The Use of a Virtual Instruction Set for the Software Synthesis of Hw/Sw Embedded Systems

A. Balboni (‡), W. Fornaciari (†,§), D. Sciuto (§), M.Vincenzi (†)

(‡) ITALTEL-SIT, CLTE, 20019 Castelletto di Settimo m.se (MI), Italy.
(†) CEFRIEL, via Emanueli 15, 20126 Milano (MI), Italy.
(§) Politecnico di Milano, Dip. di Elettronica e Informazione, P.zza L. Da Vinci 32, Milano, Italy.

Abstract

The application range of the embedded computing is going to cover the majority of the market products spanning from consumer electronic, automotive, telecom and process control. For such a type of applications, typically there is a strong cooperation between dedicated hardware modules and software systems. A important issue toward a fully automated system-level implementation is represented by the software developing process. The basic requirements are: accurate timing characterization to be used during the early stages of the design to compare alternative architectures and reliable synthesis techniques to ensure the respect of the correct functionality by avoiding, as much as possible, the direct designer's intervention during the development process.

This paper aims at describing a novel methodology to address the needs of concurrently synthesizing the software component of a control-dominated hardware-software system, possibly under real-time constraints. An intermediate model (Virtual Instruction Set) for the software is presented, suitable for both for synthesis and analysis purposes. The overall system synthesis is presented with particular emphasis on the problem of low level performance estimation, static scheduling of the software process and retargetable code synthesis.

1. Introduction

The increasing pressure of the embedded system market to shrink time-to-market, size, power and cost of products is facing the designers with the problem of managing the entire synthesis flow of architectures composed of programmable software and hardware *engines* [1]. In this paper we refer to dedicated designs as those embedded in equipment for automotive, process control and telecom applications such as public switching and networking systems. For this class of systems, the function to be implemented is well known in advance and it will never change during their operational life, apart from possible minor adjustments. As opposed to the general purpose computer systems, embedded applications do not require a significant degree of flexibility once they are running. They are usually produced in large volumes and sophisticated analyses and optimizations can be carried out to meet the design requirements. According to the particular data on which the design is focused, the system can be roughly classified into: data dominated and control dominated [1, 2]. We devote our attention mainly to the control-dominated applications whose purpose is to react to external stimuli, possibly within some tight deadlines, but where data intensive functions play a secondary role [2, 3, 4].

The challenge of finding out a methodology to develop cost-effective hardware-software systems has been driven by extending the research in high-level synthesis to cover the particular requirements of buildings blocks and specification formalisms for mixed hardware-software architectures [5]. Our goal is to identify a pragmatic design methodology for control dominated embedded systems to be actually used, and therefore interfaced with the industrial requirements, of an Italian telecom company whose main products are in the area of public switching. This project, named TOSCA (Tools for System Codesign Automation), started some years ago and is currently under development at the CEFRIEL research center [6]. The aim is to define a methodology and the related CAD support to cover a complete co-design flow: the capturing of uncommitted process-level specification including the designer' objectives and constraints (*co-specification*), the identification of a suitable system-level modularization of the initial specification (*design-space exploration* and *binding*) to be mapped onto the hardware or software components composing the target hardware-software architecture (*co-synthesis*). Hence, before committing to a specific implementation, the initial system specification can be manipulated to fulfill the target design requirements. This partitioning process, can be viewed as an incremental activity of modification of the initial specification through the application of transformations under the driving of metrics for system quality evaluation [6]. The internal model is based on a customization of the OccamII [7] computational model. The specifications and constraints can be captured through a built-in mixed textual/graphical editor. Direct import fron existing design entry tools (such as speedCHART) is also supported, based on the use of suitable OccamII process templates.

After the identification of the optimal system partitioning and binding, the following stage is the mapping of the system description onto an actual digital architecture. The TOSCA co-synthesis is performed by two different tools: a tool for VHDL generation for hardware-bound architectural units including interface

generation, and a software synthesis tool for software-bound architectural units, including a lightweight Operating System. Interfaces are generated with respect to the adopted model of communication, currently a fixed protocol is available, consistent with the target architecture adopted.

This paper emphasizes the problem of the modeling and synthesis of the software programs running on the microprocessor. The presented approach aims at providing a complete solution by taking into account the operating system including CPU scheduling and I/O management, direct generation of the software at assembly level, and the definition of a model of the software for co-design purposes to allow code retargeting and VHDL-based co-simulation of hardware-software systems.

