Platform based design from parallel C specifications

Ivan Augé, Frédéric Pétrot, François Donnet and Pascal Gomez

***Abstract*— This paper presents Disydent, a framework dedicated to System on a Chip (SoC) platform based design for shared memory multiple instructions multiple data (MIMD) architectures. A platform based design problem is a triplet *(system, application, constraints)* where the *system* is both an operating system and a hardware template that can be enhanced with dedicated co-processors. Our contribution is firstly the definition of a complete flow for platform based design, from application to integration including all necessary intermediate steps, and secondly a set of tightly bound, operational, tools to implement the flow.**

Disydent is based on 4 tools. DPN is a C library for describing Kahn Process Network-based applications. ASIM0 is a multiprocessor target platform running a micro-kernel. This platform can be enhanced with co-processors generated by the UGH high level synthesis tool. CASS is a high performance cycle-accurate simulator.

The main steps of the design flow are Kahn Process Network modeling, functional validation, design space exploration, high-level synthesis and temporal validation. The design flow starts by modeling the application as a Kahn Process Network. This initial description is done in C using the DPN library. The functional validation is performed by running the initial description directly on the host. Without modifying the initial description, the user can simulate a hardware/software partitioning by indicating the number of processors and the processes that are to be migrated to hardware. This simulation is done at the cycle-accurate level for the whole system, except for the migrated processes for which the user must provide estimated time models. The description of the processes that are selected for hardware implementation must be translated into a subset of C and then synthesized. This new description is still compatible with the DPN library, so it can be used for functional validation. The temporal validation is done at the cycle-accurate level using the initial description for the software processes and cycle-accurate models automatically generated from the C subset description for the hardware processes.

Disydent's strength relies on its formal Kahn Process Network model that ensures a behavior that is independent of the overall system scheduling, its fast cycle-accurate validation that is several order of magnitude faster than classical event driven simulators, and its single description of a process that is used as input of DPN, CASS and UGH.

***Index Terms*— Platform based design, System modeling, Cycle-accurate simulation, High-level synthesis.**

I. Introduction

THE term co-design is used in the literature every time a piece of software and a piece of hardware are designed together. Some approaches to co-design result in a large portion of hardware and a smaller one in software, and vice-versa. For instance [1], [2], [3], [4] take one loop nest and generate a systolic accelerator. In particular, [4] takes as input a parameterized static nested loop program described in Matlab and generates a network of synthesizable virtual processors. In [5], [6] the software part is reduced to the driver that allows to exchange data in the order demanded by the synthesized accelerator. In the opposite direction [7], [8], [9] extend existing processors with a few hardware modules and put efforts in the detection of hardware modules in the original software. In between stands work such as [10] and [11], in which an application is specified and a parameterized target architecture is defined. The focus is then on how to realize the application in both software and hardware, and how both worlds communicate.

The final implementation may be very different. It may be a PC with a FPGA board, an existing DSP board with dedicated coprocessors, an existing board with a processor and FPGAs, a system either on chip or on an *ad-hoc* board.

The underlying co-design techniques are quite different. [8], [9] use a VLIW approach that extracts the instruction level parallelism to generate a MISD machine. [1], [2] use spatial and temporal projections of the loop nest to generate a SIMD machine. [11], [12], [13] extract coarse grain parallelism to generate a MIMD machine, whose actual implementation may be shared or distributed memory. Finally, the implementation can be done from scratch or using existing hardware, or on top of an existing system that already includes hardware and software components.

This paper presents Disydent, a framework dedicated to System on Chip (SoC) MIMD shared memory platform-based design. The starting point of the approach is an application described as a set of communicating processes exchanging data exclusively through blocking, lossless, point to point, FIFO channels, known as Kahn Process Networks (KPN)[14]. The endpoint of the approach is an actual implementation that instantiates predefined and synthesized Intellectual Properties (IPs) on top of which runs a parallel application. The choice of the KPN model of computation[15] is motivated by the fact that the KPN behavior is deterministic. It only depends on the data flow and not at all on the scheduling as long as the blocking semantic of the channel access is preserved. This property is very interesting in co-design because the scheduling overhead can significantly vary when the number of processors changes or when a process migrates from software to hardware and vice-versa. The drawback of using KPN specifications is that it cannot describe all the applications. However, the class of applications covered by the KPN model is large. It contains all statically schedulable applications and many non statically schedulable ones.

The paper defines the platform based design problem from the point of view of a system integrator, then details the design flow using a Motion-JPEG decoder example and finally describes the tools.

