False Path Elimination in Quasi-Static Scheduling

G. Arrigoni L. Duchini
Loquendo S.p.A.
Torino, Italy
{guido.arrigoni,luca.duchini}@loquendo.com

L. Lavagno C. Passerone
Politecnico di Torino
Torino, Italy
{lavagno,passerone}@polito.it

Y. Watanabe
Cadence Berkeley Labs
Berkeley, CA
watanabe@cadence.com

Abstract

We have developed a technique to compute a Quasi Static Schedule of a concurrent specification for the software partition of an embedded system. Previous work did not take into account correlations among run-time values of variables, and therefore tried to find a schedule for all possible outcomes of conditional expressions. This is advantageous on one hand, because by abstracting data values one can find schedules in many cases for an originally undecidable problem. On the other hand it may lead to exploring false paths, i.e., paths that can never happen at run-time due to constraints on how the variables are updated. This affects the applicability of the approach, because it leads to an explosion in the running time and the memory requirements of the compile-time scheduler itself. Even worse, it also leads to an increase in the final code size of the generated software.

In this paper, we propose a semi-automatic algorithm to solve the problem of false paths: the designer identifies and tags critical expressions, and synchronization channels are automatically added to the specification to drive the search of a schedule. As a proof of concept, the proposed technique has been applied to a subsystem of an MPEG-2 decoder, and allowed us to find a schedule that previous techniques could not identify.

1. Introduction

Embedded systems are rapidly becoming the main driver of the formidable growth of the electronic market. They are generally designed as software components on top of a special-purpose hardware platform, including both programmable (for flexibility; e.g., CPUs, DSPs, FPGAs) and non-programmable (for performance and efficiency; e.g., de-modulators, A/D converters, filters) components. Current design techniques for both the software and the hardware portion emphasize modularity, re-usability, and separation of concerns (e.g., between functionality and architecture, and between communication and computation). A commonly used design flow starts from a modular concurrent specification, in the form of communicating processes, and maps them onto architectural computation and communication resources, thus defining both performance properties and an implementation path. This modularity, based on high-level abstract communication and synchronization mechanisms [5], is the key to re-usability. It allows one to refine communication specified, e.g., as FIFOs, to software drivers, shared memory areas, and DMA channels based on the available platform resources. However, since efficiency is of paramount importance in embedded electronics, one must also use efficient and effective *software and hardware synthesis techniques*, that reduce the overhead otherwise due to a modular, implementation-independent specification.

In this paper we focus on techniques for software generation from specifications in the form of a set of sequential processes communicating concurrently via FIFO channels. In particular, we consider scheduling techniques that account for not only dataflow communications but also data-dependent controls. The importance of such techniques is more and more evident as complex control operations are required in recent dataflow applications. However, a known problem of such techniques is that the scheduling problem becomes undecidable in general when data-dependent controls are taken into account [1]. We have proposed earlier an algorithm for this problem [2], in which precise data dependency of the controls are abstracted as non-deterministic choices. With this abstraction, exactly how each control is resolved is not considered during the scheduling. Instead, the algorithm searches for a schedule assuming that any resolution of the control is possible. In this way, schedules can be computed in many cases that are otherwise undecidable, where the actual resolutions of data-dependent controls are made at run-time. However, this abstraction causes a problem when two or more data-dependent controls are correlated, i.e. some resolutions of those controls never happen during the execution. For example, one condition may never be evaluated as false when the other condition is true. We call such an execution path of the program that never happens because of correlated conditions a *false path*. In this case, a scheduling algorithm that employs this abstraction could generate unnecessarily large schedules, since it generates schedules also for the false paths. It could even classify as non-schedulable a specification for which a schedule ex-

ists when the false paths are accounted for.

