CODE GENERATION FOR EMBEDDED PROCESSORS

EDITED BY

Peter MARWEDEL
University of Dortmund
Dortmund, Germany

■

Gert GOOSSENS
IMEC
Leuven, Belgium

KLUWER ACADEMIC PUBLISHERS
Boston/London/Dordrecht

15

SOFTWARE SYNTHESIS FOR REAL-TIME INFORMATION PROCESSING SYSTEMS

Marco Cornero[1†], Filip Thoen[2], Gert Goossens[2] and Franco Curatelli[1]

[1] *DIBE, University of Genova, Genova, Italy*
[2] *VLSI Systems and Design Methodologies Division, IMEC, Leuven, Belgium*

ABSTRACT

Software synthesis is a new approach which focusses on the support of embedded systems without the use of operating-systems. Compared to traditional design practices, a better utilisation of the available time and hardware resources can be achieved with software synthesis, because the static information provided by the system specification is fully exploited. In this paper a novel software synthesis approach for real-time information processing systems is presented. Specifically, a flexible execution model for multi-tasking with real-time constraints is proposed, together with an internal representation model which is well suited for the support of concurrency and timing constraints.

1 INTRODUCTION

Embedded systems are digital systems for dedicated applications, embedded in a larger (usually non-electric) environment. When the correctness of these systems does not only depend on their functional behaviour, but also on their timing behaviour, they are classified as *real-time* embedded systems. Heterogeneous architectures, composed of programmable devices, hardware accelerators and memory, are well suited for the implementation of embedded systems, since they combine the flexibility and the low cost of programmable devices with the efficiency of hardware accelerators for performing critical functionalities [89, 90]. Because of this heterogeneous implementation style, a comprehen-

†Currently with SGS-Thomson Microelectronics, Crolles, France.

sive design environment for embedded systems should support the complete hardware-software co-design trajectory, including hardware/software partitioning, as well as software, hardware and interface synthesis. In this paper we focus on software synthesis. More in particular, hardware/software partitioning is supposed to be already performed, so only portions of the system specification assigned to the software partition are considered.

Software synthesis consists of two sub-problems, namely *task handling* and *code generation*. Task handling, which is the subject of this paper, takes care of : static and/or run-time scheduling of tasks[1], managing the task resource requirements, and inter-task communication. Code generation, on the other hand, generates the static (object) code for the individual tasks. In our system, code generation is performed by means of the CHESS code generator (see Chapter 5).

Our target application domain is advanced real-time information processing systems, such as consumer electronics and personal communication systems. The distinctive characteristic of these systems is the *coexistence of signal processing and control* functions, which require the support of different kinds of timing constraints :

- *Signal processing* functions operate on sampled data streams, and are subject to *real-time* constraints derived from the required sample frequency (throughput) and latency, as determined by the environment.
- *Control* procedures may vary in nature. On the one hand, *soft deadlines* may be imposed (e.g. in a man-machine interface). In this case the procedure has to be executed as soon as possible, but an occasional delay in the execution does not compromise the integrity of the entire system. On the other hand, *hard deadlines* occur (e.g. in a critical feedback control loop). In this case the timing constraints are very stringent.

Traditionally, specialised operating systems called *real-time kernels*, are used for the software support of complex systems. *Software synthesis*, as discussed in this chapter, is an alternative approach to real-time kernels : starting from a system specification, typically composed of concurrent communicating processes with timing constraints, the aim of software synthesis is the automatic generation of the *source code*, realizing : (1) the specified functionalities while satisfying the timing constraints, and (2) the typical run-time support required for real-time systems, such as multi-tasking, and the primitives for process communication and synchronisation. Compared to real-time kernels, a better utilisation of the available time and hardware resources can be achieved

[1]Also called *processes*.

with software synthesis, because the static information provided by the system specification is fully exploited. The automatically generated run-time support is customised for each particular input specification, and does not need to be general, as in the case of real-time kernels. Moreover, an accurate static analysis, usually performed before software synthesis, provides an early feedback to the designer on the feasibility of the input specifications. In this way the trial and error design cycle, typical for real-time kernels, is avoided. Finally, since the output of software synthesis is source code, portability can be easily achieved by means of a retargetable code generator. This is especially interesting when the target processor is still subject to changes, like in the case of application specific instruction set processors (ASIPs).

2 EXISTING APPROACHES IN SOFTWARE DESIGN

Real-time kernels. Real-time kernels have been used extensively as a software support in the design of embedded systems [201, 215, 235]. Most of these kernels are stripped versions of a traditional time-sharing operating system, made appropriate for the real-time domain by reducing the run-time overhead. These small kernels, often with limited functionality, are in the first place designed to be *fast* by ensuring a fast context switch, a quick response to interrupts and a minimal interrupt disable time.

Above all real-time kernels provide : (1) the run-time support for *real-time multi-tasking* to perform software scheduling, and (2) the required primitives for inter-process communication and synchronisation, and for accessing the hardware resources (typically through an application procedural interface). Since processes are considered as black boxes, most kernels apply a *coarse grain* model for process scheduling. Usually a fixed-priority preemptive scheduling mechanism is used, where process priorities have to be specified, rather than timing constraints. Assignment of process priorities, as in the case of the fixed-priority scheduling scheme, is a *manual* task to be performed without any tool support. Typically, an iterative and error-prone design cycle, with a lot of code and priority tuning, is required. Not only is this approach inflexible (adding one task can cause the cycle to be re-iterated) and time consuming, but it also only guarantees correctness for the selected (simulated) stimuli. Additionally, the behaviour of the scheduler under peak load conditions is hard to predict, resulting often in under-utilised systems to stay on the safe side.

Alternatively, traditional scheduling approaches use timing constraints, specified under the form of a process period, release time and deadline [250]. From the designer viewpoint however, these constraints are more naturally specified with respect to the occurrence of observable events. Moreover, the scheduler has no knowledge about the points in time when the events are generated by the processes, and consequently can not exploit this. It is safer to guarantee timeliness at pre-runtime, as new family of kernels tend to achieve [81][230]. Moreover, kernels trade optimality for generality, resulting in a significant run-time and memory overhead.

On the other hand, static scheduling techniques can be used efficiently. In practice this is however only applicable to periodic processes, while asynchronous processes introduce significant overheads, such as active waiting.

Software synthesis. One of the earliest approaches for hardware-software co-design is the VULCAN system [92]. In this system software synthesis is performed in conjunction with hardware/software partitioning: as a starting point *program threads* are extracted from the system specification, composed of concurrent processes. This step is done in order to isolate operations with an unknown timing delay (*ND*-operations) at specific places, namely at the beginning of the program threads (program threads are explained in more detail in Section 4). Initially all operations are supposed to be performed in hardware, except for all *ND*-operations; partitioning is then performed by means of an iterative approach which selects operations to be transferred from the hardware to the software partition, in order to minimise the hardware cost, while satisfying the imposed timing constraints. The run-time behaviour of the software partition is controlled by a run-time scheduler which alternates the execution of the program threads in order to achieve the original process concurrency. The run-time scheduler activates the threads based on a simple control-FIFO containing pointers to the threads which are ready to run. The next thread to be executed is pushed on the control-FIFO directly by the threads under execution, while the run-time scheduler simply pops the program thread stored on the top of the control-FIFO and executes it, without performing any additional ordering of the threads which are ready to run. This simple scheduling scheme provides only a restricted support for timing constraints. Moreover interrupts are not supported, due to the choice of using a non-preemptive scheduler.

The CHINOOK hardware-software co-design system also supports software synthesis [41, 40]. The target application domain is *reactive systems*. The specification model is based on an extension of reactive-style synchronous languages, like Esterel [55] and StateCharts [93]. The system behaviour is subdivided into a number of *modes*; mode transitions are caused by the watchdog upon detection of an event. *Safe exit points* can be specified in order to ensure the integrity of the system when exiting from a mode. Scheduling is divided into two levels. At the low level, a combined scheduler/partitioner schedules those operations with constraints on or below the order of the processor instruction cycle time, as meeting these constraints may require both hardware and software. The result is a software device driver (each group of operations that have been scheduled together at the low level appears as a single atomic sequence of software instructions to be scheduled at the high level) and a structural description of the device and its interface logic [39]. High-level scheduling is divided into *intra-modal* scheduling, for determining the execution ordering within each mode, and *inter-modal* scheduling, for the support of timing constraints imposed on the transitions between different modes. A limitation of this approach is its restricted support for interrupts: although preemption is allowed by mode transitions, resuming a mode at the preemption point, after execution of the interrupting code, is difficult to achieve with the watchdog paradigm.

Software synthesis is also included in the hardware-software co-design methodology for reactive real-time applications, presented in [37]. With this approach, system functionality to be mapped on either hardware or software is expressed by means of a unified extended finite state machines formalism, called *Co-design Finite State Machine* (CFSM). The input specifications are composed of a network of interacting CFSMs that communicate by means of *events*. Software synthesis is performed in two steps: (1) transformation of the CFSM specification into an *s-graph*, i.e. a reduced form of the control-flow graph typically used in compiler technology, and (2) translation of the s-graph into portable C code, which is then compiled into the target micro-controller object code. Different from other software synthesis approaches, a small customised operating system, consisting of a scheduler and drivers for the I/O channels, is used to correctly implement the run-time behaviour of the input CFSMs. Both polling and interrupts can be used for implementing the event detection. This approach is limited to reactive, control dominated systems, which are mapped to a micro-controller. Because of the state explosion problem, the size of the systems that can be mapped is also limited.

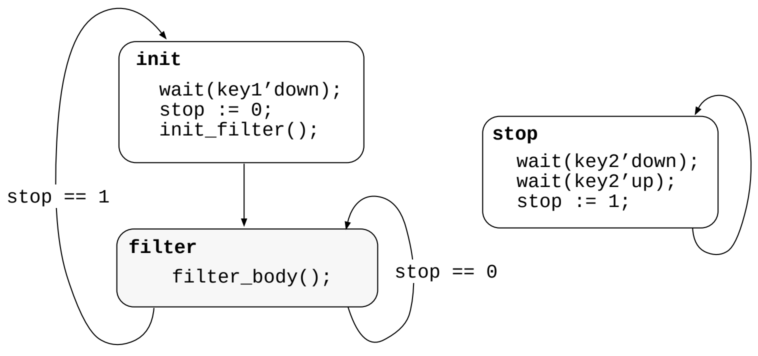
3 SYSTEM SPECIFICATION AND SYNTHESIS SCRIPT

This section discusses system specification styles and gives an overview of our methodology. A more formal discussion is the subject of Section 5.

As already mentioned, advanced information processing systems are characterised by a heterogeneous implementation style, and perform different kinds of functionalities at the same time, such as signal processing and control-intensive tasks. A specification style based on concurrent communicating processes is well suited for this application domain since it allows a better encapsulation of the different functionalities.

Consider for example the simple system described in Figure 1(a). The shaded node `filter` is a *real-time* signal processing task that, once activated, must be repeated at a fixed rate, until the `stop` signal becomes '1'. The `init` task waits for `key1` to be pressed, and then initialises and activates the `filter` task. Since the initialisation and the filter tasks must be executed one after the other, a sequential specification is adequate for these functionalities. However, as illustrated in the `stop` task, the computation of the stop condition (press and then release `key2`) requires an execution time which is unknown at compile time; therefore, if the original specification is not changed, it is not possible to check for the stop condition at each iteration of the filter body, because this would introduce an *unbounded delay* in the filter body, in contrast with its real time constraint. A concurrent specification style, as shown in Figure 1 (b), is therefore required in order to capture the original system functionality. An alternative approach is to transform the original system specification into a sequence of tasks, but as shown in Figure 1 (c) this approach has several drawbacks: the user must transform the original specification in order to avoid unbounded delays, often resulting in an *over-specification*. Additionally, he must manually interleave and statically schedule the different tasks, which is tedious, dangerous (because state explosion can occur) and difficult (because timing constraints must be satisfied and kept consistent). In general, these steps towards the implementation are not desirable at this level of abstraction.