II. Related work

The idea of doing system level design from C-like specifications is not new. A major early contribution was done by Gupta and De Micheli with HardwareC[12]. The entry point is a set of processes in HardwareC, a superset of C with specific communication primitives allowing the exchange of a single data or event. The target architecture is made of a single processor, memory and coprocessors. FIFO channels are used for communication between the software and the hardware. The authors present an algorithm that automatically allocate the processes on hardware or software based on high-level cost models for the processor and the communications. High-level

Manuscript received November 7, 2003, revised March 22, 2004, June 25, 2004, September 9, 2004 and November 24, 2004.

I. Augé and P. Gomez are with the LIP6 laboratory, Université Pierre et Marie Curie, Paris, France.

F. Pétrot was with the LIP6 laboratory, Université Pierre et Marie Curie, Paris, France. He joined the TIMA laboratory of the Institut National Polytechnique de Grenoble in September 2004.

F. Donnet was with the LIP6 laboratory, Université Pierre et Marie Curie, Paris, France. He joined the m2000 company in January 2004.

estimation does not accurately take into account cache effects, bus contention, bus arbitration policies, or the cost of the operating system. This approach does not support multiprocessor systems, although they are often useful in practice, because the nonrecurring engineering cost (NRE) can be lowered by using several existing processors, rather than specific hardware.

[11] presents Symphony, a distributed multiprocessor architecture with coprocessors on which CSP[16] applications are mapped. Their focus is on communication synthesis. They introduce *ad-hoc* parameterized hardware components to implement the hardware part as well as the software part of a hardware/software communication channel implementing the synchronous wait protocol. From a user point of view, the implementation is hidden behind the communication primitives, in C++ for the software part and in synthesizable VHDL for the hardware part. Mapping an application consists of assigning processes either to a processor or a coprocessor. This is a co-design platform generator, but no details are provided on how to use these primitives in a micro-kernel or how to evaluate the performances of the co-designed application.

More recently, the Polis[17] initiative outlined the interest of a clear separation between functionality and architecture. Polis uses a single model, based on Codesign Finite State Machines specified in Esterel, and it is dedicated to reactive systems. This model imposes constraints for defining a design flow going from specification to implementation. Polis is more dedicated to real-time systems and formal validation than to general purpose or signal processing applications and simulation. Furthermore, Polis is not a fully integrated environment (simulator, kernel, synthesis tools are external tools), and thus engineering work is required for the adaptation to these external tools.

Cadence VCC[18] extends Polis to other application domains. However, it relies on models provided by the user at the different levels of abstraction (functional, temporal estimation, cycle accurate). This makes VCC difficult to use in daily life because of the initial effort in model development, at least as long as it has not been accepted as a standard by the IP providers. An in-depth experimentation of this approach, also starting from a KPN description has been performed by Brunel *et al.*[19]. This experimentation has shown us that handling the various descriptions of each component was hardly manageable and difficult to understand for the application designer.

Metropolis[20] is an approach that aims at relaxing constraints of Polis, such as its discrete event model of computation used as input specification. It introduces a meta-model capable of capturing functionality, architecture, and the mapping of the former on the latter. The meta-model has precise semantics, allowing simulation and formal analysis. This approach is attractive, because it is general enough to handle many application domains, and allows formal reasoning. However, it suffers the same integration difficulties as its predecessor, because the burden of library development still rests on the designer's shoulders. Also the tight integration of all the tools used from specification to integration has yet to be proposed.

[11] proposes a platform generator based on a template, but not an easy way to develop applications on it. [12] proposes a restricted platform with synthesis tools, but not accurate performance estimation. [18], [20] cover the required modeling levels, from functional to temporal estimation to cycle-accurate. However, going from one level to another requires either different descriptions of the component or complex configuration parameters.

For the application designer, none of these approaches is a solution starting from the application specification to the final hardware integration. To solve this problem, we use a KPN specification for its deterministic property as in [19], a platform template to quickly target a solution, as in [11], a synthesis tool, as in [12] and the three simulation levels of [18] for functional validation, design space exploration and temporal validation.

III. DISYDENT PRINCIPLES

A. Definition of platform based design

The problem we face is to enhance an existing device (*i.e.* PDA, Cellular phone) by integrating a new application.