The essential problem is that the exact characterization of data-dependency makes the scheduling undecidable while its abstraction causes false paths. This is the issue we address in this paper. While employing the abstraction of all the data computations, we propose a semi-automatic algorithm that eliminates false paths. We observed that in many practical cases, the designer can identify sets of correlated data-dependent controls in a relatively straightforward manner. We take this information as input, and automatically insert extra communication between the relevant processes to explicitly model the correlation. The resulting specification is then taken by the scheduling algorithm, which searches for schedules without exploring the false paths. Once schedules are found, the extra communication can be eliminated from the resulting code, since it is used only for finding the schedules. We have applied this technique to a part of an MPEG-2 decoder, and schedules have been successfully computed. We report this experiment and its results.

This paper is organized as follows. Section 2 summarizes previous work and describes the false path problem with a simple example. Section 3 describes the MPEG decoder that we used to illustrate our technique. Section 4 contains the main algorithm for the elimination of false negative results and of redundant code. Section 5 provides experimental results on the MPEG decoder. Section 6 draws some conclusions and outlines opportunities for future work.

2. The QSS algorithm

Our contribution starts from an existing algorithm for synthesizing code for the software component of a system ([2]). The specification of a system is given as a set of concurrent processes that communicate through ports connected by FIFO channels. The depth of a channel may be specified to be either bounded or unbounded. We restrict our attention to processes described as sequential programs and implemented as software on a programmable embedded processor. The sequential program for each process is specified in a language called *FlowC*, which is based on C and extended in order to specify communication operations. The synthesis process statically (at compile time) creates a set of tasks, each grouping portions of the code of the functional processes, so that the tasks are dynamically scheduled at run time. The number of generated tasks is equal to the number of global input ports from the environment that trigger the system executions. In general, we classify the global inputs into either *controllable* or *uncontrollable*. The former is a type of inputs for which the system issues an operation to read data when it needs to. The latter is opposite, i.e. the environment has the control and the system reacts to events given at input ports of this type. We generate a task for each uncontrollable global input port by grouping code of different processes. This grouping reduces the dynamic scheduling overhead, since generally we obtain fewer tasks than processes. It also provides a bound on the maximum size of originally unbounded FIFOs, and thus permits their implementation in an embedded system without virtual memory.

The FlowC language is defined by adding to ANSI C four primitives to specify processes and their communication via FIFOs:

- `Process(`*`portlist`*`){`*`body`*`}` declares a process and lists all the input and output ports. *body* contains C code plus the following communication primitives.
- `Write_Data(`*`port,var,n`*`)` writes n tokens from variable *var* (possibly an array) on port *port*.
- `Read_Data(`*`port,var,n`*`)` reads n tokens from port *port* into variable *var* (possibly an array).
- `Select(`*`port1,port2,...`*`)` non deterministically returns the index of exactly one port that has at least a token available. This primitive can be used in a `switch` statement to select a port from which tokens can be read[1].

A network of processes is transformed into a single Petri net [8] for scheduling. Ports are represented using place nodes, and each transition node is annotated with a fragment of code. Due to the fact that edge weights on Petri nets (determining how many tokens are read or written by a firing of a transition) are constants, we must restrict ourselves to FlowC specifications in which the third arguments of `Write_Data()` and `Read_Data()` are set to constants at compile-time. If this condition is not satisfied by the original specification, but the variables specified for the arguments can take only a finite (small) number of values, then a (manual) translation using `switch` statements is possible. In the case that we describe next this condition was always satisfied.

The QSS algorithm then finds a *schedule* for the Petri net: in order to do it, the algorithm creates dynamically a subtree of the reachability tree of the Petri net. The created subtree is composed of nodes that represent the system states and arcs that represent transitions which produce new states. A schedule is obtained by:

1. finding operations, specified in various concurrent processes, that need to be executed when the ports receive inputs from the environment, and
2. sequentializing them, while ensuring their execution using finite memory for the communication channels.

[1] `Select` destroys the determinacy property of Kahn process networks, in that the I/O behavior now depends on the scheduling choices. It is, however, very important to efficiently specify systems in which input arrival rates are not known a priori and are widely different. One example is video images and user commands in a multi-media system.