In order to support the different requirements of concurrent processes with real-time constraints, several transformations must be applied to the original specification. In Figure 2, our software synthesis script is depicted. The *input specification*, composed of concurrent communicating processes with real-time constraints, is first translated into a set of program threads [92], which allow a better static analysis. Specifically, *non-deterministic* operations (i.e. operations



(a)

Concurrent Specification

```
Process init
{
 wait(key1'down);
 stop := 0;
 init_filter();
 go := 1;
}

Process filter
{
 wait(go==1);
 while(stop==0) loop
     filter_body();
 end loop;
}

Process stop
{
 wait(key2'down);
 wait(key2'up);
 stop := 1;
}
```

(b)

Sequential Specification

```
{
 while(TRUE) loop
     key2_down := FALSE;
     key2_up   := TRUE;

     wait(key1'down);
     init_filter();

 L1: while(TRUE) loop
         filter_body();
         if (key2'down) then
             key2_down := TRUE;
         endif;
         if (key2_down && key2'up) then
             exit L1;
         endif;
     end loop;
 end loop;
}
```

(c)

Figure 1 Digital filter with user interface process : (a) system description, (b) concurrent and (c) sequential specification style.

with an unbounded timing delay), such as "wait for an event to occur", are allowed only at the beginning of a program thread, so that apart from the first operation, for a given programmable device, the latency of each thread can be determined statically by calling a code generator. A representation model, based on *constraint graphs* [121], is then extracted from the program threads. By means of a static analysis of the constraint graphs, program threads

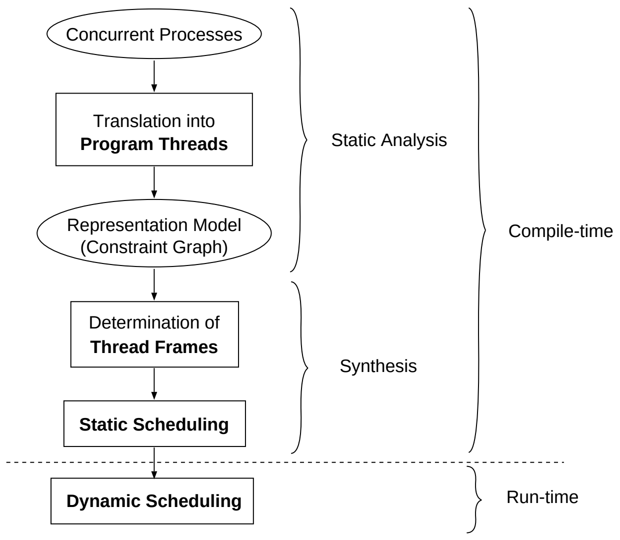
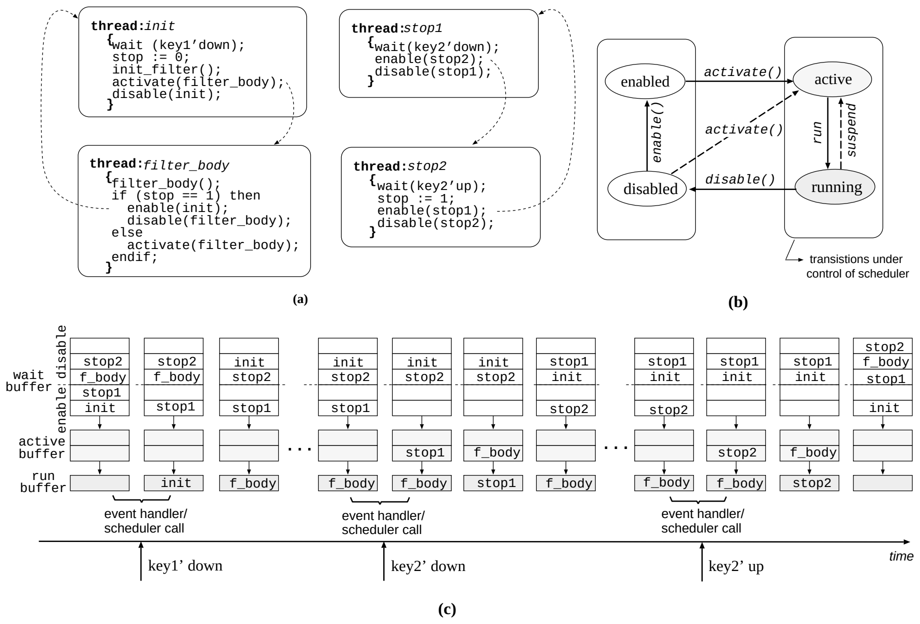

Concurrent Processes
Translation into **Program Threads**
Representation Model (Constraint Graph)
Determination of **Thread Frames**
Static Scheduling
Dynamic Scheduling
Static Analysis
Synthesis
Compile-time
Run-time

Figure 2 Software synthesis script.

which are triggered by the same non-deterministic operation are clustered into *thread frames*. Static scheduling is then performed for determining the relative ordering of the threads belonging to the same thread frame. Based on the imposed timing constraints and on the relative thread ordering within each frame, the time *slack* of each thread is determined, indicating the amount of time the end of a thread can be postponed, relatively to its static schedule, before violating a timing constraint. This static information is finally used at run-time by the *dynamic scheduler*, whose purpose is to combine the different thread frames according to the system evolution. In the sequel, each step of the script is described in more details.

4 PROGRAM THREADS

Non-deterministic (*ND*) operations are operations whose execution delay is unknown at compile time. Examples are synchronisation with internal or external events, such as `wait(key1'down)` in Figure 1, unbounded loops, such as `filter`, and others. The purpose of extracting program threads from concurrent processes is to isolate all the uncertainties, related to the execution delay of a given program, at the beginning of the program threads. At run-



```
thread:init
  {wait (key1'down);
   stop := 0;
   init_filter();
   activate(filter_body);
   disable(init);
  }

thread:filter_body
  {filter_body();
   if (stop == 1) then
     enable(init);
     disable(filter_body);
   else
     activate(filter_body);
   endif;
  }

thread:stop1
  {wait(key2'down);
   enable(stop2);
   disable(stop1);
  }

thread:stop2
  {wait(key2'up);
   stop := 1;
   enable(stop1);
   disable(stop2);
  }
```

Figure 3 (a) Program threads, (b) state transitions, (c) time execution trace.

time, a dynamic scheduler activates the different program threads according to the original specification while taking into account the sequence of occurred events and the imposed timing constraints. In this way the unknown delay in executing a program thread appears as a delay in scheduling the program thread, and is not considered as part of the thread latency [92]. Besides, process concurrency can be achieved by alternating the execution of the different threads.

Compared to the execution model of the VULCAN system [92], our approach is more sophisticated, since it takes into account run-time thread scheduling driven by *timing constraints*. Furthermore, the execution model allows for *preemption*, and *interrupts* are supported in the event detection mechanism, as explained in Section 6.2.

As an example consider the set of program threads illustrated in Figure 3(a), derived from the concurrent specification of Figure 1(b). Four program threads have been extracted from the three concurrent processes, since the `stop` process has been decomposed into two program threads (`stop1` and `stop2`), one for each *ND*-operation (`wait` statements). As illustrated in Figure 3(b), a program thread can be in one of four different states:

- *Disabled*, indicating a state of inactivity.
- *Enabled*, i.e. waiting for a specific event to occur.
- *Active*, indicating that it is ready to run.
- *Running*, i.e. under execution.

Threads in the disabled or enabled states are stored into the `wait buffer`, while threads in the active and running state are stored respectively in the `active buffer` and `run buffer`. In general, state transitions are caused by internal or external *events*, except for the transition from the active to the run state, which is triggered by the dynamic scheduler. In case of an internal event, the thread which is executing and which generates the actual event, can cause another thread to become active, and thus runnable. When the thread under execution terminates, it is set back into the enable state (ready to be reactivated by the occurrence of its event) or to the disable state (when a dependency relation exists between this thread and a preceding thread). Remark that thread dependencies can be built into the code using `enable()` or `activate()` commands. The execution order of the threads in the active state is determined by the dynamic scheduler, and it is driven by the imposed timing constraints, as illustrated in Section 6.2.

Figure 3(c) illustrates the run-time evolution of the wait and active buffers for the example of Figure 3(a): at startup the `init` and `stop1` threads are in the enabled state, when the event `key1'down` occurs. This event causes the `init` thread to be set in the active state, and then to be executed. As a consequence, `filter_body` is set in the active state by the `init` thread, while `init` disables itself. The `filter_body` is then executed repeatedly, until the event `key2'down` occurs, which causes an interruption of the currently executing thread by invoking the event handler and the scheduler. An immediate transition of `stop1` from the enabled to the active state is made and then the scheduler decides which thread to execute next between `filter_body` and `stop1`. In this case the `filter_body` is selected, so that its execution is resumed starting from the same point where it was interrupted by the scheduler at the occurrence of the event `key2'down`. When `filter_body` terminates, `stop1` is executed, and consequently `stop2` is enabled, while `stop1` is disabled. `filter_body` is then run again until a third event, `key2'up`, occurs, causing `stop2` to be set into the active state. After `filter_body` has been resumed and executed, `stop2` is run, causing the `stop` signal to be set to '1', `stop1` to be enabled again, and `stop2` to be disabled. Finally, `filter_body` is run again, and since the stop condition is true, `init` is enabled, while `filter_body` is disabled, so that the system returns to its initial state.

This example nicely illustrates how the concurrency of the `filter` and the `stop` process present in the original specification of Figure 1 can be simulated by interleaving of thread frames.

In the previous example the decomposition of the original specifications into program threads was trivial. However, in general, different choices are available. For example, branches in the control flow introduce non-determinism in the execution delay of a program; in this case different possibilities are available, like for example making each branch a thread on its own. Other choices are available when taking into account inter-process communication issues, where the decomposition into program threads may have an impact on the size of inter-process communication buffers. The execution time of the program threads also has a big impact on the execution model explained in the following sections, since shorter threads allow faster reaction times at the cost of additional scheduling overhead.

In general, due to the structure of the flow of control and to optimisation issues, such as the minimisation of inter-process communication buffers and system reaction time, several threads are extracted from each single process, each of them *not* necessarily starting with a *ND*-operation. Although a more detailed analysis would be required, in this chapter we do not deal with these issues, since we concentrate more on the representation and execution models.

5 REPRESENTATION MODEL

5.1 Constraint graph

Our representation model is based on *Constraint Graphs* (CG) [121], with **vertices** representing program threads and **edges** the precedence relationships and the timing constraints between threads.

Specifically, let $\delta(v_i)$ be the execution delay of the thread represented by vertex v_i. A forward edge $e_{i,j}$ with weight $w_{i,j}$ represents a *minimum* timing constraint between v_i and v_j, i.e. the requirement that the start time of v_j must occur *at least* $w_{i,j}$ units of time later than the start time of v_i. For simplicity we do not deal with forward edges $e_{i,j}$ for which $w_{i,j} \neq \delta(v_i)$. Similarly, a *maximum* timing constraint between two threads v_i and v_j is indicated as an edge from v_j to v_i with negative weight $w_{i,j}$, representing the requirement that the end time of v_j must occur no later than $\mid w_{i,j} \mid$ units of time later than the end time of v_i. Edges with a non-negative weight are called *forward* edges, edges with a negative weight *backward* edges. As in [120], we assume that the subgraph containing only forward edges is acyclic.