In literature, the identification of the software side of the system is frequently unbalanced between hardware and software, since typically it is considered at a higher level such as the C language. A comprehensive analysis of the proposals can be found in [8]. For many dedicated applications, it is becoming important to consider the real time constraints not only during the hardware generation but also during the system-level partitioning and software synthesis. Recent works have considered the software properties at a finer granularity [9, 10]. Specific scheduling algorithms are emerging to manage the coexistence of interacting hardware and software parts in the global design [4, 11] as well as to optimize the interfaces [9]. Software performance modeling and compiling techniques are now entering as a substantial stage of the co-design flow both for reactive systems [3, 10] and data dominated applications [8, 12, 13], possibly with retargeting capabilities [12, 14].

For our target application field, frequently requiring hard real-time performance and low-level characterization of I/O interfaces, we found advantageous to consider the software at a lower level of detail with respect to C to predict the effects of the compiler and of the operating system.

The software generation is based on the use of a generic Virtual Instruction Set (VIS) acting as an intermediate language between OCCAMII and the target CPU assembly aiming at capturing the minimum set of features shared by microcontrollers for embedded applications. According to the needs of the application field, requiring simplicity and predictability, a static schedule approach with a coroutine scheme has been adopted as the target software structure. The VIS model is useful both for evaluation and synthesis purposes and it is conceived to achieve the following goals:

- simulation of the mixed hardware-software system within the same VHDL simulation environment by using simplified non-proprietary microprocessor models [15].
- extension of the analysis to cover also multiple processor families with minimal user intervention;
- good predictability of the final running software behavior and cost;
- integrated design flow able to cover the entire system synthesis in terms of hardware, interfaces and software.

The actual use of VIS within the TOSCA environment provides similar properties of flexibility and retargeting typically reserved to high level programming languages. In fact, it is possible to map the VIS code onto different target assembly instruction sets while accurately predicting the performance of the CPU and I/O. Even if the VIS code is readable, it has to be pointed out that user should interact with it only during the final system-level co-simulation, the maintenance of the system has to be performed at system-level.

The paper is organized as follows. After a brief description of the target architecture we are considering to physically implement the system, the problem of scheduling the software-bound modules identified during the system exploration is detailed. In particular, section 3 presents an overview of our solution based on a static scheduling using the information obtained during the VIS generation, detailed in section 4. The main features of the VIS notation are introduced by means of a small illustrative example. The paper is concluded by an outline of the results of our researches in the field of software modeling and generation for embedded systems and by the guidelines of current investigation effort.

2. The target architecture for system synthesis

Once a pair of hardware and software bound sets of modules have been defined, the following step is to produce their implementations. The synthesis stage will produce a mapping of the system onto the target architecture reported in terms of:

- assembly-level code for each sw-bound process, according to the target microprocessor instruction set;
- operating system support for process to process communication (both between sw to sw and sw to hw), as well as for CPU scheduling;
- VHDL code for each architectural unit (coprocessor) corresponding to hw-bound processes; this includes also the implementation of the hardware side of the interfacing subsystem, allowing the mapping of the abstract process to process communication onto an actual system architecture.

The entire system is intended to be implemented via a single ASIC including a microprocessor core with its memory, even if part of the memory can be external, and the dedicated logic implementing the set of hw-bound modules (called coprocessors) identified during the system-level design phase.

The master processor is programmable and the software can be either on-chip resident or read from an external memory; dedicated units operate as peripheral coprocessors. Hardware and software bound elements are interfaced by means of a master-slave shared bus communication strategy. All hardware to hardware communications are managed through dedicated lines. The RAM memory required for program/data storage shares the main data bus with the coprocessors, but can be accessed only by the master CPU. Communications among CPU and coprocessors are based on a memory mapped I/O scheme with one bus interface manager per coprocessor based on a common I/O buffered protocol manager.

A suitable VHDL generator has been developed, starting from the OCCAMII description stored within the database and building a tree modeling the statements nesting. It produces a set of modules corresponding to the hardware bound architectural units (coprocessors) with their communication interfaces. The VHDL code generator

performs a depth-first scan of the tree representing the OCCAMII structure and produces two output files: the first contains the entities declarations with the corresponding behavioral description while the second is a package containing all the procedures necessary to realize the communication among processes.