The resulting schedule, if it can be found[2], has the property that each node has a path to a state (marking) where inputs from the environment can be accepted. Thus the schedule can be executed every time a new input is received (e.g., as an interrupt). It also determines an upper bound on the quantity of objects stored at a time for each channel during the execution. The specification may contain data-dependent control constructs, such as `if-then-else` or `for` loops, and thus a total order of these operations cannot be determined in general until run-time, when values of data used by the constructs become known.

The concurrent processes are sequentialized in order to reduce run-time overhead maximally, while data-dependent control constructs are resolved at run-time (Quasi-Static Scheduling [1, 9, 10]). The schedule is then transformed into tasks by traversing the graph to generate software code using the original code annotated with the transition nodes.

2.1. The false path problem

The false path problem arises when data-dependent controls are abstracted as non-deterministic choices. Figure 1 shows a simple but typical example of the problem. There are two communicating processes, where each has a `for` loop. Process1 executes the loop body N times, while Process2 executes the loop body M times. However, Process1 sends the value of N to Process2, and the latter uses it as the value for M. Therefore, the number of iterations is identical between the processes. Figure 1-(b) shows a Petri net model of this specification. Each rectangle represents a transition node, in which the associated code is shown. The name of a node is shown in bold. Note that the loops of the specification are modeled as cycles in the net, but the correlation of the values of N and M is not modeled. Figure 1-(c) presents a part of a schedule for this Petri net, where each node represents a marking of the net and an arc is associated with a sequence of transitions. The two markings indicated by circles, p1p6 and p3dtp4, are not reachable during an execution of the specification due to the correlation of the `for` loops. The former is reachable only if the number of iterations of the loop in Process1 is less than that of the loop of Process2, the latter corresponds to the opposite case. Since N and M always assume the same value, we know that neither is possible. Hence, any path originating from these markings is false. However, the Petri net-based scheduling algorithm does not have this information, and thus it would continue to search for a schedule along the false paths, which is completely unnecessary.

There is a large body of research on the identification of such false paths in standard sequential programs, ranging from simple constant propagation and elimination of impossible branches, to full symbolic interpretation of a program [3], often limited to linear expressions and "standard" loops [6]. Such techniques, however, can be applied only once a program (and hence a schedule) has been generated, while we want to apply them *on the fly* during schedule generation. In this paper we discuss a more pragmatic, semi-automated approach to false path pruning, that relies on some user intervention at the local (process pair) level. It is similar in spirit to the manual introduction of linear constraints to represent such false paths in [7], but it works in the context of QSS rather than worst-case execution time analysis.

[2]Our algorithm is conservative, in that it may declare a schedulable specification to be non-schedulable, due to the undecidability of the schedulability problem for Kahn process networks.