We define the platform based design problem as a *(system, application, constraints)* triplet in which the system already exists and the application is not supported by the system. By existing system, we mean a collection of hardware and software that run applications. However, the hardware can eventually be incrementally enhanced by adding hardware accelerators. Fitting into an existing system generally implies that hardware accelerators use the system frequency whenever possible for simplicity reasons, and compulsorily use the same target technology. Furthermore, software drivers and hardware components are needed for HW/SW communications. Finally, a functional and temporal validation is necessary for the new application within the system, especially when the target system is a SoC.

B. Inputs

To make an implementation of his *(system, application, constraints)* triplet, the user has to provide two inputs for describing the system and the application, the constraints are not Disydent inputs but design properties (area, duration of a treatment, ...) that the implementation must respect. The first input is a model of the application in the form of a restricted Kahn Process Network (KPN)[21] written in the C language. The restricted KPN is a set of sequential processes communicating through lossless FIFOs of finite size. The FIFOs have a single producer and a single consumer, the read primitive is blocking if the FIFO is empty, and the write primitive is blocking if the FIFO is full. Parks[22] proved that bounding the sizes of the FIFOs and having a blocking write conserve the determinate behavior property, however the restricted KPN may introduce deadlocks. In the rest of the paper, we keep using the name KPN for this kind of network. The second input is an extended system built by adding coprocessors to the basic ASIM0 system (see Figure 1). This system is a fairly standard architecture with components plugged on a bus and an operating system.

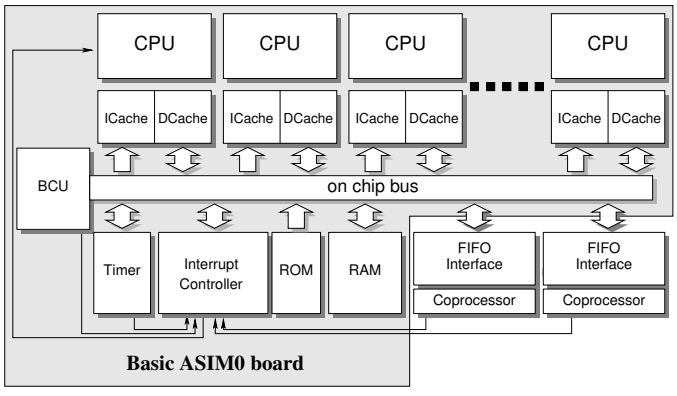

Fig. 1. Extended hardware.

The KPN specification does not permit the user to describe all the applications. At the user's level, the KPN input limits the communication to FIFOs. However, the FIFOs are implemented using solid, hand-optimized components that internally use shared memory, semaphores and interrupts and more complex communication schemes such as direct memory access (DMA). Designing by using this approach is quick, because it avoids repeated creation of similar communication schemes [19].

Nevertheless, we cannot prevent a user from using a hidden shared variable in the C description. Such a variable is usable if the processes that share it all end up in software. If it is not the case, communication and high level synthesis will not be able to implement it. Identically, a

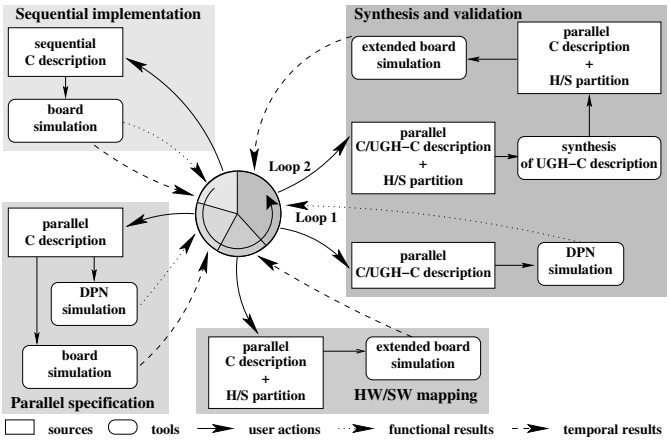

Fig. 2. Disydent design-flow.

hardware process can ignore the hardware FIFO protocol by writing to (or reading from) a FIFO without checking if the FIFO is full (or empty), or even abandon the deterministic world of Kahn by performing a select between several FIFOs.

The final system is an extension of the ASIM0 platform. This can be seen as a limitation, but the target board and system are the part of the specification. By setting the target system early in the design process the application designer is freed from hardware design, netlist creation, addition of drivers to operating systems, and software problems, such as cross-compilation of the application, library ports, co-simulation (e.g. VHDL and C), and so on. Furthermore, this reflects what is done presently in the industrial world. There are semiconductor houses who create platforms for the system houses. There exist also operating system houses that provide OS for the platform.