Since channels are not supported by VHDL, *ad hoc* fully hardware interface structures covering both buffered and unbuffered communication have been introduced [6, 15].

3. The software scheduling strategy

The software system is designed according to the reference architecture that is itself strongly influenced in terms of programming paradigm and hardware by the application field and the cost/performance goals. The run-time support provided in TOSCA has been kept minimal and includes only those features that are actually needed to support exception handling, configuration control, communication management and process activation, chosen during the customization phase. The operating system micro-kernel actually acts as a high-level process manager whose evolution is controlled by a deterministic algorithm, with synchronization among processes or with the environment (i.e. the coprocessors or external devices connected to the system).

Since the current target architecture considers just one microprocessor, concurrency is emulated through interleaving of processes, each corresponding to a software-bound part of the system modularization, whose ordering is statically defined, i.e. a pre-runtime schedule has been adopted. This solution has been chosen because high processor utilization is foreseen to reduce implementation costs, therefore not much spare CPU time is available. As a consequence, a solution able to guarantee *a priori* that all the stringent timing constraints will be met, seems to be the only viable. We found many advantages of this solution compared to the presence of an on-line schedule policy [16], such as the significant reduction in the share of run-time resources necessary to implement context switching and the scheduling itself, but the most important is that it is easier to satisfy our primary goal of meeting the real-time deadline.

Software-bound processes, that are viewed as a set of sequential cooperating threads with shared memory similar to a coroutine scheme, are constituted by operations that must be executed in a prescribed order. The number of processes is known in advance, and it will never change run-time. This implies that the operating system does not require a dynamic scheduling since the scheduling policy can be computed off-line and *code-wired*. Therefore, the solution proposed requires only a small operating system providing the mechanisms for process activation and the communication support.

In general, two classes of processes can be present: *periodic*, whose computation is executed repeatedly in a fixed amount of time and *asynchronous*, that usually consist of computations responding to an event (internal or external). Different and more general techniques for mapping asynchronous processes onto an equivalent set of periodic processes can be found in [17], therefore we considered only the problem of scheduling periodic processes.

The algorithm we implemented, given the set of processes constituting the sw-bound part of the system, determines a schedule (whenever it exists) such that each process is activated after its release time and carries out its computation before its deadline. Even though the exact timing characteristics of system components and events sometimes cannot be predicted, we overcome such a problem by using a worst case estimation of these parameters so that the scheduling algorithm can guarantee a predictable behavior.

The methodology we adopted to obtain the pre-run time schedule is based upon a systematic improvement of an initial schedule until a feasible (near optimal) schedule is found. The analysis of the VIS code corresponding to each software-bound part allows the VIS scheduler to consider each software process characterized by a *release* time, the *duration* and a *deadline*, which can be broken into a set of code segments. An analysis of the internal composition of the process provides the start time of each code segment relative to the beginning of the process it belongs. Exclusion relations may be present among segments when some of them must avoid interruption by others to prevent possible errors caused by simultaneous access to shared resources, e.g. data structures, I/O devices, coprocessors. Precedence relations, that occur when a segment requires some information produced by other process segments, are also considered.

The scheduler produces an ordered set of code segments fulfilling deadlines and constraints (if they exist) where the lateness of all segments is minimized. The context switching overhead has been considered by including an additional delay to the purely computational time. To characterize the software running onto the microprocessor, specific information is produced by the scheduler by inspecting the final schedule produced: the level of process fragmentation and the relative overhead, the slack time, the CPU utilization and, in case of non feasible schedule, the critical segments responsible of the algorithm failure.

The details of the implemented scheduling algorithm can be found in [15].

4. The VIS-based software synthesis

The VIS is defined in terms of a customizable and orthogonal register-oriented machine with a common address space for both code and data. This means that each register can act as accumulator and all the operations (e.g. addressing, arithmetic-logic, data transfer) can be performed no matter which register is used as operand. The instruction set has been designed in order to be easily retargeted onto different CPUs: a mix of CISC and RISC typical instructions are included. A generic VIS instruction can either be one-to-one mapped on a native target assembly instruction or correspond to a group of assembly instructions. In such a way, if the selected CPU does not match the VIS instruction, the retargeting of the code is performed via an alternative definition of the instruction using only the RISC-*side* of the VIS, thus reducing the effort to reconfigure the software whenever alternative CPUs are evaluated.