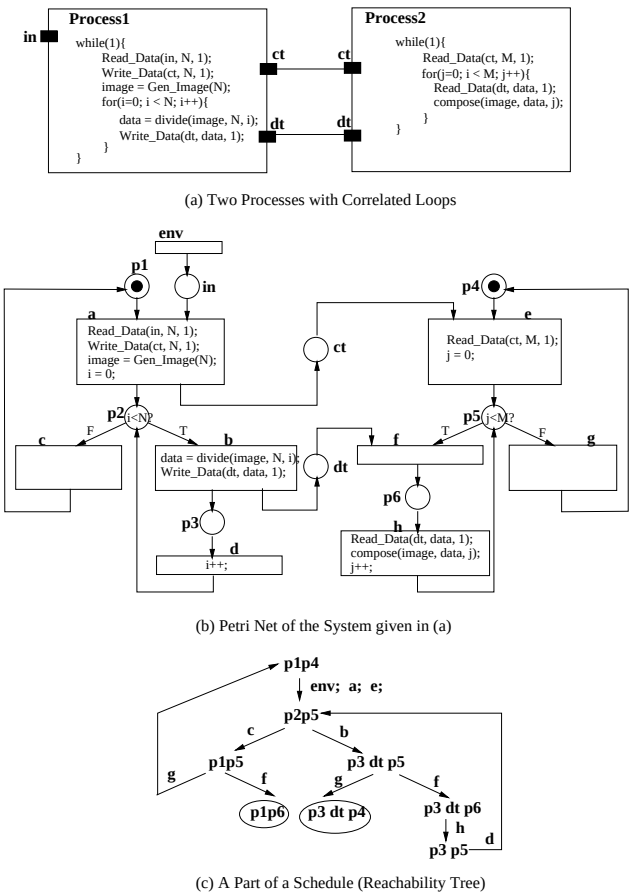

Figure 1. Example of Specification with False Path

3. The MPEG-2 decoder

We used as our test system an MPEG-2 video decoder developed by Philips (see [11]). The system is composed of a set of concurrent processes, as shown in Figure 2. Processes Thdr and Tvld parse the input video stream; Tisiq and Tidct implement spatial compression decoding; TdecMV, Tpredict and Tadd are responsible for decoding temporal compression (i.e., forward and backward predictions) and generating the image; Tmemory, TwriteMB, TmemMan and Toutput manage the frame store and produce the output to be sent to a visualization device. Communication is by means of channels, which have a FIFO semantics and can handle

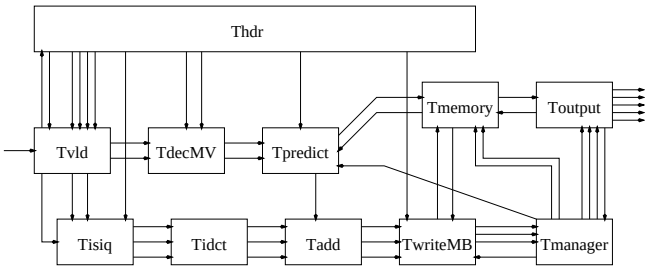

Figure 2. MPEG-2 video decoder block diagram

arbitrary data types. Philips used approximately 7700 lines of code to describe the 11 processes, and 51 channels to connect them. An average of 16 communication primitives per process are used to transfer data through those channels.

In the original implementation, all processes were scheduled at run time using a Real Time Operating System. Our objective was to reduce run-time scheduling overhead due to context switchings, by merging processes as much as possible into quasi-statically scheduled ones. This also leads to further improvements in performance, since internal communication between merged processes reduces to simple assignments rather than a full FIFO implementation (e.g., as a circular buffer in memory).

We focussed our attention on five processes: Tisiq, Tidct, TdecMV, Tpredict and Tadd. Namely, these processes constitute our system to be scheduled, and inputs to any of the processes are considered as global inputs from the environment. They consist of about 3000 lines of code and account for more than half of all communication occuring in the system. Albeit we generated Petri Nets for other processes as well, we did not schedule the entire system because we wanted to preserve some concurrency between processes and to verify the interaction between the generated code and the rest of the specification. Moreover, we report profiling results on a single processor machine, but this partition would also allow to map the MPEG-2 video decoder to different threads on multiple processors. Our procedure generated a single process with the same interface as the original ones, that could be plugged into the MPEG-2 netlist, replacing the original five processes.

We first specified the selected processes using the FlowC language. The original specification of the MPEG-2 decoder used the YAPI language ([4]), to which FlowC is closely related, and translation was generally straightforward.