Finally, *ND*-operations are represented by separate **event nodes**. We assume that the arrival time of an event is unknown at compile time, and this is reflected in the specification model by associating a non-deterministic delay to event nodes. This assumption makes event nodes very similar to the *anchors* in relative scheduling [121]; however in our model event nodes must not have any predecessor. As a consequence the CG can be disconnected. All the timing attributes of an event, such as the inter-arrival period for periodic events or the minimum inter-arrival time for asynchronous events, are attached to event nodes. This feature is very important because it decouples the timing characteristics of any particular functionality from the specification style used. Different from relative scheduling, where conflicts in resource access are supposed to be solved before scheduling (by inserting forward edges between conflicting operations in the CG), in our model threads may be conflicting with each other (in most cases we are using a single processor), even if no edges exist between them. This is due to the fact that most threads compete for the main resource, namely the CPU; however, concurrent threads mapped to e.g. the processor peripherals do not conflict with the ones competing for the CPU.

Given a CG $G(V, E)$, in the following text we will denote the set of forward edges as $F \subseteq E$, the set of backward edges as $B \subseteq E$, and the set of event nodes as $T \subseteq V$.

5.2 Thread frames

As already noticed, all the uncertainties related to the timing behaviour of a system specification are captured by event nodes. Since the arrival time of an event is unknown at compile time, event nodes limit the amount of analysis and synthesis which can be performed statically. The purpose of identifying *thread frames* is to partition the initial constraint graphs into disjoint clusters of threads triggered by a single event, so that analysis and synthesis (e.g. scheduling) can be performed for each cluster *relatively* to the associated event. Similarly to the *anchor sets* defined for constraint graphs in [121], the *event set*

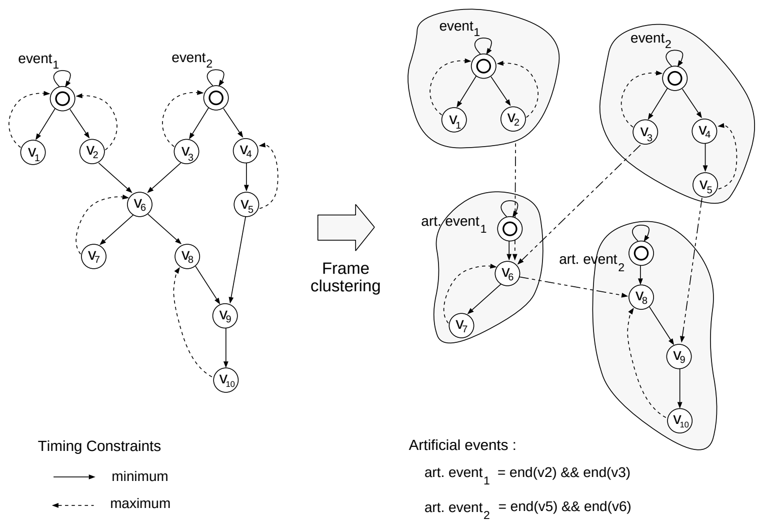


Figure 4 Thread clustering.

$E(v_i)$ of a node v_i is defined as the set of event nodes which are predecessors of v_i.

Definition 15.1 (Thread frame) *Given a CG $G(V, E)$ and an event node $v_k \in T$, a thread frame TF_k is defined as a set of nodes which are triggered only by event v_k, i.e. :*

$$TF_k = \{v_i \in V \mid E(v_i) = \{v_k\}\} \cup \{v_k\}$$

From the above definition it follows that : (1) thread frames are mutually disjoint, and (2) nodes with multiple events in their event set do not belong to any thread frame. In order to increase the amount of static analysis which can be performed, *all* the nodes in the CG are clustered into thread frames by introducing *artificial* events, as shown in Figure 4, without changing the original timing semantics. Besides, since events introduce an overhead during thread scheduling (see next sections), we also want to minimise the number of clusters. More formally the *clustering* problem can be formulated as follows:

Definition 15.2 (Immediate predecessors) *Let $pred(v_i)$ be the set of immediate predecessors of node v_i, i.e.*
$pred(v_i) = \{v_j \mid \exists e_{j,i} \in F\}$. *Given a node cluster $C_k \subset V$, let*

$$pred(C_k) = \{v_i \in V \mid \exists v_j \in C_k : v_i \in pred(v_j) \text{ and } v_i \notin C_k\}$$

be the set of immediate predecessors of a cluster C_k.

Definition 15.3 (Thread clustering) *Given a CG $G(V, E)$, find the clustering $\{C_1, ..., C_n\}$ with minimum n, such that $\forall i \neq j : C_i \cap C_j = \emptyset$, $\bigcup_{i=1}^{n} C_i = V$, and $\forall C_i$ and $\forall v \in C_i$, one of the following two conditions holds: (1) $pred(v) \subset C_i$ or (2) $pred(v) \equiv pred(C_i)$.*

As it is easy to verify, the first conditions guarantee that a complete clustering is performed and that all clusters are disjoint, while the last two conditions guarantee that the threads depend on the same predecessors as their cluster C_i. This is necessary to ensure that the clustering does not implicitly introduce additional dependencies between the threads in the cluster C_i and threads in another cluster; dependencies which were not initially present in the original CG.

A simple approach for thread clustering is to start with a separate cluster for each thread, and then iteratively merge all clusters C_i and C_j such that $pred(C_j) \subseteq C_i$ (condition 1), or $pred(C_j) \equiv pred(C_i)$ (condition 2), until no further merging is possible. Once thread clustering has been performed, thread frames are easily constructed by just inserting in each cluster C_i an event node representing the termination of all the threads contained in $pred(C_i)$ and linking it by means of forward edges to all the vertices $v_j \in C_i \mid \exists v_k \in pred(v_j) : v_k \in pred(C_i)$.

5.3 Well posed-ness

The concept of *well posed-ness* has been introduced in [121] for indicating the consistency of a CG with respect to the timing constraints. Given the similarity between anchors and event nodes, the same condition can be checked also in our specification model. Specifically, a timing constraint is *feasible*, if it can be satisfied when the delays of all events (i.e. the ND-operations) in the CG are set to zero. Then a feasible timing constraint is *well-posed* if it can be satisfied for all values of the unbounded delays.

Given a CG $G(V, E)$, let $E(v_i)$ be the *event set* of vertex $v_i \in V$, defined as the set of *events* which are predecessors of v_i; a necessary condition for G to be well-posed is that for each edge $e_{i,j}$ $E(v_i) \subseteq E(v_j)$. However, in order to effectively isolate thread frames from each other, a more strict condition must be imposed on CGs. Specifically, a necessary condition for a CG to be *strictly well posed* is that, after thread clustering, for all backward edges $e_{i,j}$, both v_i and v_j belong to the same thread frame and do not cross frame boundaries.

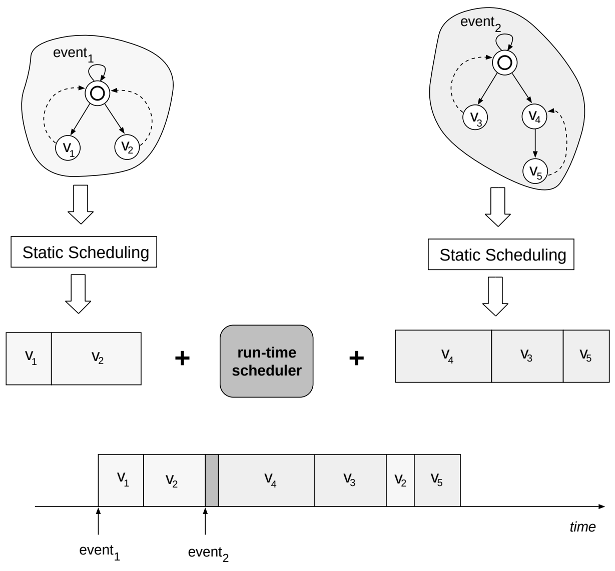


Figure 5 The execution model.

6 EXECUTION MODEL

Figure 5 illustrates the execution model of the proposed software synthesis approach. After thread clustering, all threads in a thread frames are scheduled statically (i.e. at compile time). The thread clustering procedure illustrated in the previous section, together with the condition of strict well posed-ness, guarantee that all frames are isolated (i.e. do not overlap) from each other. Static scheduling is performed for each frame individually, resulting in a relative ordering between the threads in the same frame, and this ordering is not changed anymore in the following phases.

The run time behaviour is illustrated in the lower part of Figure 5: starting from an idle state, suppose that *event 1* occurs; this event activates the run-time scheduler, and since no other frames are currently active, the threads of the first frame are executed in the order determined previously with static scheduling. After some time, while executing thread 2, suppose that event 2 occurs (e.g. by means of an interrupt). This second event causes the following actions :

1. Thread 2 is interrupted.

2. The run-time scheduler is invoked for determining the following execution order (in the example (2, C, 3, A, B)).
3. Execution proceeds with the newly determined thread ordering.

It is important to note that the *relative ordering* between the threads of the same frame, determined at compile time by the static scheduler, is not changed by the run-time scheduler. As illustrated in the sequel, this characteristic is essential for an efficient implementation of the run-time scheduler, which must be necessarily very fast.

6.1 Static scheduling

The relative ordering among the threads within the same frames is determined during *static scheduling.* In this section we do not illustrate a specific scheduling algorithm; instead we indicate the relevant information of the static frame schedule, used at run-time by the dynamic scheduler.

In the following text the start time of a thread v_i is denoted by $T(v_i)$, its end time by $F(v_i)$, and its execution delay by $\delta(v_i)$.

Definition 15.4 (Ordering) *Given a CG $G(V, E)$ and a thread frame $TF \subseteq V$ containing $|TF|$ threads, let O_{TF} be an* ordering *function :*

$$O_{TF} : TF \rightarrow [1, |TF|]$$

such that $\forall v_i \neq v_j$: $O_{TF}(v_i) \neq O_{TF}(v_j)$, indicating that under the ordering O_{TF}, $O_{TF}(v_i) < O_{TF}(v_j)$ implies that v_i is executed before v_j.

As already noticed, the weight $w_{i,j}$ of any forward edge between two threads v_i and v_j is supposed to be always equal to the execution delay of v_i, i.e. $w_{i,j} = \delta(v_i)$. It follows that, given an ordering O_{TF}, assuming that each thread is activated immediately after the other, the *start time* of each thread $v_i \in TF$, relative to the start time of TF, is given by:

$$T_{O_{TF}}(v_i) = \sum_{v_k \in TF | O_{TF}(v_k) < O_{TF}(v_i)} \delta(v_k).$$

In particular, the *time distance* between two threads $v_i, v_j \in TF$ can be determined as follows:

$$TD_{O_{TF}}(v_i, v_j) = \sum_{v_k \in TF | O_{TF}(v_i) \leq O_{TF}(v_k) < O_{TF}(v_j)} \delta(v_k).$$

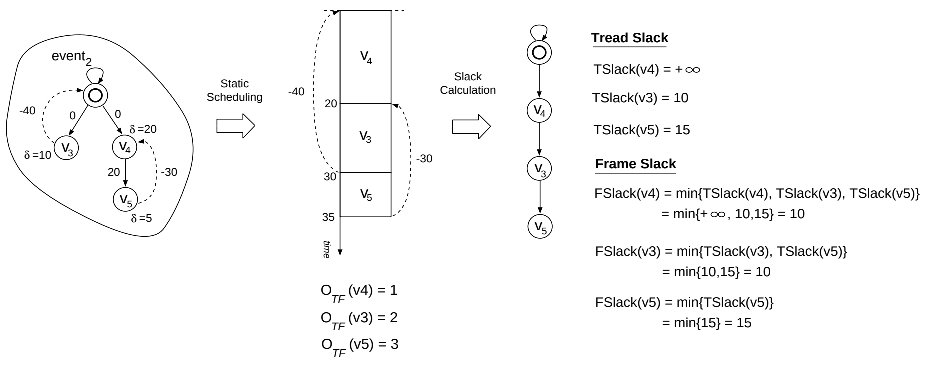


Figure 6 Static scheduling and slack calculation.