The VIS supports unsigned/signed integer data types (BIT, BYTE, INT16 or *word* and 32-bits integer called

longword) as well as all typical arithmetic/logic operations. The address space spans over 32 bits so that each VIS argument is always contained within a longword. The memory format for the data is aligned in terms of 32-bits words, as a consequence four memory locations are necessary to store a byte. Boolean variables can be packed to save memory space.

The instruction format is similar to the one of the MC68000 with a suffix indicating the operand type, e.g. **`MOVE.B R1, R2`** copies a byte from **`R1`** to **`R2`**. The number of arguments of each VIS instruction ranges from 0 to three. Control instruction are usually characterized by zero operands, while three operands instructions are typically arithmetic-logic operations such as **`SUB.W R0,R1,R2`** where the **`R0`** and **`R1`** are the operands and **`R2`** will contain the result. Both *destination* and *source* can be registers or memory references (*source* can also be an immediate operand).

The generation of the VIS code as well as the final implementation of the software running on the target microprocessor follows the phases depicted in fig.1. Three main steps compose the top part of such an activity: initialization, code generation, estimation of time delays and binary code size.

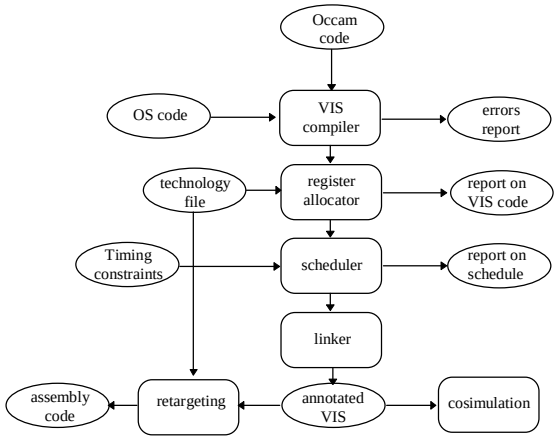

Figure 1. The software generation process.

At the beginning the system is initialized by reading from a technology file the information characterizing the selected CPU, e.g. the clock period, the instruction set, the registers number, type and size of the registers, the number of clock cycles per instruction, etc. The code generation involves registers use optimization and automated packing of bit variables. The estimation of timing performance and memory requirements for back-annotation towards the system design exploration phase can be obtained for several possible microprocessor cores. In fact, its actual behavior is represented by the following three groups of information (for each foreseen CPU):

- retargeting rules (RR): specifying the rules for mapping VIS code onto the target microprocessor instruction set;
- time/size table (TST): reporting for each VIS instruction the number of clock cycles and bytes of the corresponding target CPU mapping (which, in general, is not composed of a single instruction).
- technology (TF): containing information on the adopted CPU as the BUS width, the power consumption, the pin-out of the microprocessor, the particular characteristics of the adopted model of microprocessor with respect to the rest of the CPU family, such as, for instance the memory size.

The first step is the compilation of the OCCAMII specification in VIS. Although the entire system specification will not probably be implemented in software, the estimation of the VIS performance and cost is initially carried out for all the OCCAMII modules composing the description. The obtained result is employed during the system partitioning to compare/drive alternative modularizations and hardware vs software bindings. The obtained code is not executable since the following decisions have yet to be taken: register mapping, process scheduling, system bootstrap, memory allocation including symbolic vs actual address determination.

A pre-allocation of the register to extract execution times and memory requirements is performed according to the information included within the technology file. The VIS code is then annotated with the information needed by the scheduler to produce a correct ordering of the processes execution, by adding some bracket-encapsulated tags. A simple example of VIS compilation for a sub-part (Hamming encoder) belonging to a car antitheft system that we use as a small benchmark, is here reported. The example is composed of a process receiving data from the channel **`dataIn`** where two parallel sections allow the system to compute the Hamming encoding of the input to be transmitted on the **`dataOut`** channel (see fig.2).