Figure 3 shows a small fragment of code taken from processes Tpredict and Tadd. They both implement a `while` loop during which they exchange some data (a macro-block is written from Tpredict to Tadd). The condition to exit the loop is the same in both cases, although each processes uses a local variable to evaluate it. Moreover, the number of iterations of the loops is not known at compile time. This is a typical situation, where it is difficult to statically determine that the two conditions are really the same, but it is fairly

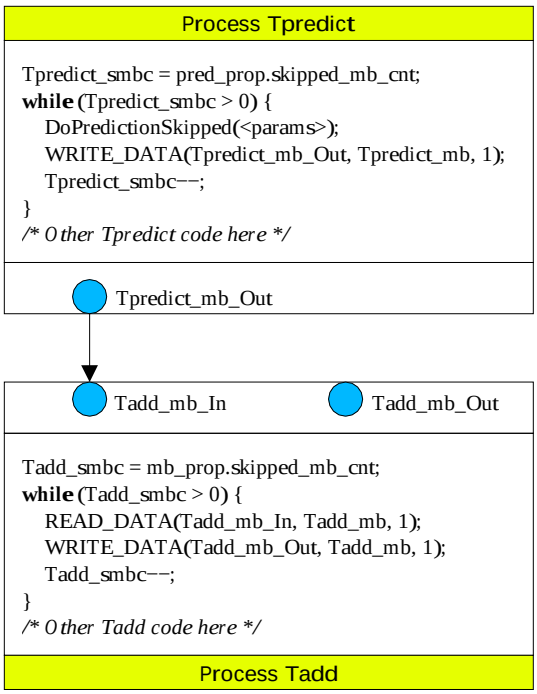

Figure 3. Example of FlowC specification

easy to identify by the designer.

4. False path elimination and synchronization

As we discussed above, false paths are due to the presence of *correlated* conditional expressions in concurrent processes (e.g., they use the same value, and hence always both take the same branch). Such expressions may appear in C constructs such as `while` and `for` loops, `if-then-else` and `switch`. Without loss of generality, in the following we will consider only `if-then-else` statements, to which both `while` and `for` loops can be easily reduced, by using labels and `goto` statements to implement loops. The same technique, with slight obvious modifications, applies also to `switch` statements.

We require the designer to identify sets of conditions in different processes that always give the same outcome throughout the execution of the system. For each set, an appropriate number of synchronization channels and primitives that read and write on them can be automatically added to the specification based on the designer labeling. The condition is removed from all processes but one, and the flow of control is maintained using the `Select` construct. *We replace an implicit data-dependent synchronization between the processes with an explicit communication and control-dependent synchronization*, that can now be exploited by our Petri net-based scheduling algorithm. Since only one process now evaluates the condition, the false path is eliminated and a static schedule can be found. Once the schedule is obtained, *synchronization channels and communication primitives operating on them can be removed, so there is no over-*

head in the final generated code.

Merging such multiple synchronized loops into a single loop in the generated schedule is an important aspect of our Quasi-Static Scheduling technique. Their specification in FlowC is much easier and more natural, and leads to better re-use opportunities, than the often cumbersome graphical notations, requiring the use of pre-defined patterns of `Select` and `Merge` nodes, used by scheduling algorithms for graphical dataflow-like networks [1, 9].

For each correlated set, that has been identified by the designer with a unique *tag* using `pragma`s, we apply the following algorithm (assuming for the sake of simplicity that the set has cardinality two and that P1 and P2 are the processes where the conditions appear):

1. Add two ports, called <*tag*>`true` and <*tag*>`false`, to both processes P1 and P2, where <*tag*> is the name used by the designer.

2. Connect the ports together in the netlist.

3. Add two `Write_Data` statements at the beginning of both branches of the `if-then-else` construct in process P1, one writing on port <*tag*>`true` and the other on port <*tag*>`false` (the written values are not important).

4. Delete the `if-then-else` construct from process P2 and add a `switch` on the output of a `Select` statement, with the two new input ports as arguments. The condition should still be evaluated before the `switch` (in case it has side effects), but its returned value is discarded.

5. Fill in the `case` clauses in the `switch` by first reading the appropriate input port, and then executing the statements from the original specification.