Given a thread frame TF, a *valid* ordering O_{TF} is such that all the timing constraints specified *within* TF are satisfied. By means of the time distance function defined above, this condition can be formalised as follows: O_{TF} is a *valid* ordering iff $\forall w_{i,j} \mid v_i, v_j \in TF$,

$$\begin{cases} TD_{O_{TF}}(v_i, v_j) \geq w_{i,j} & \textit{if } w_{i,j} \geq 0 \\ TD_{O_{TF}}(v_j, v_i) + \delta(v_i) - \delta(v_j) \leq \mid w_{i,j} \mid & \textit{otherwise} \end{cases}$$

Definition 15.5 (Thread slack) *Given a thread frame TF and a valid ordering O_{TF}, the thread slack function $TSlack$ of a thread $v_i \in TF$ is defined as follows:*

$$TSlack_{O_{TF}}(v_i) = \begin{cases} \min_{\forall v_j \in TF : w_{i,j} < 0}\{ \ \mid w_{i,j} \mid & \textit{if } \exists v_j \in TF \mid w_{i,j} < 0 \\ -\delta(v_i) + \delta(v_j) - TD_{O_{TF}}(v_j, v_i) \ \} & \\ +\infty & \textit{otherwise} \end{cases}$$

Definition 15.6 (Frame slack) *Given a thread frame TF and a valid ordering O_{TF}, the frame slack function $FSlack$ of a thread $v_i \in TF$ is defined as follows:*

$$FSlack_{O_{TF}}(v_i) = min\{TSlack_{O_{TF}}(v_k) \mid O_{TF}(v_k) \geq O_{TF}(v_i)\}.$$

In Figure 6 a simple example of static scheduling with the related ordering information is illustrated. The $FSlack$ function defined above is very useful for determining the time available for alternating the execution of different thread frames in a multi-tasking environment. Specifically, given a thread frame TF

```
static ThreadFrame $CTF$;
static ThreadOrdering $O_{CTF}$;
DynamicScheduler(Thread $\tilde{v}$, ThreadFrame $NTF$,
                 ThreadOrdering $O_{NTF}$) {
1.    $CTF$ = $\{v_i \in CTF \mid O_{CTF}(v_i) \geq O_{CTF}(\tilde{v})\}$;
2.    $O_{CTF}$ = composeOrderings($CTF$, $O_{CTF}$, $NTF$, $O_{NTF}$);
3.    $CTF$ = $CTF \cup NTF$;
}
```

Figure 7 The dynamic scheduler.

and an ordering O_{TF}, the end time of a thread $v_i \in TF$ can be delayed up to $FSlack_{O_{TF}}(v_i)$ time units with respect to $T_{O_{TF}}(v_i) + \delta(v_i)$ without changing the ordering function and without violating any timing constraint.

6.2 Dynamic scheduling

Starting from statically scheduled frames, the dynamic (or run-time) scheduler determines the thread ordering among different frames which are active at the same time. The run-time composition of different frames is needed since we assume that the arrival time of events is unknown at compile time. The task of dynamic scheduling is simplified by the assumption that the relative ordering between threads of the same frame is not changed, so that the information determined statically (e.g. the $FSlack$ functions) can be exploited effectively in order to simplify the dynamic scheduling problem.

Each time an event occurs, the run-time scheduler is activated, e.g. by implementing it as an interrupt routine, with the following input data :

- The current thread frame under execution at the moment of the event occurrence, denoted by CTF, together with the associated static information, i.e. O_{CTF} and $FSlack_{O_{CTF}}(v_i)$ for all threads $v_i \in CTF$.
- The thread frame triggered by the new event, denoted by NTF, together with the associated static information, i.e. O_{NTF} and $FSlack_{O_{NTF}}(v_i)$ for all threads $v_i \in NTF$.
- The current thread under execution at the moment of the event occurrence, denoted by $\tilde{v} \in CTF$.

The outline of the dynamic scheduler is illustrated in Figure 7. The current thread frame CTF and its relative ordering function O_{CTF} are represented as *static variables* which are retained between successive calls of the scheduler, while the current thread $\tilde{v}$ and the new thread frame, with the associated static ordering, are passed to the scheduler as input parameters. In step 1 the thread frame consisting of the current thread and all the successive ones is constructed, since already executed threads do not have to be taken into account. The heart of the scheduler is the `composeOrderings` function which returns the new thread ordering for the composition of the current and the new frames and which updates the slacks (by decreasing them with the amount by which the threads are effectively delayed). Finally, in step 3 the current frame is updated in order to include also the threads of the new frame. After calling the dynamic scheduler, execution may proceed according to the new ordering stored in O_{CTF}.

The composition of the orderings performed by the `composeOrderings` function must be such that all the timing constraints are still satisfied, or equivalently, after dynamic scheduling, the current ordering must be such that $\forall v_i \in CTF$, $FSlack_{OCTF}(v_i) \geq 0$. Situations may exist in which such a condition can not be satisfied. This may be due to an inappropriate scheduling algorithm, but also because the system can be overloaded. A useful feature of our approach is that when the slack of a thread becomes smaller than a given threshold, event detection mechanisms can be disabled (e.g. disable interrupts) for the duration of the thread, in order to guarantee the timing constraints of the functionalities already in progress. It is important to note that we assume that the scheduler does not know the exact time at which an event occurs. In fact such a knowledge would require the use of a hardware timer, and we do not want to rely on this assumption. It follows that if the current thread $\tilde{v}$ is interrupted by an event, the scheduler must assume the worst case, i.e. that the event occurred just at the beginning of $\tilde{v}$. This drawback of the approach may cause inefficiencies, since valid orderings may be discarded because of the above approximation. However this effect can be reduced by limiting the length of the threads. Finally note that, since slacks are computed relatively to the thread ends, their computation is not affected by the above approximation.

7 CONCLUSIONS

In this paper we have presented a novel approach for software synthesis targeted to real-time information processing systems. Real-time information processing systems are characterised by the coexistence of both signal processing and control functionalities, with different kind of timing constraints. The proposed approach, based on a representation model composed of program threads and constraint graphs, decouples the timing characteristics of any particular functionality from the specification style used, and is therefore well suited for heterogeneous specifications. A general and flexible execution model combines a detailed static analysis of the input specifications, resulting in a static partitioning of the input specifications and a static schedule for each partition, with a run-time scheduler for the dynamic composition of the specification partitions. Starting from the general framework presented in this paper, future work will focus on the different optimisation issues involved in the translation of the input specifications into program threads and in the definition of efficient static and dynamic scheduling algorithms.

Acknowledgement

The work described in this chapter has been supported by the European Commission, under contract *Esprit-9138 (Chips)*.

REFERENCES

[1] "*80C196KC User's Guide*", Document No. 270704–003, Intel Corp., 1990.

[2] "*ADSP2101/2102 user's manual (architecture)*", Analog Devices, 1991.

[3] A. Aho, M. Corasick, "Efficient string matching : an aid to bibliographic search", *Commun. ACM*, Vol. 18, No. 6, June 1975, pp. 333–340.

[4] A.V. Aho, S.C. Johnson, "Optimal code generation for expression trees", *J. of the ACM*, Vol. 23, No. 3, July 1976, pp. 488–501.

[5] A.V. Aho, S.C. Johnson, J.D. Ullman, "Code generation for expressions with common subexpressions", *J. ACM*, Vol. 24, No. 1, Jan. 1977, pp. 146–160.

[6] A.V. Aho, R. Sethi, J.D. Ullman, "*Compilers – principles, techniques, and tools*", Addison-Wesley, Reading (Mass., U.S.A.), 1986.

[7] A.V. Aho, M. Ganapathi, S.W.K. Tjiang, "Code generation using tree matching and dynamic programming", *ACM Trans. Prog. Lang. and Systems*, Vol. 11, No. 4, Oct. 1989, pp. 491–516.

[8] C. Albrecht, S. Bashford, P. Marwedel, A. Neumann, W. Schenk, "The design of the PRIPS microprocessor", *Proc. 4th Eurochip Workshop on VLSI Training*, Toledo (Spain), Sept. 1993, pp. 254–259.

[9] A. Alomary, T. Nakata, Y. Honma, M. Imai, N. Hikichi, "An ASIP instruction set optimization algorithm with functional module sharing constraint", *Proc. IEEE/ACM Int. Conf. Comp.-Aided Design*, Santa Clara (Calif., U.S.A.), Nov. 1993, pp. 526–532.

[10] F. Anceau, P. Liddell, J. Mermet, C. Payan, "CASSANDRE : a language to describe digital systems", *Proc. 3rd Symp. on Comp. and Inform. Sciences*, 1969, pp. 179–204.

[11] F. Anceau, "FORCE : a formal checker for executability", in : D. Borrione (ed.), "*From HDL descriptions to guaranteed correct circuit designs*", North Holland, 1986.

[12] P.J. Ashenden, "The VHDL cookbook", *Techn. Report*, Dept. of Comp. Science, Univ. of Adelaide, Adelaide (Australia), 1990.

[13] T. Baba, H. Hagiwara, "The MPG system : a machine-independent efficient microprogram generator", *IEEE Trans. Computers*, Vol. C-30, June 1981, pp. 373–395.

[14] T. Baba, H. Minakawa, K. Okuda, "A visual microprogramming system", *ACM*, 1987, pp. 23–30.

[15] M.R. Barbacci, "Instruction set processor specifications (ISPS) : the notation and its applications", *IEEE Trans. Computers*, Vol. C-30, No. 1, Jan. 1981, pp. 24–40.

[16] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, D. Voggenauer, "The MIMOLA language – version 4.1", *Techn. Report*, Comp. Science Dept., Univ. of Dortmund, Dortmund (Germany), Sept. 1994.

[17] K. Baudendistel, J.H. McClellan, "Code generation for the AT&T DSP32", *Proc. IEEE Int. Conf. Acoustics, Speech, Signal Proc.*, April 1990, pp. 1073–1076.

[18] R. Beckmann, P. Marwedel, D. Pusch, and W. Schenk, "The MIMOLA language reference manual – version 4.0 – 2nd edition", *Techn. Report No. 401*, Comp. Science Dept., Univ. of Dortmund, Dortmund (Germany), Feb. 1992.

[19] R. Beckmann, U. Bieker, I. Markhof, "Application of constraint logic programming for VLSI CAD tools", *Proc. 1st Int. Conf. Constraints in Comput. Logic*, Munich (Germany), Sept. 1994, pp. 183–200.

[20] C.G. Bell, A. Newell, "*Computer structures : readings and examples*", MacGraw-Hill, New York (New York, U.S.A.), 1971.

[21] F. Benhamou, A. Colmerauer (ed.), "*Constraint logic programming : selected research*", MIT Press, Cambridge (Mass., U.S.A.), 1993.

