototyping, we need a fast but efficient heuristic with a graphical global view of the results (Gantt chart) where the user may easily find out critical paths or bottlenecks. Hence, if the rapidly predicted execution time does not match the real-time constraint, the user can either modify the algorithm graph to offer more parallelism, or add distribution constraints (in order to work around singular aberrant results inherent to any heuristic). Then the user applies the heuristic again. On the contrary, if the real-time constraint is met, the user can try to reduce the architecture graph and applies the heuristic again.

In order to reach the above requirements, our heuristic is based on a fast and efficient greedy list scheduling algorithm [18, 2, 25], with a cost function that takes into account the execution durations of operations and of communications. These durations are obtained by a preliminary step of characterization.

3.1 Characterization

As mentioned in section 2.1, each operator can execute its own operation set. It is characterized by a lookup table that associates an execution duration to each operation of this set. The first way to obtain characteristics is based on duration estimation: an operation is coded by a sequence of CPU instructions, so its duration depends on the number of clock cycles needed for each instruction, and on the clock period, but also on the peculiarities of the operator (cache management, instruction pipeline). This method is manageable only for simple operators or as a rapid first approximation when the operator is not yet available. The other way is based on duration measurement: each operation may be timed in situ, with the help of an executive automatically generated with chronometric logging operations, inserted between other operations [6].

Every data dependence may happen to be inter-operator, this is why the execution duration of communication operations, for each data type (int, float…) used in the algorithm, have to be characterized for each communication medium used in the chosen architecture. At this point there are usually two subsets of data types to consider, basic scalar types and composite types. It is not always possible to extrapolate the transfer duration of a composite type from the transfer durations of its components, whereas it is always possible to measure the duration of the transfer of each data type (including composite types) for each medium, and to store the result in a lookup table characterizing the medium.

3.2 Definitions

The greedy list scheduling algorithm used in our heuristic, is based on a cost function defined in terms of the start and end dates of the operations executions, themselves expressed recursively from the execution durations of operations and communications. Whereas in [21] the architecture graph is assumed homogeneous, i.e. the execution duration $\Delta(o)$ of an operation (resp. communication operation) o is the same on each operator (resp. medium), here we consider heterogeneous architectures where the execution duration of an operation (resp. communication operation) o executed by an operator (resp. medium) p in *Gar* (the architecture graph) is defined by $\Delta_p(o)$ (this value is found in p's characteristics lookup table). As in [21], the set of successors (resp. predecessors) of an operation o in *Gal* (the algorithm graph) is denoted $\Gamma(o)$ (resp. $\overline{\Gamma}(o)$), and its "earliest start from start" date $S(o)$, its "earliest end from start" date $E(o)$, its "latest end from end" date $\overline{E}(o)$, its "latest start from end" date $\overline{S}(o)$, the makespan (algorithm graph critical path length) R, and its "schedule flexibility" $F(o)$, are defined by[2]:

$$
\begin{aligned}
S(o) &= \max_{x\in\overline{\Gamma}(o)} E(x) \quad (\text{or } 0 \text{ if } \overline{\Gamma}(o)=\emptyset)\\
E(o) &= S(o)+\Delta_p(o)\\
\overline{E}(o) &= \max_{x\in\Gamma(o)} \overline{S}(x) \quad (\text{or } 0 \text{ if } \Gamma(o)=\emptyset)\\
\overline{S}(o) &= \overline{E}(o)+\Delta_p(o)\\
R &= \max_{o\in Gal} E(o) = \max_{o\in Gal} \overline{S}(o)\\
F(o) &= R-S(o)-\Delta_p(o)-\overline{E}(o)
\end{aligned}
$$