As an example, consider the two processes P1 and P2 shown in Figure 4(a). Suppose that the designer identified `condition1` and `condition2` as identical in all cases, and tagged them with the name *sync* using a `pragma` directive. Then it is possible to automatically synchronize the two processes, thus avoiding the false path, by structurally changing the code as shown in Figure 4(b). This is because when `condition1` is true, P2 receives data at the port *synctrue*. Since port *syncfalse* does not have data, P2 deterministically selects `case` 0 to execute stm3. The situation is similar in case `condition1` is false. If the `else` clause is absent in either of the branches, we create a no-op statement to take its place.

Some optimizations are possible in special cases:

- If the condition in process P2 does not change the value of any variable (i.e., by using post-increment, for instance) and does not call functions with static variables, then it can be omitted in the generated code.

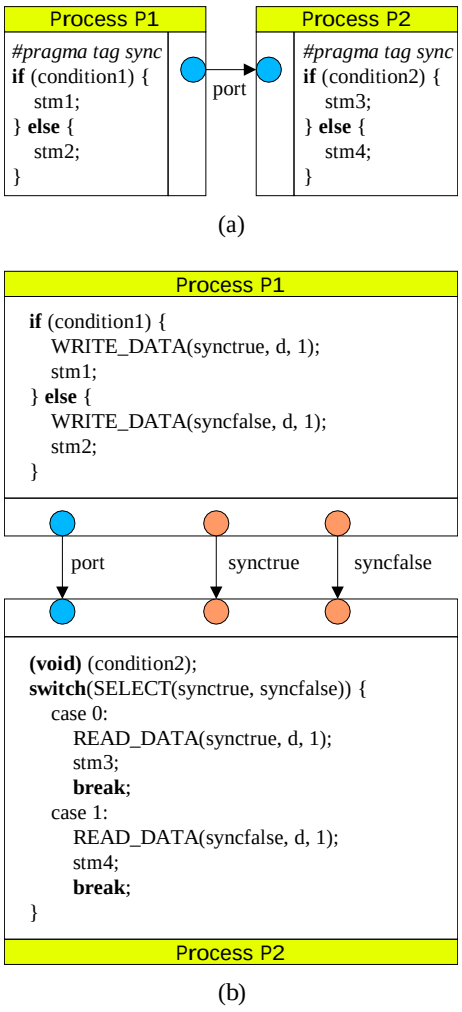

Figure 4. False path elimination by explicit synchronization

- If statements in the `if-then-else` of the original code already perform some communication operations, then those operations can be used to synchronize the two processes. This is often the case, at least for one of the two branches, that thus does not require the addition of a `Read_Data` statement, only of the appropriate `select` clause. Suppose, for instance, that `stm1` in Figure 4(a) contains a write operation on port `port`, and that `stm3` contains the corresponding read operation. Then port `synctrue` and communication primitives defined on it are not needed, as long as the `Select` statement in process P2 of Figure 4(b) is modified to look at ports `port` and `syncfalse` (in that order).

5. Application to the MPEG-2 decoder

We applied the Quasi Static Scheduling technique to the subset of the MPEG-2 decoder identified in Section 3. The

Petri net generated from the FlowC specification has 115 places, 106 transitions and 309 arcs. For this Petri net, we tried to find a schedule without inserting any synchronization. However, because of the false path problem, the algorithm failed to find a schedule and ran out of memory (we used a Sun Ultra Enterprise 450 with 512 Mbytes of main memory). By following the dynamically created reachability tree, it was obvious that the reason for this behavior was due to the presence of false paths, which caused the exploration of a large number of nodes which could not be reached during real execution.

We used the technique described in Section 4 to solve this problem. The task of identifying the conditions that had to be tagged in order to reduce the size of the problem so that it could be scheduled required only one day. The code was then automatically instrumented using the algorithm that we presented above. 18 point-to-point channels were added for synchronization; an average of 2 `Select` statemens and 7 read or write operation per process were needed to make use of the newly created channels. After these changes, we successfully generated a schedule for the five processes and synthesized a task to be used in the complete MPEG-2 decoder. The running CPU time of the scheduler on the same Sun workstation was 12 seconds, and the peak memory usage 13 MBytes.