[22] D. Bernstein, M. Rodeh, "Global instruction scheduling for superscalar machines", *Proc. SIGPLAN Conf. Prog. Lang. Design and Implement.*, June 1991, pp. 241–255.

[23] D. Berson, R. Gupta, M.L. Soffa, "Resource spackling : a framework for integrating register allocation in local and global schedulers", *Proc. Working Conf. Parallel Archit. and Compil. Techn.*, Aug. 1994.

[24] U. Bieker, A. Neumann, "Using logic programming and coroutining for electronic CAD", *Proc. 2nd Int. Conf. Practical Applic. of Prolog*, London (U.K.), April 1994, pp. 67–78.

[25] G. Borriello, "Software scheduling in the co-synthesis of reactive real-time systems", *Proc. 31th ACM/IEEE Design Autom. Conf.*, 1994, pp. 1–4.

[26] J.G. Bosch, G. van Burken, S.S. Schukking, R. Wolff, A.J. van de Goor, J.H.C. Reiber, "Real-time frame-to-frame automatic contour detection on echocardiograms", *Proc. Computers in Cardiology*, 1994.

[27] D.G. Bradlee, S.J. Eggers, R.R. Henry, "Integrating register allocation and instruction scheduling for RISCs", *Int. Conf. Archit. Support for Progr. Lang. and Operating Systems*, April 1991, pp. 122–131.

[28] D.G. Bradlee, "*Retargetable instruction scheduling for pipelined processors*", Ph.D. Thesis, Univ. of Washington, Seattle (Wash., U.S.A.), 1991.

[29] D. Brahme, J. A. Abraham, "Functional testing of microprocessors", *IEEE Trans. Computers*, Vol. C-33, No. 6, June 1984, pp. 475–485.

[30] P. Briggs, K.D. Cooper, L. Torczon, "Rematerialization", *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Implement.*, 1992, pp. 311–321.

[31] J. Bruno, R. Sethi, "Code generation for a one-register machine", *J. ACM*, Vol. 23, No. 3, July 1976, pp. 502–510,

[32] R.E. Bryant, "Graph-based algorithms for boolean function manipulation", *IEEE Trans. Computers*, Vol. 35, No. 8, Aug. 1986, pp. 677–691.

[33] J.P. Calvez, "*Embedded real-time systems*", Wiley Series in Software Engineering Practice, 1993.

[34] R. Camposano, J. Wilberg, "Embedded system design", *Submitted for publication*, 1995.

[35] R.G.G. Cattell, "*Formalization and automatic derivation of code generators*", Ph.D. thesis, Carnegie-Mellon Univ., Pittsburgh (U.S.A.), 1978.

[36] G.J. Chaitin, "Register allocation and spilling via graph coloring", *Proc. ACM SIGPLAN Conf. Progr. Lang. Design and Implement.*, Vol. 17, 1982, pp. 98–105.

[37] M. Chiodo, P. Giusto, A. Jurecska, H. C. Hsieh, A. Sangiovanni-Vincentelli, L. Lavagno, "Hardware-software codesign of embedded systems", *IEEE Micro*, Vol. 14, No. 4, Aug. 1994.

[38] "*CHIP user's guide*", COSYTEC SA, Orsay (France), 1991.

[39] P. Chou, R. Ortega, G. Borriello, "Synthesis of hardware/software interface in the microcontroller-based systems", *Proc. IEEE/ACM Int. Conf. Comp.-Aided Design*, Santa Clara (Calif., U.S.A.), Nov. 1992.

[40] P. Chou, G. Borriello, "Software scheduling in the co-synthesis of reactive real-time systems", *Proc. 31st ACM/IEEE Design Autom. Conf.*, June 1994.

[41] P. Chou, E.A. Walkup, G. Borriello, "Scheduling for reactive real-time systems", *IEEE Micro*, Vol. 14, No. 4, Aug. 1994.

[42] W. F. Clocksin, "Logic programming and digital circuit analysis", *J. Logic Progr.*, March 1987, pp. 59–82.

[43] M.E. Conway, "Proposal for an UNCOL", *Comm. of the ACM*, Vol. 1, 1958.

[44] K. Cools, D. Devisch, K. Van Nieuwenhove, S. Vernalde, I. Bolsens, K. Chansik, O. Younguk, R. Lee, "ASIC synthesis of a flexible 80 Mbit/s Reed-Solomon codec", *Proc. EuroDAC*, Grenoble (France), Sept. 1994, pp. 658–663.

[45] H. Corporaal, H. (J.M.) Mulder, "MOVE : a framework for high-performance processor design, *Proc. Supercomputing*, Albuquerque (New Mex., U.S.A.), Nov. 1991, pp. 692–701.

[46] H. Corporaal, R. Lamberts, "TTA processor synthesis", *Proc. 1st Annual Conf. ASCI*, May 1995.

[47] R.E. Crochiere, A.V. Oppenheim, "Analysis of linear digital networks", *Proc. IEEE*, Vol. 63, No. 4, April 1975, pp. 581–595.

[48] R. Cytron, J. Ferrante, "What's in a name ?", *Proc. Int. Conf. Parallel Processing*, Aug. 1987, pp. 19–27.

[49] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph", *ACM Trans. Prog. Lang. and Systems*, Vol. 13, No. 4, Oct. 1991, pp. 451–490.

[50] H. De Man, J. Rabaey, P. Six, "CATHEDRAL II : a synthesis and module generation system for multiprocessor systems on a chip", in : G. De Micheli, A. Sangiovanni-Vincentelli, P. Antognetti, "*Design systems for VLSI circuits – Logic synthesis and silicon compilation*", Martinus Nijhoff Publ., 1987.

[51] H. De Man, F. Catthoor, G. Goossens, J. Vanhoof, J. Van Meerbergen, S. Note, J. Huisken, "Architecture driven synthesis techniques for VLSI implementation of DSP algorithms", *Proc. of the IEEE*, Vol. 78, No. 2, Feb. 1990, pp. 319–336.

[52] G. De Micheli, "Computer-aided hardware-software codesign", *IEEE Micro Magazine*, Aug. 1994, pp. 10–16.

[53] F. Depuydt, W. Geurts, G. Goossens, H. De Man, "Optimal scheduling and software pipelining of repetitive signal flow graphs with delay line optimization", *Proc. European Design and Test Conf.*, Paris (France), Feb. 1994, pp. 490–494.

[54] F. Depuydt, G. Goossens, H. De Man, "Scheduling with register constraints for DSP architectures", *Integration – the VLSI J.*, Vol. 18, No. 1, Dec. 1994.

[55] R. De Simone, F. Boussinot, "The Esterel language", *Proc. IEEE*, Vol. 79, No. 9, Sep. 1991.

[56] D. Desmet, D. Genin, "ASSYNT : efficient assembly code generation for digital signal processors starting from a data flowgraph", *Proc. IEEE Int. Conf. Acoustics, Speech, Sign. Proc.*, Minneapolis (Minn., U.S.A.), April 1993, pp. 45–48.

[57] P. Dewilde, E. Deprettere, R. Nouta, "Parallel and pipelined VLSI implementation of signal processing algorithms", in : S.Y. Kung, H.J. Whitehouse, T. Kailath (ed.), "*VLSI and modern signal processing*", Prentice-Hall, 1985.

[58] "*DSP Architect – DFL – User's and Reference Manual, Software Version 8.2*", Mentor Graphics Corp., Leuven (Belgium), 1993.

[59] "*ECLIPSE 3.4 user manual*", ECRC Common Logic Programming System, ECRC GmbH, Munich (Germany), 1994.

[60] J.R. Ellis, "BULLDOG *: a compiler for VLIW architectures*", MIT Press, Cambridge (Mass., U.S.A.), 1986.

[61] H. Emmelmann, "Code selection by regular controlled term rewriting", in : G. Giegerich, S.L. Graham (ed.), "*Code generation - concepts, tools, techniques*", Workshops in Computing, Springer, 1992, pp. 3–29.

[62] C.J. Evangelisti, G. Goertzel, H. Ofek, "Using the dataflow analyzer on LCD descriptions of machines to generate control", *Proc. 4th Int. Workshop Hardware Descr. Lang.*, Oct. 1979, pp. 109–115.

[63] C. Ewering, "Automatic high level synthesis of partitioned busses", *Proc. IEEE/ACM Int. Conf. Comp.-Aided Design*, Santa Clara (Calif., U.S.A.), Nov. 1990, pp. 304–307.

[64] H. Farreny, M. Ghallab, "*Éléments d'intelligence artificielle*", Hermès, Paris (France), 1987.

[65] A. Fauth, A. Knoll, "Automated generation of DSP program development tools using a machine description formalism", *Proc. IEEE Int. Conf. Acoustics, Speech, Signal Proc.*, Minneapolis (Minn., U.S.A.), 1993, pp. 457–460.

[66] A. Fauth, A. Knoll, "Translating signal flowcharts into microcode for custom digital signal processors", *Proc. IEEE Int. Conf. Signal Processing*, Beijing (P.R. China), Oct. 1993, pp. 65–68.

[67] A. Fauth, M. Freericks, A. Knoll, "Generation of hardware machine models from instruction set descriptions", *Proc. IEEE Workshop VLSI Signal Proc.*, Veldhoven (Netherlands), Oct. 1993, pp. 242–250.

[68] A. Fauth, G. Hommel, C. Müller, A. Knoll, "Global code selection for directed acyclic graphs", *Proc. ACM Int. Conf. Compiler Construction*, Edinburgh (Scotland, U.K.), April 1994, pp. 128–142.

[69] A. Fauth, J. Van Praet, M. Freericks, "Describing instruction set processors using nML", *Proc. European Design and Test Conf.*, Paris (France), March 1995, pp. 503–507.

[70] A. Fauth, J. Van Praet, M. Freericks, "Describing instruction sets using nML (Extended version)", *Techn. Report*, T.U. Berlin and IMEC, Berlin (Germany)/Leuven (Belgium), 1995.

[71] J.A. Fisher, "Trace scheduling : a technique for global microcode compaction", *IEEE Trans. Computers*, Vol. C-30, No. 7, 1981, pp. 478–490.

[72] C.W. Fraser, R.R. Henry, "BURG — Fast optimal instruction selection and tree parsing", *ACM Sigplan Notices*, Vol. 27, No. 4, April 1992, pp. 68–76.

[73] C.W. Fraser, D.R. Hanson, T.A. Proebsting, "Engineering a simple, efficient code-generator generator", *ACM Letters of Prog. Lang. and Systems*, Vol. 1, No. 3, Sept. 1992, pp. 213–226.

[74] M. Freericks, "The nML machine description formalism", *Techn. Report No. 1991/15*, Comp. Science Dept., T.U. Berlin, Berlin (Germany), 1991.

[75] D. Gajski, F. Vahid, S. Narayan, J. Gong, "*Specification and design of embedded systems*", Prentice Hall, 1994.

[76] M. Ganapathi, C.N. Fisher, J.L. Hennessy, "Retargetable compiler code generation", *ACM Computing Surveys*, Vol. 14, No. 4, Dec. 1982, pp. 573–592.

[77] J.G. Ganssle, "*The art of programming embedded systems*", Academic Press Inc., San Diego (Calif., U.S.A.), 1992.

[78] M.R. Garey, D.S. Johnson, "*Computers and intractability : a guide to the theory of NP-completeness*", W.H. Freeman and Co., 1979.

[79] C. Gebotys, M. Elmasry, "Simultaneous scheduling and allocation for cost constrained optimal architectural synthesis", *Proc. 28th ACM/IEEE Design Autom. Conf.*, 1991, pp. 2–7.