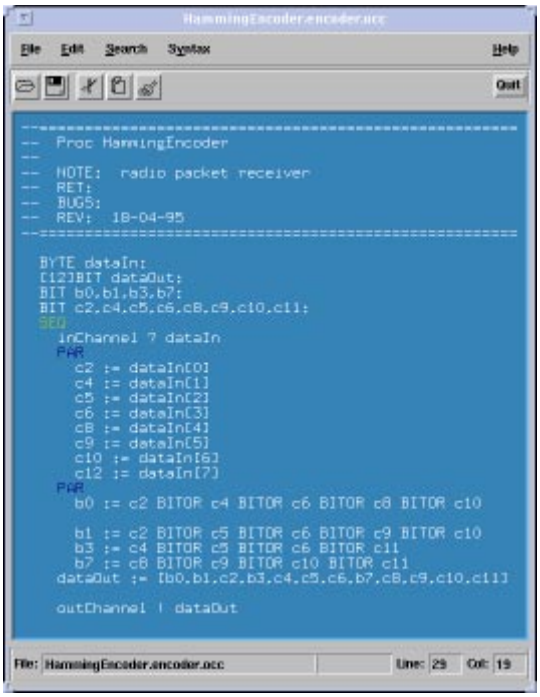

Figure 2. The textual part of a sub-module of a car antitheft system containing the description of a Hamming decoder captured by using the TOSCA OCCAMII visual editor.

The VIS code maintains a structure similar to the original OCCAMII model: the body of each process is identified through the **`<process>`** and **`</process>`**

tags while concurrent processes (corresponding to the PAR construct of OCCAM) fall within the scope of a **<group>** identifier.

As reported above, the scheduler may require to break the processes to meet time deadlines; as a consequence, it is necessary to consider the impact of additional context-switching overheads. The scheduler performs such an analysis by considering the **<USE Regs-List>** and **<LEAVE Regs-List>** tags which represent, incrementally, the registers necessary to be saved at any point in time. Critical sections, corresponding to non-breakable actions such as an interrupt handling, are enclosed within **<atomic> </atomic>** to prevent a possible preemption.

Data transfer from software to hardware and viceversa is modeled via memory-mapped coprocessor registers, associated with each port. In this example, the 12-bits channel has been mapped onto a 16 bits word corresponding to a pair of contiguous memory locations.

```
    //PROC HammingEncoder (CHAN OF BYTE inChannel, CHAN OF [12]BIT outChannel)
          <ports>
          inChannel
          outChannel
          </ports>
          <data>
    //BYTE dataIn:
    dataIn defb                0
//[12]BIT dataOut:
    dataOut                    defw          0
    //BIT b0,b1,b3,b7:
    b0     defb                0
    b1     defb                0
    b3     defb                0
    b7     defb                0
    //BIT c2,c4,c5,c6,c8,c9,c10,c11:
    c2     defb                0
    c4     defb                0
    c5     defb                0
    c6     defb                0
    c8     defb                0
    c9     defb                0
    c10    defb                0
    c11    defb                0
          </data>
          <code>
    //SEQ
    //inChannel ? dataIn
    //PAR
          <group>
    //b0 := c2 BITOR c4 BITOR c6 BITOR c8 BITOR c10
          <process><use R0>
          move.b               @c2(BP),R0    <2 4 4>
          or.b                 @c4(BP),R0    <2 4 4>
          or.b                 @c6(BP),R0    <2 4 4>
          or.b                 @c8(BP),R0    <2 4 4>
          or.b                 @c10(BP),R0   <2 4 4>
          move.b               R0,@b0(BP)    <2 4 4><free R0>
          </process 12 24 24>
```

... similarly for b1 and b3 ...

```
    //b7 := c8 BITOR c9 BITOR c10 BITOR c11
          <process><use R0>
          move.b               @c8(BP),R0    <2 4 4>
          or.b                 @c9(BP),R0    <2 4 4>
          or.b                 @c10(BP),R0   <2 4 4>
          or.b                 @c11(BP),R0   <2 4 4>
          move.b               R0,@b7(BP)    <2 4 4><free R0>
          </process 10 20 20>
          </group>
    //dataOut := [b0,b1,c2,b3,c4,c5,c6,b7,c8,c9,c10,c11]
          <process><use R0>
          move.b               @b0(BP),R0    <2 4 4>
          <use R1>
          move.w               0001h,R1      <2 2 6>
          and.w                R1,R0         <1 1 2>
          <use R2>
          move.b               @b1(BP),R2    <2 4 4>
          shl.w                #1,R1         <1 1 2>
          and.w                R1,R2         <1 1 2>
          or.w                 R2,R0         <1 1 2><free R2>
          <use R2>
```