Figure 5 shows the updated code of processes Tpredict and Tadd from Figure 3. Note that the `while` loop in process `Tpredict` has been transformed in order to use an `if-then-else` construct (and an `else` clause has been added), and the `while` loop in `Tadd` is now implemented using the newly created synchronization channels.

Figure 6 shows the generated code after the scheduling process. Several optimizations are still possible (only the first two have been applied in our experiments):

- All channels, with the exception of `Tadd_mb_Out`, are now internal to the generated single task, and therefore can be implemented using assignments rather than external FIFOs.
- Writes and reads to and from synchronization channels can be completely eliminated, since they were useful only to drive the scheduling process.
- The `else` clause only contains statements that can be safely (and automatically) deleted, therefore it disappears completely.
- Variable `Tadd_smbc` is never used in any assignment and can be eliminated.

If all the above optimizations are performed, we are left with a *single* `while` *loop which contains the appropriate interleaving of statements from the two original processes that an experienced designers would have written for a monolithic implementation*. Using our technique this result is achieved automatically from a modular specification. The resulting code not only avoids run-time scheduling overhead

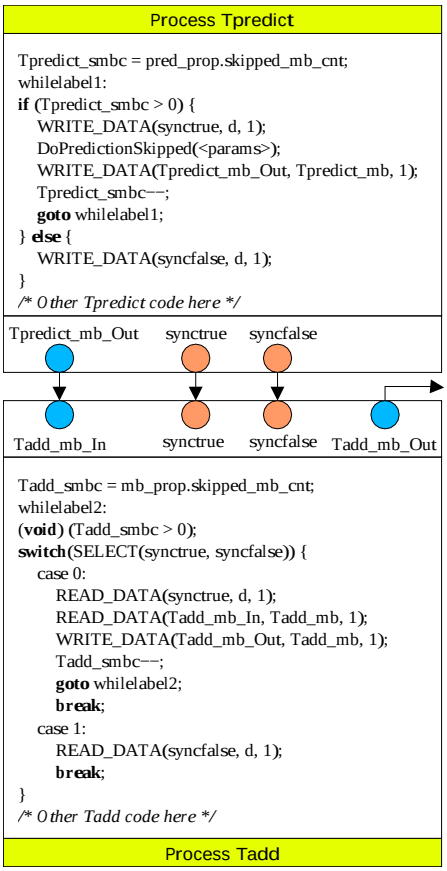

Figure 5. FlowC specification instrumented with synchronization channels

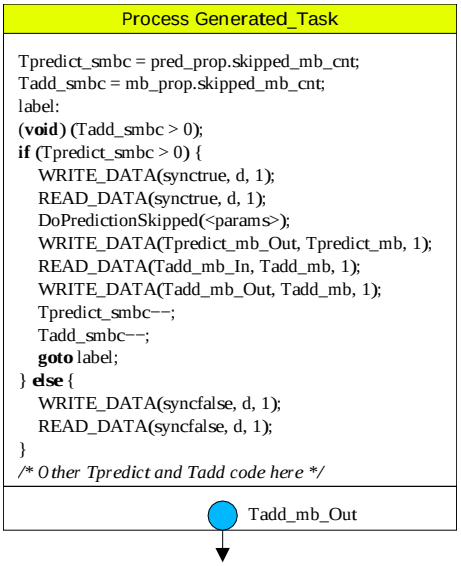

Figure 6. Portion of the generated code for the MPEG-2 decoder

	Total	MPEG2			Test bench	OS
		Total	Parser	5Procs		
Orig.	7.5	4.66	0.94	3.72	0.27	2.58
QSS	4.1	2.51	0.94	1.57	0.28	1.31

Table 1. CPU time, in seconds, of the MPEG-2 example

	5 Processes				
	Total	**Comp.**	**Int. Comm.**	**Ext. Comm.**	**Code size**
Orig.	3.72	1.01	2.23	0.48	18K
QSS	1.57	0.96	0.13	0.48	24K

Table 2. CPU time, in seconds, and code size of the five selected processes

and reduces data transfers, but it is also a better starting point for a standard compiler, since opportunities for optimizations across process boundaries are now possible.