When the heuristic cost function considers an operation o, all o's predecessors are already scheduled (the operators or media that will execute them have already been chosen), but no successor of o's is yet scheduled, so in the computation of $\overline{E}$ and $\overline{S}$, Δ has to be defined independently of any operator. We define the execution duration of an operation o not yet scheduled by the average execution duration of o on all operators able to execute it:

$$Gar/o = \{p \in Gar | \Delta_p(o) \text{ is defined}\}$$

$$\Delta(o) = \frac{\sum_{p\in Gar/o} \Delta_p(o)}{\text{Card}(Gar/o)}$$

As soon as an operation o is scheduled on an operator p, its duration changes from $\Delta(o)$ to $\Delta_p(o)$, and for each predecessor o' of o's not scheduled on p, communication operations have to be inserted between o' and o, and to be scheduled on media. Consequently, $S(o)$ may change to a bigger value $S'(o)$, and the makespan R may also change to a bigger value R'. As in [21], our cost function $\sigma(o)$, called the *schedule pressure*, is the difference between the *schedule penalty* $P(o)=R'-R$ (critical path increases due to this scheduling step) and the schedule flexibility $F(o)=R'-S'(o)-\Delta_p(o)-\overline{E}(o)$ (schedule margin before a critical path increase), which gives:

$$\sigma(o) = S'(o)+\Delta_p(o)+\overline{E}(o)-R$$

$\sigma(o)$ is an improved version of the often used $F(o)$, because when o becomes critical, $F(o)$ stops decreasing and remains null, whereas $P(o)$ starts increasing from 0, i.e. $\sigma(o)$ continues to increase; moreover the comparison between the schedule pressures of two operations eliminates R and its expensive computation.

[2] Note the symmetry of formulas: "From start" and "From end" are relative to opposite time directions and origins; $\min_{o\in Gal} S(o) = 0 = \min_{o\in Gal} \overline{E}(o)$ by definition. In the literature [13], $ASAP = S$ and $ALAP = R-\overline{S}$.

OPR1	OPR1	OPR2	OPR3	OPR4	OPR5
M1		×	×		
M4			×	×	×
Route's length	0	1	2	1	1

Table 1: Route table of OPR1

3.3 Operations Scheduling

The heuristic iterates on a set O_s of "schedulable" operations. An operation not yet scheduled is schedulable when all its predecessors are already scheduled. At each main step of the heuristic, one schedulable operation o is elected, and scheduled on its best operator (see details hereunder), then because o becomes scheduled, some of its successors may become schedulable on their turn. This main step is repeated until O_s is empty, which happens only after a full exploration of the "successor" partial order relation, i.e. after all operations are scheduled, in an order compatible with that partial order. The best operator for a schedulable operation o is either the one where o is constrained by the user to be executed, or the one for which o has the lowest schedule pressure $\sigma(o)$. In order to compare schedule pressures, a schedulable operation is tentatively scheduled on each operator able to execute it. Each operator p has an associated ordered set of operations, called its *schedule*: to be scheduled on p, an operation o is first appended to the end of p's schedule, and then, for each predecessor o' of o's, which is scheduled on $p' \neq p$, communication operations are inserted between o' and o and scheduled on the media of a chosen route connecting p and p'. All computed values corresponding to the best operator ($S_{best}(o)$, $E_{best}(o)$, $\sigma_{best}(o)$, $p_{best}(o)$) are associated with the operation o, in order to be easily retrieved later. In order to fill-up the resources as much as possible, O_s is restricted to $O'_s = \{o \in O_s | S_{best}(o) < E_{best}(o_{min})\}$ where o_{min} is such that $S_{best}(o_{min}) = \min_{o \in O_s} S_{best}(o)$. In this subset, the operation having the highest σ_{best} (i.e. the most critical) is elected, removed from O_s, and definitely scheduled on its best operator. The successors of this scheduled operation that become schedulable (because all of their predecessors are now scheduled) are added to O_s. The next heuristic main step is then ready to begin on this new O_s.

3.4 Communications Scheduling

When an operation o is scheduled on an operator p, for each inter-operator data dependence (between o and an o's predecessor o' which is not scheduled on p), a route must be chosen to transfer the data from the producer operation o' to the consumer operation o. For each medium of that route, a communication operation c must be inserted (between o' and o) and scheduled. The choice of the route is incremental. It starts from the operator executing o', and is simplified by the use of pre-computed routing tables. In each operator p, this table provides for each other operator p' the medium connected to p that allows to reach p' through a minimum number of intermediate media (table 1 gives the routing tables of the Opr1 in the example of the figure 1). At each incremental step of this scheduling procedure a communication operation c is scheduled on a medium. Among the media which connect p to a next operator belonging to one of the shortest route towards p', one elects the medium that induces the minimum execution end date $E(c)$. Each medium m has an associated ordered set of communication operations, called its *schedule*: to be scheduled on m, c is simply appended to the end of m's schedule, except if m's schedule contains a previous communication operation c' transferring the same data (diffusion), in which case $E(c')$ is considered instead. This incremental step is repeated until the destination operator p' is reached. So, at each step of the heuristic, communications are evaluated on each medium of the shortest routes to the destination operator. Thereby, the load of each medium is balanced, the available communication parallelism offered by the hardware architecture is then exploited.