... similarly for c2, b3, c4, c5, c6, b7, c8, c9, c10 ...

```
          move.b               @c11(BP),R2   <2 4 4>
          shl.w                #1,R1         <1 1 2>
          and.w                R1,R2         <1 1 2><free R1>
          or.w                 R2,R0         <1 1 2><free R2>
          <live R0></process 59 84 122>
//outChannel ! dataOut
          <process>
          <atomic><live R0>                  //in this case atomic it is not
                                             //actually necessary
          move.l               outChannel,R1 <2 2 6>
          call                 write_int     <2 4 6>       //dataOut is yet in R0
          </atomic>
          </process 4 6 12>
//:
          ret                                              <2 2 2><live none>
          </code>
          <process>
          <atomic><live none>                //not necessary
          move.l               inChannel,R1  <2 2 6>
          call                 read_byte     <2 4 6>       //result in R0
          move.b               R0,@dataIn(BP)<2 4 4>       //save in R1
          </atomic>
          </process 6 10 16>
//PAR
          <group>
//c2 := dataIn[0]
          <process><use R0>
          move.b               @dataIn(BP),R0<2 4 4>
          and.b                01h,R0        <1 2 4>//result in R0
          move.b               R0,@c2(BP)    <2 4 4><free R0>
          </process 5 10 12>
```

.... similarly for c4, c5, c6, c8, c9, c110 ....

```
//c11 := data[7]
          <process><use R0>
          move.b               @dataIn(BP),R0<2 4 4>
          and.b                80h,R0        <1 2 4>
          move.b               R0,@c11(BP)   <2 4 4><free R0>
          </process 5 10 12>
          </group>
```

Figure 3. VIS code corresponding to the Hamming decoder of fig.2.

Timing characterization is also performed with a fine granularity to improve the freedom of the scheduler to choose the point where to break processes. The left-most tags of each VIS instruction contain the computation of **<min-delay max-delay bytes>** where the first two items are the delays to execute the operation according to the target CPU and **bytes** is the corresponding memory occupation. Up to now, according to the most common types of embedded system microprocessors, effects concerning pipelined instruction execution or parallel fetch have not been considered. For an analysis on how these issues can be managed for DSP applications, see [12].

During software synthesis, processes as well as the operating system microkernel are directly assembled into VIS code. As reported above, the software system is composed of processes and of a kernel basically operating as a context “switcher”: although no sophisticated

mechanisms for memory protection are necessary, particular attention has been devoted to the software section responsible for communication by adopting *ad-hoc* solutions to suit each specific circumstance. Our software synthesis system has to implement two different communication schemes: software to software, hardware to software (and viceversa). Processes communication takes place through buffered channels that will be implemented according to the type of data protocol and the hw/sw binding of the source and target processes.

Protocol implementation of complex data types is defined in terms of composition of basic types, such as BOOL, BYTE, INT16 (16-bit integer). The needs for communication involving the system bus have been reduced during the system partitioning phase since, under the scheduling algorithm viewpoint, the bus is a shared resource that will originate a *critical section* within the software-bound process requiring its use, thus increasing the difficulty to determine a feasible schedule.

Even though the basic OCCAMII model is composed of direct, point-to-point, asynchronous channels, our implementation has been extended to provide also a broadcasting node by expanding its definition into a software process able to copy the datum on all the target channels. The channel is mapped onto a pair {memory variable, data ready flag} shared by all processes. Since communication rates can vary across different processes, no matter if they belong to the same hardware or software partition, appropriate FIFO buffering capability has been introduced. hardware/software interface is performed via memory mapped registers.

A parametrizable retargeting tool, able to map VIS code on different target CPU has been implemented and tested for a Motorola 68000 microprocessor family.

5. Concluding remarks

A design methodology (which is currently part of the TOSCA co-design environment) to synthesize the software-bound sub-systems of control dominated embedded applications has been presented.

The model adopted to represent the software has been discussed, based on the concept of Virtual Instruction Set. Such a level of representation has been adopted since it provides the following advantages with respect to the use of high-level languages:

- fine-grain predictability of software cost and performance;
- code retargeting;
- hw-sw co-simulation for different microprocessor families with minimal customization effort based on the use of VHDL models (see [15] for more details).