We compared the performance of the original concurrent specification of the MPEG-2 decoder with those of the same system where a single statically scheduled process is used in place of the five initial ones. In both cases, we removed the processes that manage and implement the memory, but we kept those that parse the input MPEG stream. Both systems received as input a video stream composed of 4 images (1 intra, 1 predicted, 2 bidirectional predicted).

Table 1 summarizes the total execution time (on the same Sun machine described above) for the two implementations. It also shows the individual contributions due to the processes implementing the MPEG-2 decoder (split among the parser and the five processes that we scheduled together), the testbench and the operating system (that dynamically schedules the tasks). The increase in performance is around 45%. The gain is concentrated in the statically scheduled processes, due to the reduction in the number of FIFO-based communications, and in the operating system due to the reduction in the number of context switches.

Table 2 compares the execution times due to computation and communication of the five considered processes, both in the original system and in the quasi-statically scheduled one. As expected, computation and external communication (i.e., with the environment) are not significantly affected by our procedure. However, internal communication is largely improved: this is because after scheduling we could statically determine that all channels connecting the five considered processes never have more than one element at a time. Therefore, communication is performed by assignment, rather than by using a FIFO or a circular buffer. The table also report the object code size, which increases in the generated single task with respect to the 5 separated process: this is due to the presence of control structures representing the static schedule in the synthesized code.

6. Conclusions

This work showed that significant gains can be obtained both in terms of off-line scheduling time, and in terms of final system performance, by avoiding the consideration of false paths during Quasi-Static Scheduling of concurrent specifications. The resulting code can be compared, by inspection, with the level of quality that is expected from a careful designer that takes into account the full system behavior simultaneously. However, it can be obtained by simply tagging appropriate portions of a modular specification.

In the future, we are considering how techniques such as those described in [6] can be applied to reduce the amount of manual intervention.

References

[1] J. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, U.C. Berkeley, 1993.

[2] J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, and A. S. giovanni Vincentelli. Task Generation and Compile-Time Scheduling for Mixed Data-Control Embedded Software. In *Proceedings of the 37th Design Automation Conference*, June 2000.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. on Principles of Programming Languages*, Los Angeles, January 1977.

[4] E. de Kock, G. Essink, W. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers. YAPI: Application Modeling for Singal Processing Systems. In *Proceedings of the* 37^{th} *Design Automation Conference*, pages 402–405, June 2000.

[5] S. Edwards, L. Lavagno, E. Lee, and A. S. giovanni Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, Mar. 1997.

[6] N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In B. LeCharlier, editor, *International Symposium on Static Analysis, SAS'94*, 1994.

[7] Y. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the Design Automation Conference*, June 1995.

[8] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, pages 541–580, Apr. 1989.

[9] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, and et al. Scheduling hardware/software systems using symbolic techniques. In *International Workshop on Hardware/Software Codesign*, 1999.

[10] F. Thoen, M. Cornero, G. Goossens, and H. D. Man. Real-time multi-tasking in software synthesis for information processing systems. In *Proceedings of the International System Synthesis Symposium*, 1995.

[11] P. van der Wolf, P. Lieverse, M. Goel, D. Hei, and K. Vissers. An MPEG-2 Decoder Case Study as a Driver for a System Level Design Methodology. In *Proceedings of the* 7^{th} *International Workshop on Hardware/Software Codesign*, pages 33–37, May 1999.