4 Executive Generation

We give here the main ideas about the rules that allow to generate automatically the application-specific executive corresponding to the implementation of a given algorithm on a given architecture.

The executive is generated into several source files, one for each operator, and one for automating the architecture specific compilation chain. Each file contains an intermediate code composed of a list of macro-calls, which will be translated by a macro-processor into a source code in the preferred compilable language for each target operator. Executive macros generate either the desired instructions inline, or a call to a separately compiled operation. They may be classified into two sets. The first one is a fixed set of *system macros*, which support code downloading, memory management, sequence control, inter-sequence synchronization, inter-operator transfers, and runtime timing (in order to characterize algorithm operations and to profile the application as in [6, 1]). The second one is an extensible set of *application specific macros*, which support the algorithm operations.

For each operator, the generated list of macro-calls is composed, in order, of macros describing a tree covering the architecture graph and rooted on a "host" operator (used for downloading application code, and for collecting runtime timings), of macros allocating memory buffers (for data dependences), of one communication sequence for each medium connected to the operator (built from the medium's schedule), and of one computation sequence (built from the operator's schedule). In this executive, context switches only occur between the communication sequences (which are composed of system macros only) and the computation sequence, then only a few registers need to be saved and restored. For example, on an architecture based on TMS320C40 DSPs, the CPU overhead for each communication is only 56 instruction cycles, including "data ready" synchronization, DMA programming, end of transfer interrupt, and "buffer free" synchronization between communication and computation sequences. In order to avoid the overheads of usual communication protocols (such as data transfers between protocol layers, or the addition of routing informations) these sequences share the memory buffers of communicated data, and include synchronization macros to access these buffers in the order required by the data dependences, in a way that this order is satisfied at runtime independently of the operations execution durations. Therefore the implementation optimization, even if it may be biased by inaccurate architecture characteristics, is safe in the sense that it cannot induce runtime synchronization errors (such as deadlocks, or lost data) or an inefficient schedule of communicated data. This certitude allows big savings in application coding and debugging times. Once the executive libraries has been developed for each type of processor, it takes only a few seconds to automatically generate, compile and download the deadlock free code for each target processor of the architecture. It is then easy to experiment different architectures with various interconnection schemes.

5 Related Work

The whole methodology is implemented in the system level CAD software SynDEx[3][22, 14]. Its Graphical User Interface allows the user to specify both the algorithm and the architecture graphs, to

[3] http://www-rocq.inria.fr/syndex

execute the aforementioned heuristic and then to display the resulting distribution and scheduling on a temporal diagram (the bottom of figure 1 presents such a diagram where the vertical size of the operations are proportional to their duration). When the user is satisfied by the predicted timing, SynDEx can automatically generate the dead-lock free executive for the real-time execution of the algorithm on the multiprocessor. Real-time distributed executive libraries have been developed for networks based on DSPs (TMS320C40, ADSP21060), Transputers, microcontrollers, and general purpose processors (PC and UNIX workstations). SynDEx has been used to develop several real-time heterogeneous applications, among which [11]: a new urban electric vehicle controlled by a distributed embedded computer system (based on the CAN bus and five MC68332, one i80c196, one i80486) providing features such as autonomous driving and route planning.

6 Conclusion

The global approach of the AAA methodology aims to rapidly prototype real-time embedded applications. It is based on a static scheduling policy to minimize execution time and program size overheads. AAA uses a fast heuristic which accurately takes into consideration the heterogeneous characteristics not only of operators (computations) but also of media (communications). Moreover, this heuristic optimizes inter-processor communications by exploiting the *communication parallelism* offered by the architecture, and by balancing the load of each communication medium. Finally, the automatically generated executive allows to rapidly carry out an optimized prototype without wasting the user's time in multiprocessor programming and debugging. Presently, we are working on the optimization heuristic in the case of conditioned operations, and on an extension of the AAA methodology for configurable circuits (FPGA) to allow future co-design optimization.