[80] D. Genin, J. De Moortel, D. Desmet, E. Van de Velde, "System design optimization and intelligent code generation for standard digital signal processors", *Proc. IEEE Int. Symp. Circuits and Systems*, Portland (Oreg., U.S.A.), May 1989, pp. 565–569.

[81] K. Ghosh, B. Mukherjee, K. Schwan, "A survey of real-time operating systems", *Techn. Report No. GIT-CC-93/18*, College of Computing, Georgia Inst. of Technology, Atlanta (Georg., U.S.A.), Feb. 1994.

[82] R. Giegerich, S. Graham, "Code generation – concepts, tools, techniques", *Seminar Report No. 9121*, Schloß Dagstuhl (Germany), 1991.

[83] M. Girkar, "*Functional parallelism : theoretical foundations and implementations*", Ph.D. Thesis, Univ. of Illinois, Urbana-Champaign (Ill., U.S.A.), 1991.

[84] M. Girkar, C.D. Polychronopoulos, "Automatic extraction of functional parallelism from ordinary programs", *IEEE Trans. Paral. and Distrib. Systems*, Vol. 3, No. 2, March 1992, pp. 166–178.

[85] R.S. Glanville, "*A machine independent algorithm for code generation and its use in retargetable compilers*", Ph.D. thesis, U.C. Berkeley, Berkeley (Calif., U.S.A.), 1978.

[86] R.S. Glanville, S.L. Graham, "A new method for compiler code generation (Extended abstract)", *Conf. Record 5th Annual ACM Symp. on Principles of Progr. Lang.*, 1978, pp. 231–240.

[87] J.R. Goodman, W. Hsu, "Code scheduling and register allocation in large basic blocks", *Proc. Int. Conf. Supercomputing*, July 1988.

[88] G. Goossens, J. Rabaey, J. Vandewalle, H. De Man, "An efficient microcode compiler for application-specific DSP-processors", *IEEE Trans. on Comp.-Aided Design*, Vol. 9, No. 9, Sept. 1990, pp. 925–937.

[89] G. Goossens, I. Bolsens, B. Lin, F. Catthoor, "Design of heterogeneous ICs for mobile and personal communication systems", *Proc. IEEE/ACM Int. Conf. Comp.-Aided Design*, San Jose (Calif., U.S.A.), Nov. 1994, pp. 524–531.

[90] G. Goossens, D. Lanneer, M. Pauwels, F. Depuydt, K. Schoofs, A. Kifli, M. Cornero, P. Petroni, F. Catthoor, H. De Man, "Integration of medium-throughput signal processing algorithms on flexible instruction-set architectures", *J. VLSI Signal Proc.*, Vol. 9, No. 1, 1995, pp. 49–65.

[91] R. Gupta, M.L. Soffa, "Region scheduling : an approach for detecting and redistributing parallelism", *Trans. Software Eng.*, Vol. 16, No. 4, April 1990.

[92] R.K. Gupta, "*Co-synthesis of hardware and software for digital embedded systems*", Ph.D. Thesis, Stanford Univ., Stanford (Calif., U.S.A.), Dec. 1993.

[93] D. Harel, "StateCharts : a visual formalism for complex systems", *Science of Programming*, Vol. 8, 1987.

[94] R. Hartmann, "Combined scheduling and data routing for programmable ASIC systems", *Proc. European Conf. on Design Autom.,* Brussels (Belgium), March 1992, pp. 486–490.

[95] R. Hartmann, "Synthese von DSP-Algorithmen für flexible Multiprozessorstrukturen", *Ph.D. Thesis*, T.U. München, Munich (Germany), 1993.

[96] W. Heinrich, "*Formal description of parallel computer architectures as a basis of optimizing code generation*", Ph.D. thesis, T.U. München, Munich (Germany), 1993.

[97] L.J. Hendren, G.R. Gao, E.R. Altman, C. Mukerji, "A register allocation framework based on hierarchical cyclic interval graphs", *Proc. ACM Int. Conf. Compiler Construction*, 1992.

[98] J. Henkel, R. Ernst, U. Holtmann, T. Benner, "Adaption of partitioning and high-level synthesis in hardware/software co-synthesis", *Proc. IEEE/ACM Int. Conf. Comp.-Aided Design*, San Jose (Calif., U.S.A.), Nov. 1994. pp. 96–100.

[99] J.L. Hennessy, D.A. Patterson, "*Computer architecture : a quantitative approach*", Morgan Kaufmann Publ. Inc., San Mateo (Calif., U.S.A.), 1990.

[100] G.T. Herman, A. Lindenmayer, G. Rozenberg, "Description of developmental languages using recurrence systems", *Mathem. Systems Theory*, Vol. 4, No. 4, 1974, pp. 316–341.

[101] P. Hilfinger, J. Rabaey, "DSP specification using the SILAGE Language", in : R.W. Brodersen (ed.), "*Anatomy of a silicon compiler*", Kluwer Acad. Publ., Boston (Mass., U.S.A.), 1992.

[102] J. Hoogerbrugge, H. Corporaal, "Transport-triggering vs. operation-triggering", *Proc. ACM Int. Conf. Compiler Construction*, Edinburgh (Scotland, U.K.), 1994.

[103] J. Hoogerbrugge, H. Corporaal, "Register file port requirements of transport triggered architectures", *Proc. 27th Micro*, Santa Clara (Calif., U.S.A.), Dec. 1994.

[104] J. Hoogerbrugge, H. Corporaal, "Automatic synthesis of transport triggered processors", *Proc. 1st Annual Conf. ASCI*, May 1995.

[105] P.W. Horstmann, "*Automation of the design for testability using logic programming*", Ph.D. Thesis, Univ. of Missouri, Oct. 1983.

[106] P.Y.T. Hsu, E.S. Davidson, "Higly concurrent scalar processing", *Proc. ISCA-13*, June 1986, pp. 386–395.

[107] I.-J. Huang, A. Despain, "Generating instruction sets and microarchitectures from applications", *Proc. IEEE/ACM Int. Conf. Comp.-Aided Design*, San Jose (Calif., U.S.A.), Nov. 1994, pp. 391–396.

[108] C.T. Hwang, J.H. Lee, Y.C. Hsu, "A formal approach to the scheduling problem in high level synthesis", *IEEE Trans. Comp.-Aided Design.* Vol. 10, No. 4, 1991, pp. 464–475.

[109] K. Hwang, "*Advanced computer architecture : parallelism, scalability, programmability*", McGraw-Hill, 1993.

[110] W.M.W. Hwu et al, "The superblock : an effective technique for VLIW and superscalar compilation", *J. Supercomputing*, Vol. 7, No. 1/2 May 1993, pp. 229–248.

[111] "*Instruction set details*", DSP development software kit, Motorola Inc., 1992.

[112] R. Jain, K. Kucukcakar, M.J. Mlinar, A.C. Parker, "Experience with the ADAM synthesis system", *Proc. 26th ACM/IEEE Design Autom. Conf.*, Las Vegas (Nev., U.S.A.), 1989.

[113] K. Keutzer, W. Wolf, "Anatomy of a hardware compiler", *Proc. SIGPLAN Conf. Prog. Lang. Design and Implement.*, 1988, pp. 95–104.

[114] A. Kifli, G. Goossens, H. De Man, "A unified scheduling model for high-level synthesis and code generation", *Proc. European Design and Test Conf.*, Paris (France), March 1995, pp. 234–238.

[115] P.M. Kogge, "*The architecture of pipelined computers*", Hemisphere Publishing Corp., 1981.

[116] S. Kohavi, "*Switching and finite automata theory*", McGraw-Hill, 1978.

[117] M. Kozuch, A. Wolfe, "Compression of embedded systems programs", *Proc. IEEE Int. Conf. Computer Design*, Oct. 1994.

[118] G. Krüger, "Automatic generation of self-test programs – a new feature of the MIMOLA design system", *Proc. 23rd ACM/IEEE Design Autom. Conf.*, Las Vegas (Nev., U.S.A.), June 1986, pp. 378–384.

[119] G. Krüger, "A tool for hierarchical test generation", *IEEE Trans. Comp.-Aided Design*, Vol. 10, No. 4, April 1991, pp. 519–524.

[120] D. Ku, G. De Micheli, "HardwareC – a language for hardware design (version 2.0)", *Techn. Report No. CSL-TR-90-419*, Stanford Univ., Stanford (Calif., U.S.A.), Apr. 1990.

[121] D. Ku, G. De Micheli, "Relative scheduling under timing constraints", *Proc. 27th ACM/IEEE Design Autom. Conf.*, Orlando (Flor., U.S.A.), June 1990.

[122] D. Ku, G. De Micheli, "*High level synthesis under timing and synchronization constraints*", Kluwer Acad. Publ., 1992.

[123] D.J. Kuck, Y. Muraoka, S.C. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup", *IEEE Trans. Computers*, Vol. 21, No. 12, Dec. 1972, pp. 1293–1310.

[124] F.J. Kurdahi, A.C. Parker, "REAL : a program for register allocation", *Proc. 24th ACM/IEEE Design Autom. Conf.*, 1987, pp. 210–215.

[125] R. Lamberts, "A move processor generator", *Techn. Report*, Delft Univ. of Technology, Delft (Netherlands), Nov. 1993.

[126] D. Landskov, S. Davidson, B. Shriver, P. Mallett, "Local microcode compaction techniques", *ACM Computing Surveys*, Vol. 12, No. 3, Sept. 1980, pp. 261–294.

[127] B. Landwehr, P. Marwedel, R. Dömer, "OSCAR: optimum simultaneous scheduling, allocation and resource binding based on integer programming", *Proc. EuroDAC*, Sept. 1994, pp. 90–95.

[128] M. Langevin, E. Cerny, "An automata-theoretic approach to local microcode generation", *Proc. European Design and Test Conf.*, Paris (France), 1993, pp. 94–98.

[129] M. Langevin, E. Cerny, "A recursive technique for computing lower-bound performance of schedules", *Proc. IEEE Int. Conf. Computer Design*, Cambridge (Mass., U.S.A.), 1993.

[130] M. Langevin, E. Cerny, "Microcode generation for datapath with multiport memories", *Proc. IFIP Int. Workshop Logic and Archit. Synth.*, Grenoble (France), 1993.

[131] M. Langevin, E. Cerny, "An extended OBDD representation for extended FSM-s", *Proc. European Design and Test Conf.*, Paris (France), 1994, pp. 208–213.

[132] D. Lanneer, G. Goossens, F. Catthoor, M. Pauwels, H. De Man, "An object-oriented framework supporting the full high-level synthesis trajectory", *Proc. Int. Symp. Computer Hardware Descr. Lang.*, Marseille (France), 1991, pp. 281–300.

[133] D. Lanneer, "*Design models and data-path mapping for signal processing architectures*", Ph.D. thesis, K.U. Leuven, Leuven (Belgium), 1993.

[134] M. Lam, "Software pipelining : an effective scheduling technique for VLIW machines", *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Implement.*, Atlanta (Georg., U.S.A.), 1988, pp. 318–328.

[135] D. Lanneer, M. Cornero, G. Goossens, H. De Man, "Data routing : a paradigm for efficient data-path synthesis and code generation", *Proc. 7th ACM/IEEE Int. Symp. on High-Level Synthesis*, Niagara-on-the-Lake (Ont., Canada), May 1994, pp. 17–22.

[136] J.C. Laprie, "*Dependability : basic concepts and terminology*", Springer, Vienna (Austria)/New York (New York, U.S.A.), 1992.

[137] E.A. Lee, D.G. Messerschmitt, "Synchronous data flow", *Proc. IEEE*, Vol. 75, No. 9, Sept. 1987, pp. 1235–1245.