The operating system has been also discussed in the paper. A static schedule algorithm has been adopted, based on the use of the information derived from the VIS model to assure the fulfillment of real-time constraints.

Evaluation of these strategies has been performed on a number of medium size examples, allowing the identifi cation of the optimal solution in a reduced time. We are currently developing a large telecom example to test all features of the proposed approach. Furthermore we are extending the model to consider also pipelined microprocessors and to analyze the power consumption of the software as one of the figures of merit driving the co-design activity.

6. References

[1] W.H.Wolf, *Hardware-Software Co-design of Embedded Systems*, Proceedings of the IEEE, vol.82, n.7, July 1994.

[2] D.Harel, *Biting the Silver Bullet: Toward a Brighter Future for System Development*, IEEE Computer, pp. 8-20, January 1992.

[3] P.Chiodo, P.Giusto, L.Lavagno, H.Hsieh, K.Suzuki, A.Sangiovanni-Vincentelli, *Synthesis of software Programs for Embedded Control Applications*, in Proc. of 32nd ACM/IEEE Design Automation Conference, 1995.

[4] R.K.Gupta, G.De Micheli, *Constrained Software Generation for Hardware-Software Systems*, in Proc. of third Int. workshop on Hardware/Software Co-design, pp 56-63, Grenoble, Sept. 1994.

[5] G.De Micheli and M.G.Sami editors, *Hardware/Software Co-Design:* NATO ASI Series, Series E: Applied Sciences - vol.310, Kluwer Academic Publishers, The Netherlands, 1996.

[6] A.Balboni, W.Fornaciari, D.Sciuto, *TOSCA: a pragmatic approach to co-design automation of control dominated systems*, Hardware/Software Co-design, NATO ASI Series, Series E: Applied Sciences - vol.310, pp.265-294, Kluwer Academic Publisher, 1996.

[7] H.Jifeng, I.Page, J.Bowen, *Towards a Provably Correct Hardware Implementation of Occam*. Technical Report, Oxford University Computing Laboratory, 1994.

[8] D.D.Gajsky, F.Vahid, *Specification and Design of Embedded Hardware-Software Systems*, in Design&Test of Computers, pp 53-67, vol.12, n.1, Spring 1995.

[9] P. Chou, G. Borriello, *Interval Scheduling: Fine-Grained Code Scheduling for Embedded Systems*, in proc. of 32nd ACM/IEEE Design Automation Conf., 1995.

[10] Y.T.S. Li, S. Malik, *Performance Analysis of Embedded Software using Implicit Path Enumeration*, in proc. of 32nd ACM/IEEE Design Automation Conf., 1995.

[11] T.Bener, R.Ernst, A. Osterling, *Scalable Performance Scheduling for Hardware/Software Cosynthesis*, in Proc. of EURO-DAC'95, Brighton, UK, Sept., 1995.

[12] P.G. Paulin, C.Liem, T.C. May, S.Sutarwala, *Codesyn: A retargetable Code Synthesis Systems*, In Proc. of Int. Symposium on HLS, May, 1994.

[13] A.H.Timmer, M.T.J.Strik, J.L.van Meerbergen, J.A.G.Jess, *Conflict Modeling and Instruction Scheduling in code Generation for In-house DSP Cores*, in Proc. of 32nd ACM/IEEE Design Automation Conference, 1995.

[14] G.Gossens, J. van Praet, D.Lanner, W.Geurts, *Programmable chips in consumer electronics and telecommunications*, NATO ASI Series, Series E: Applied Sciences - vol.310, Kluwer, 1996.

[15] A.Balboni, W.Fornaciari, D.Sciuto, *Co-synthesis and Co-simulation of Control-Dominated Embedded Systems*, Int. Journal Design Automation for Embedded Systems, vol.1, n.3, July 1996, Kluwer Academic Pub., Norwell, MA, USA.

[16] C.Liu, J.W.Layland, *Scheduling algorithm for multiprogramming in a hard real-time environment*, in Journal of the ACM, 20 (1), pp. 46-61, 1973.

[17] A.K.Mok, *The design of real-time programming systems based on process models*, in Proc. of IEEE Real-Time Systems Symposium, Dec 1984, pp 5-17.