References

[1] Behrooz A. Shirazi, A. Hurson, and Krishna M. Kavi. *Scheduling and load balancing in parallel and distributed system*. IEEE Computer Society Press, 1995.

[2] I. Ahmad and Y. K. Kwok. Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors. In *Int. Symp. on Parallel Architecture, Algorithms and Networks*, pages 207–213, Beijing China, June 1996.

[3] F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-Vincentelli. Scheduling for embedded real time systems. *IEEE Design and Test of Computers*, pages 71–82, January-March 1998.

[4] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Proc. of the IEEE*, volume 79(9), pages 1270–1282, 1991.

[5] T.L. Casavant and J.G. Kuhl. A taxonony of scheduling in general-purpose distributed computing systems. *IEEE Trans. Software Eng.*, 14, No. 2:141–154, 1988.

[6] F. Ennesser, C. Lavarenne, and Y. Sorel. Méthode chronométrique pour l'optimisation du temps de réponse des exécutifs syndex. In *INRIA Research Report*, vol. 1769. 1992.

[7] R. Ernst. Codesign of embedded systems: Status and trends. *IEEE Design and Test of Computers*, pages 45–53, April-June 1998.

[8] Garey and Johnson. *Computers and intractability : a guide to the theory of NP-completeness*. W.H. Freeman, 1979.

[9] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publishers, Dordrecht Boston, 1993.

[10] A.R. Hurson. A program allocation scheme for data flow computers. In *Proc. Int. Conference on Parallel Processing*, volume 1, pages 415–422, University Park, Peen, 1990. Pennsylvania State Univ.

[11] R. Kocik and Y. Sorel. A methodology to design and prototype optimized embedded robotic systems. In *Proc. of the Computational Engineering in Systems Applications CESA'98, Tunisia*, April 1998.

[12] Y. Kopidakis, M. Lamari, and V. Zissimopoulos. On the task assignment problem: Two new efficient heuristic algorithms. *J. of Parallel and Distributed Computing*, 42:21–29, 1997.

[13] Y. K. Kwok and I. Ahmad. Dynamic critical-path scheduling : An effective technique for allocating task graphs to multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[14] C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine. The syndex software environment for real-time distributed systems design and implementation. In *Proc. of the European Control Conference*, 1991.

[15] C. Lavarenne and Y. Sorel. Modèle unifié pour la conception conjointe logiciel-matériel. *Revue Traitement du Signal*, 14(6), 1997.

[16] B. Lee, A.R. Hurson, and T.-Y. Feng. A vertically layered allocation scheme for data flow systems. *Parallel and Distributed Computing*, 11(3):175–187, 1991.

[17] E.A. Lee and J.C. Bier. Architectures for statically scheduled dataflow. *J. of Parallel and Distributed Computing*, 10:333–348, 1990.

[18] Z. Lui and C. Corroyer. Effectiveness of heuristics and simulated annealing for the scheduling of concurrent task. an empirical comparison. *Proc. of PARLE'93, 5th Int. PARLE conference, Munich, Germany, June 14-17*, pages 452–463, Nov. 1993.

[19] V. Sarkar. *Partitioning and scheduling parallel programs for multiprocessors*. MIT press, 1989.

[20] V. Sarkar and J. Hennessy. Compile-time partioning and scheduling of parallel programs. In New York ACM Press, editor, *Symp. Compiler Construction*, pages 17–26, 1986.

[21] Y. Sorel. Massively parallel computing systems with real time constraints, the algorithm architecture adequation methodology. In *Proc. of the Massively Parallel Computing Systems*, May 1994.

[22] Y. Sorel. Real-time embedded image processing applications using the A^3 methodology. In *Proc. of the IEEE Int. Conference on Image Processing*, November 1996.

[23] A.M. Turing. On computable numbers, with an application to the entscheindungs problem. In *Proc. London Math. Soc.*, 1936.

[24] M. Wang. Accurate communication cost estimation in static task scheduling. In Calif. IEEE CS Press, Los Alamitos, editor, *Proc. 24th Ann. Hawaii Int. Conference System Sciences*, volume I, pages 10–16, 1991.

[25] T. Yang and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Computing Journal*, 19:1321–1344, 1993.