[138] E. Lee, "Programmable DSP architectures, Parts I and II", *IEEE ASSP Magazine*, Oct. 1988, pp. 4–19, and Jan. 1989, pp. 4–14.

[139] J. Lee, J. Patel, "An instruction sequence assembling methodology for testing microprocessors", *Proc. Int. Test Conf.*, Baltimore (U.S.A.), Sept. 1992, pp. 49–58.

[140] C.E. Leiserson, J.B. Saxe, "Optimizing synchronous systems", *Proc. 22nd Annual Symp. Foundations of Comp. Science*", 1981, pp. 23–36.

[141] R. Leupers, W. Schenk, P. Marwedel, "Retargetable assembly code generation by bootstrapping", *Proc. 7th Int. Symp. High-Level Synthesis*, Niagara-on-the-lake (Ont., Canada), May 1994, pp. 88–93.

[142] R. Leupers, W. Schenk, and P. Marwedel, "Microcode generation for flexible parallel target architectures", *Proc. Working Conf. Parallel Archit. and Compil. Techn.*, North-Holland, 1994.

[143] R. Leupers, P. Marwedel, "A BDD-based frontend for retargetable compilers", *Proc. European Design & Test Conf.*, Paris (France), March 1995, pp. 239–243.

[144] S.Y. Liao, S. Devadas, K. Keutzer, "Code density optimization for embedded DSP processors using data compression techniques", *Proc. Chapel Hill Conf. Advanced Research in VLSI*, March 1995.

[145] S.Y. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang, "Storage assignment to decrease code size", *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Implement.*, June 1995.

[146] C. Liem, T. May, P. Paulin, "Instruction-set matching and selection for DSP and ASIP code generation", *Proc. European Design and Test Conf.*, Paris (France), Feb. 1994, pp. 31–37.

[147] C. Liem, T. May, T., P. Paulin, "Register assignment through resource classification for ASIP microcode generation", *Proc. ACM/IEEE Int. Conf. Comp.-Aided Design,* San Jose (Calif., U.S.A.), Nov. 1994, pp. 397–402.

[148] A. Lindenmayer, "Mathematical models for cellular interactions in development", *J. Theoretical Biology*, Vol. 18, 1968, pp. 280–315.

[149] A. Lindenmayer, G. Rozenberg, "*Automata, languages, development*", North Holland, 1976.

[150] F. Löhr, A. Fauth, M. Freericks, "SIGH/SIM – an environment for retargetable instruction set simulation", *Techn. Report No. 1993/43*, Comp. Science Dept., T.U. Berlin, Berlin (Germany), 1993.

[151] S.A. Mahlke et al., "Sentinel scheduling for VLIW and superscalar processors", *Proc. Int. Conf. Archit. Support for Prog. Lang. and Operating Systems*, Boston (Mass., U.S.A.), Sept. 1992, pp. 238–247.

[152] M. Mahmood, F. Mavaddat, M.I. Elmasry, M.H.M. Cheng, "A formal language model of microcode synthesis", in : L.J.M. Claesen (ed.), "*Formal VLSI specification and synthesis : VLSI design methods-I*", North Holland, 1990, pp. 23–41.

[153] M. Mahmood, F. Mavaddat, M.I. Elmasry, "A formal approach to control unit synthesis", *Proc. Working Conf. Logic and Archit. Synthesis*, Paris (France), May 1990, pp. 126–135.

[154] M. Mahmood, "*A formal approach to VLSI control unit and local microcode synthesis*", Ph.D. Thesis, Univ. of Waterloo, Waterloo (Ont., Canada), 1990.

[155] M. Mahmood, F. Mavaddat, M.I. Elmasry, "Experiments with an efficient heuristic algorithm for local microcode generation", *Proc. IEEE Int. Conf. Computer Design*, Cambridge (Mass., U.S.A.), Sept. 1990, pp. 319–323.

[156] Z. Manna, R. Waldinger, "*The deductive foundations of computer programming*", Addison-Wesley, 1993.

[157] M.M. Mano, "*Computer system architecture*", Prentice-Hall, 1993.

[158] P. Marwedel, "A retargetable compiler for a high-level microprogramming language", *Proc. Micro-17*, New Orleans (Louis., U.S.A.), Oct. 1984, pp. 267–274.

[159] P. Marwedel, W. Schenk, "Cooperation of synthesis, retargetable code generation and test generation in the MSS", *Proc. European Conf. on Design Autom.*, Paris (France), Febr. 1993, pp. 63–69.

[160] P. Marwedel, W. Schenk, "Implementation of IF-statements in the TODOS-microarchitecture synthesis system", in : "*Synthesis for control dominated circuits*", North-Holland, 1993, pp. 249–262.

[161] P. Marwedel, "MSSV : tree-based mapping of algorithms to predefined structures", *Proc. IEEE/ACM Int. Conf. Comp.-Aided Design*, Santa Clara (Calif., U.S.A.), Nov. 1993, pp. 586–593.

[162] H. Massalin, "Superoptimizer : a look at the smallest program", *Proc. 2nd Int. Conf. Archit. Support for Prog. Lang. and Operating Systems*, 1987.

[163] F. Mavaddat, "Architecture and layout of a speech recognition chip", *Int. J. Mini and Microcomputers*, Vol. 9, No. 1, 1987, pp. 1–5.

[164] F. Mavaddat, M. Mahmood, "An application of L systems to local microcode synthesis", *Proc. 23rd Annual Workshop and Symp. Microprogr. and Microarchit.*, Orlando (Flor., U.S.A), Nov. 1990, pp. 166–175.

[165] F. Mavaddat, "Designing and modeling VLSI systems at register-transfer level", *Int. J. Comp.-Aided VLSI Design*, Vol. 2, 1990, pp. 281–314.

[166] F. Mavaddat, M. Mahmood, "On compiling behaviour to silicon : a formal language approach", *Integration : the VLSI J.*, Dec. 1991, pp. 239–266.

[167] F. Mavaddat, "Data-path synthesis as grammar inference", in : G. Saucier, J. Trilhe, "*Synthesis for control dominated circuits*", North Holland, 1993.

[168] F. Mavaddat, "Catenation machines and their relation to DT0L systems", *Techn. Report No. CS-91-26*, Univ. of Waterloo, Waterloo (Ont., Canada), Dec. 1991.

[169] J. Mountjoy, M. Beemster, "Functional languages and very fine grained parallelism : initial results", *Techn. Report No. CS-94-24*, Dept. of Comp. Systems, Univ. of Amsterdam, Amsterdam (Netherlands), Dec. 1994.

[170] S. Moon, K. Ebcioglu, "An efficient resource constrained global scheduling technique for superscalar and VLIW processors", *Proc. 25th Annual Int. Symp. Microarchitecture*, Portland (Oreg., U.S.A.), Dec. 1992.

[171] R.A. Mueller, J. Varghese, "Flow graph machine models in microcode synthesis", *Proc. 17th Ann. Workshop on Microprogr.*, 1983, pp. 159–167.

[172] T. Nakatani, K. Ebcioglu, "Using a lookahead window in a compaction-based parallelizing compiler", *Proc. 23rd Annual Int. Symp. Microarchitecture*, 1990.

[173] J. Nestor et al., "MIES : A microarchitecture design tool", *Proc. 22nd Workshop on Microprogr. and Microarch.*, Dublin (Ireland), Aug. 1989.

[174] A. Nicolau, "Uniform parallelism exploitation in ordinary programs", *Proc. Int. Conf. Parallel Processing*, 1985.

[175] A. Nicolau, R. Potasman, "Incremental tree height reduction for high level synthesis", *Proc. 28th ACM/IEEE Design Autom. Conf.*, San Francisco (Calif., U.S.A.), June 1991.

[176] A. Nicolau, R. Potasman, H. Wang, "Register allocation, renaming and their impact on parallelism", *Proc. 4th Int. Workshop on Lang. and Compilers for Paral. Proc.*, 1991.

[177] S. Novack, A. Nicolau, "An efficient global resource constrained technique for exploiting instruction level parallelism", *Proc. Int. Conf. Paral. Processing*, St. Charles (Ill., U.S.A.), Aug. 1992.

[178] S. Novack, A. Nicolau, "Trailblazing : a hierarchical approach to percolation scheduling", *Proc. Int. Conf. Paral. Processing*, St. Charles, (Ill., U.S.A.), Aug. 1993.

[179] S. Novack, A. Nicolau, "A hierarchical approach to instruction-level parallelization", To appear in : *Int. J. Paral. Programming.*

[180] S. Novack, A. Nicolau, N. Dutt, "A unified code generation approach using mutation scheduling", *Techn. Report No. UCI-ICS-94-35*, Dept. of Inform. and Comp. Science, U.C. Irvine, Irvine (Calif., U.S.A.), 1994.

[181] S. Novack, A. Nicolau, "Resource-directed loop pipelining", *Techn. Report No. TR-95-05*, Dept. of Inform. and Comp. Science, U.C. Irvine, Irvine (Calif., U.S.A.), 1995.

[182] L. Nowak, "SAMP : a general purpose processor based on a self-timed VLIW-structure", *ACM Comp. Arch. News*, Vol. 15, 1987, pp. 32–39.

[183] L. Nowak, "Graph based retargetable microcode compilation in the MIMOLA design system", *Proc. Micro-20*, 1987, pp. 126–132.

[184] L. Nowak, P. Marwedel, "Verification of hardware descriptions by retargetable code generation", *Proc. 26th ACM/IEEE Design Autom. Conf.*, Las Vegas (Nev., U.S.A.), June 1989, pp. 441–447.

[185] T. Ohtsuki, "*Layout design and verification*", North-Holland, 1986.

[186] C.A. Papachristou, S.B. Gambhir, "Microcontrol architectures with sequencing firmware and modular microcode development tools", *Microprocessing and Microprogramming 29*, North-Holland, 1991, pp. 303–328.

[187] C.H. Papadimitriou, K. Steiglitz, "*Combinatorial optimization : algorithms and complexity*", Prentice Hall, Englewood Cliffs (New Jers., U.S.A.), 1982.

[188] N. Park, A.C. Parker, "SEHWA : a software package for synthesis of pipelines from behavioral specifications", *IEEE Trans. on Comp.Aided Design of Integr. Circ. and Systems*, vol. CAD-7, No. 3, March 1988, pp. 356–370.

[189] K. Patel, B.C. Smith, L.A. Rowe, "Performance of a software MPEG video decoder", *Proc. Int. Conf. Multimedia*, Anaheim (Calif., U.S.A.), 1993, pp. 75–82.

[190] P.G. Paulin, "*High level synthesis of digital circuits using global scheduling and binding algorithms*", Ph.D. Thesis, Carlton Univ., Ottawa (Ont., Canada), Feb. 1988.

[191] P.G. Paulin, C. Liem, T.C. May, S. Sutarwala, "CodeSyn : a retargetable code synthesis system", *Proc. 7th Int. Symp. High-Level Synthesis*, Niagara-on-the-Lake (Ont., Canada), May 1994.

[192] P.G. Paulin, C. Liem, T.C. May, S. Sutarwala, "DSP design tool requirements for embedded systems : a telecommunications industrial perspective", *J. VLSI Signal Processing*, Vol. 9, No. 1, 1995.

[193] S.S. Pinter, "Register allocation with instruction scheduling : a new approach", *Proc. SIGPLAN Conf. Prog. Lang. Design and Implement.*, June 1993, pp. 248–257.

[194] R. Potasman, "*Percolation-based compiling for evaluation of parallelism and hardware design trade-offs*", Ph.D. thesis, U.C. Irvine, Irvine (Calif., U.S.A.), 1991.

[195] D.B. Powell, E.A. Lee, W.C. Newman, "Direct synthesis of optimized DSP assembly code from signal flow block diagrams", *Proc. IEEE Int. Conf. Acoustics, Speech, Sign. Proc.*, San Francisco (Calif., U.S.A.), March 1992, pp. 553–556.

[196] "*Prolog III reference manual*", PrologIA, Marseille (France), 1991.

[197] J. Rabaey, H. De Man, J. Vanhoof, G. Goossens, F. Catthoor, "Cathedral-II : a synthesis system for multiprocessor DSP systems", in : D. Gajski (ed.), "*Silicon Compilation*", Addison-Wesley, Reading (Mass., U.S.A.), 1988, pp. 311–360.

[198] J.M. Rabaey, M. Potkonjak, "Estimating implementation bounds for real time DSP application specific circuits", *IEEE Trans. Comp.-Aided Design*, Vol. 13, No. 6, June 1994, pp. 669–683.

[199] K.R. Rao, "*Discrete cosine transform*", Academic Press, 1990.

[200] B.R. Rau, J.A. Fisher, "Instruction-level parallel processing : history, overview and perspective", *J. Supercomputing*, Vol. 7, No. 1/2 May 1993, pp. 9–50.

[201] J.F. Ready, "VRTX : a real-time operating system for embedded microprocessor applications", *IEEE Micro*, pp. 8–17, Aug. 1986.

[202] M. Rim, R. Jain, "Representing conditional branches for high-level synthesis applications", *Proc. 29th ACM/IEEE Design Autom. Conf.*, Anaheim (Calif., U.S.A.), June 1992, pp. 106–111.

[203] M. Rim, R. Jain, "Lower-bound performance estimation for the high-level synthesis scheduling problem", *IEEE Trans. Comp.-Aided Design*, Vol. 13, No. 4, Apr. 1994, pp. 451–458.

[204] M. Rim, R. Jain, "RECALS II : a new list scheduling algorithm", *Proc. IEEE Int. Conf. Acoustics, Speech, Signal Proc.*, 1994.

[205] K. Rimey, P.N. Hilfinger, "A compiler for application-specific signal processors", in : R.W. Brodersen, H.S. Moscovitz, "*VLSI Signal Processing, III*", IEEE Press, New York (New York, U.S.A.), 1988, pp. 341–351.

[206] K. Rimey, P.N. Hilfinger, "Lazy data routing and greedy scheduling for application-specific processors", *Proc. 21st Annual Workshop on Microprogr.*, 1988, pp. 111–115.

[207] J.P. Roesgen, "The ADSP-2100 DSP microprocessor", *IEEE Micro*, Dec. 1986, pp. 49–59.

[208] J. Sato, M. Imai, T. Hakata, A.Y. Alomary, N. Hikichi, "An integrated design environment for application-specific integrated processors", *Proc. IEEE Int. Conf. on Computer Design*, Rochester (New York, U.S.A.), Oct. 1991, pp. 414–417.

[209] H.P. Schlaeppi, "A formal language for describing machine logic, timing, and sequences (LOTIS)", *IEEE Trans. Electronic Computers*, Aug. 1964, pp. 439–448.

[210] U. Schmidt, "A new codegenerator-generator based on VDM" *Techn. Report No. 4/83* (in German), Comp. Science Dept., Univ. of Kiel, Kiel (Germany), 1983.

[211] K. Schoofs, G. Goossens, H. De Man, "Bit alignment for retargetable code generators", *Proc. 7th ACM/IEEE Int. Symp. on High-Level Synthesis*, Niagara-on-the-Lake (Ont., Canada), May 1994, pp. 76–81.

[212] R. Sethi, J. Ullman, "The generation of optimal code for arithmetic expressions", *J. ACM*, Vol. 17, No. 4, 1970, pp. 715–728.

[213] C. Shung et al., "An integrated CAD system for algorithm-specific IC design", *IEEE Trans. Comp.-Aided Design*, Vol. 10, No. 4, April 1991, pp. 447–463.

[214] H. Simonis, "Test generation using the constraint logic programming language CHIP", *Proc. 6th Int. Conf. Logic Progr.*, Lisbon (Portugal), June 1989, pp. 101–112.

[215] "SPOX – the DSP operating system", Spectron Microsystems, Santa Barbara (Calif., U.S.A.), June 1992.

[216] "*ST7291 User Manual*", SGS-Thomson Microelectronics, 1993.

[217] "*ST18950 User Manual*", SGS-Thomson Microelectronics, 1993.

[218] R.M. Stallman, "*Using and porting GNU CC, version 2.4*", Free Software Foundation, June 1993.

[219] "*Standard VHDL*", IEEE Standard No. 1076–1987, IEEE, 1987.

[220] "*Standard VHDL language reference manual*", IEEE Design Automation Standards Subcommittee, IEEE Press, New York (New York, U.S.A.), 1988.

[221] "*Standard VHDL language reference manual*", IEEE Design Automation Standards Subcommittee, IEEE Press, New York (New York, U.S.A.), 1992.

[222] J.A. Storer, T.G. Szymanski, "Data compression via textual substitution", *J. ACM*, Vol. 29, No. 4, Oct. 1982, pp. 928–951.

[223] S. Sutarwala, P.G. Paulin, Y. Kumar, "Insulin : an instruction set simulation environment", *Proc. Int. Symp. Computer Hardware Descr. Lang.*, 1993, pp. 355–362.

[224] D. Svanaes, E.J. Aas, "Test generation through logic programming", *Integration – the VLSI J.*, Feb. 1984, pp. 49–67.

[225] S. Takagi, "Rule based synthesis, verification and compensation of data paths", *Proc. IEEE Int. Conf. Computer Design*, 1984, pp. 133–138.

[226] S.M. Thatte, J.A. Abraham, "Test generation for microprocessors", *IEEE Trans. Computers*, Vol. C-29, No. 6, June 1980, pp. 429–441.

[227] M.R. Thistle, B.J. Smith, "A processor architecture for horizon", *Proc. Supercomputing*, Nov. 1988, pp. 35–41.

[228] S.W.K. Tjiang, "An olive twig", *Techn. Report*, Synopsys Inc., 1993 (software available from `ftp://suif.stanford.edu/pub/tjiang`).

[229] "*TMS320C2x user's guide*", Texas Instruments, 1990.

[230] H. Tokuda, C. Mercer, "Arts : a distributed real-time kernel", *Operating Systems Review*, Vol. 23, No. 3, July 1989, pp. 29–53.,

[231] J. Van Praet, G. Goossens, D. Lanneer, H. De Man, "Instruction set definition and instruction selection for ASIPs", *Proc. 7th ACM/IEEE Int. Symp. on High-Level Synthesis*, Niagara-on-the-Lake (Ont., Canada), May 1994, pp. 11–16.

[232] K. Van Rompaey, I. Bolsens, H. De Man, "Just in time scheduling", *Proc. IEEE Int. Conf. Computer Design*, Cambridge (Mass., U.S.A.), Oct. 1992.

[233] S.R. Vegdahl, "*Local code generation and compaction in optimizing microcode compilers*", Ph.D. thesis, Techn. Report No. CMUCS-82-153, Carnegie-Mellon Univ., Pittsburgh (U.S.A.), 1982.

[234] S.R. Vegdahl, "Phase coupling and constant generation in an optimizing microcode compiler", *Proc. 15th Micro*, 1982, pp. 125–133.

[235] E. Verhulst, "Virtuoso : providing sub-microsecond context switching on DSPs with a dedicated nanokernel", *Proc. Int. Conf. Signal Proc. Applic. and Technology*, Santa Clara (Calif., U.S.A.), Sep. 1993.

[236] K. Wakabayashi, H. Tanaka, "Global scheduling independent of control dependencies based on condition vectors", *Proc. 29th ACM/IEEE Design Autom. Conf.*, Anaheim (Calif., U.S.A.), June 1992, pp. 112–115.

[237] R.A. Walker, R. Camposano (ed.), "*A survey of high-level synthesis systems*", Kluwer Acad. Publ., 1991.

[238] B. Wess, "On the optimal code generation for signal flow graph computation", *Proc. IEEE Int. Symp. Circuits and Systems*, New Orleans (Louis., U.S.A.), May 1990, pp. 444–447.

[239] B. Wess, "Automatic code generation for integrated digital signal processors", *Proc. IEEE Int. Symp. Circuits and Systems*, Singapore (Singapore), June 1991, pp. 33–36.

[240] B. Wess, “Automatic instruction code generation based on trellis diagrams”, *Proc. IEEE Int. Symp. Circuits and Systems*, San Diego (Calif., U.S.A.), May 1992, pp. 645–648.

[241] B. Wess, “Optimizing signal flow graph compilers for digital signal processors”, *Proc. Int. Conf. Signal Proc. Applic. and Technology*, Dallas (Tex., U.S.A.), Oct. 1994, pp. 665–670.

[242] N. Weste, “OK, if these CAD tools are so great, why isn’t my chip design on schedule ?”, *Proc. IEEE Int. Conf. Computer Design*, Cambridge (Mass., U.S.A.), 1994, pp. 2–8.

[243] J. Wilberg, R. Camposano, U. Westerholz, U. Steinhausen, “Design of an embedded video compression system – a quantitative approach”, *Proc. IEEE Int. Conf. Computer Design*, Cambridge (Mass., U.S.A.), 1994, pp. 428–431,

[244] J. Wilberg, R. Camposano, M. Langevin, P. Plöger, H.-T. Vierhaus, “Tools for generating a VHDL processor description and a compiler back-end from a single schematic”, *Proc. IFIP Int. Worksh. Logic and Archit. Synth.*, Grenoble (France), 1994.

[245] T. Wilson, A. Basu, D. Banerji, J. Majithia, “Commutative operand alignment for interconnect optimization in behavioural synthesis”, *Int. J. Electronics*, Vol. 73, No. 2, 1992, pp. 417–431.

[246] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, J. Hennessy, “SUIF : a parallelizing and optimizing research compiler”, *Techn. Report No. CSL-TR-94-620*, Stanford Univ., Stanford (Calif., U.S.A.), May 1994 (available from `http://suif.stanford.edu`).

[247] T. Wilson, G. Grewal, B. Halley, D. Banerji, “An integrated approach to retargetable code generation”, *Proc. 7th Int. Symp. on High-Level Synthesis*, Niagara-on-the-Lake (Ont., U.S.A.), May 1994, pp. 70–75.

[248] T. Wilson, N. Mukherjee, M. Garg, D. Banerji, “An ILP solution for optimum scheduling, module and register allocation, and operation binding in datapath synthesis”, To appear in *VLSI Design*, 1995.

[249] A. Wolfe, A. Chanin, “Executing compressed programs on an embedded RISC architecture”, *Proc. Micro-25*, Dec. 1992.

[250] J. Xu, D.L. Parnas, “Scheduling processes with release times, deadlines, precedence and exclusion relations”, *IEEE Trans. Software Eng.*, Vol. 16, No. 3, Mar. 1990

[251] J. Zeman, G.S. Moschytz, “Systematic design and programming of signal processors, using project management techniques”, *IEEE Trans. Acoustics, Speech, Signal Proc.*, Vol. ASSP-31, No. 6, Dec. 1983, pp. 1536–1549.

[252] J. Ziv, A. Lempel, “A universal algorithm for sequential data compression”, *IEEE Trans. Info. Theory*, Vol. IT-23, No. 3, May 1977, pp. 337–343.