C. Tools

Disydent is based on 4 tools that permit the user to move from the functional specifications to the actual implementation. Each tool is briefly described below and detailed in Section V.

DPN is a C library that implements the FIFO communications of the Kahn model using the POSIX threads,

ASIM0 is a target platform (see Figure 1). The hardware is basically composed of one or more MIPS R3000 CPUs with instruction and data caches, a PI-Bus[23], interrupt controller, RAM and ROM, and can optionally be extended with FIFO interfaces and hardware coprocessors. For multiprocessor architectures, the memory coherency is ensured by the use of a write-through cache with bus snooping. The RAM and ROM sizes are parameters that should be adjusted depending on the application. All the components share the same clock, and the maximum frequency is 133MHz for a 0.35μm technology. Peak performances are 1 instruction per cycle for the R3000s and one transfer per cycle for the PI-Bus.

CASS is a cycle-accurate simulator[24]. CASS modules are written in C. Furthermore, each component of ASIM0 has a CASS simulation module.

UGH is a synthesis tool[25]. Its inputs are a C program and the clock frequency. It produces both a synthesizable VHDL model and a CASS simulation module.

IV. Design flow

The Disydent design flow is based on 4 main phases, as shown in Figure 2. The *sequential implementation* is a bare C program without any parallelism. The *parallel specification* is the application rewritten as a Kahn Process Network to exhibit coarse grain parallelism. In the *HW/SW mapping* stage, the designer looks for suitable HW/SW partitions among the processes of the parallel specification. Finally, *synthesis and validation* includes synthesis of processes mapped to hardware and the temporal validation of the triplet.

The design flow is not a fully automated approach in that the designer plays a central role in each phase. At every iteration, Disydent provides information that guides the designer to a solution and prevents him to enter a track that leads to a dead end.

A. The Motion-JPEG decoder example

In this paper, we illustrate the Disydent design approach with a video decoder. The decoder reads a stream of JPEG[26] images, called Motion-JPEG, from an input peripheral and writes pixels into an output ramdac. The general structure of JPEG decoding is shown on Figure 3. The *Motion-JPEG triplet* is defined as follows:
Motion-JPEG = (*system:* ASIM0 with an input stream peripheral and an output ramdac; *application:* Motion-JPEG decoder for 256-level grayscale images; *constraints:* frequency of 50 MHz, 25 frames/sec for 128x128 images, area of added hardware less than 35 mm^2).

We target a 0.35 μm technology. In this technology, the area of a MIPS R3000 with two 2 Kbyte caches occupies 15 mm^2, and 1 Kbyte of static RAM occupies 1 mm^2. The initial specification of the Motion-JPEG is a C program that reads the compressed images from a file and writes their pixel maps into a file.

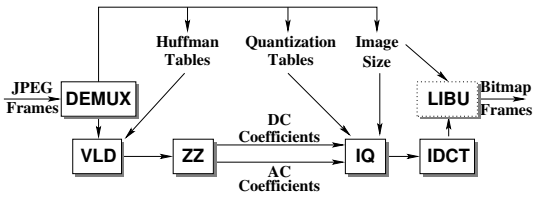

Fig. 3. Motion-JPEG Decoding principle.

1) DEMUX dispatches the input stream to the other blocks,
2) VLD performs a Huffman variable length decoding,
3) ZZ reorders the stream of coefficients,
4) IQ performs the inverse quantization,
5) IDCT performs the inverse discrete cosine transform,
6) LIBU is not a JPEG operation, but is necessary to adapt the pixel stream to a given output controller.

B. Sequential implementation

Goals One must enter the application into the system and then get profiling information to help in the next phases.

Designer work
The initial description is in C for general purpose computers. It is necessary to modify the input/output functions to adapt them to the I/O components of the target platform. In our example, we plug a traffic generator into the bus (camera, videophone, ...) and a ramdac (screen) to replace the I/O files.

Disydent run
The designer executes the sequential application on the target platforms simulated using CASS. CASS gives the duration in cycles of the run. If these timings fit the constraints, then the design is over. If constraints are close to being met, then it is worthwhile to optimize the application.
If not, the application is executed again while profiling information is gathered. The profiling information is the number of cycles spent in each function.

Example
We modify the C description to read the data from the stream generator and write the pixels into the ramdac. The execution takes $n = 124 \cdot 10^6$ cycles at a frequency of $f = 50$ MHz, the duration is $d = n/f = \frac{124}{50}$ or 2.5 seconds, so the number of frames per seconds is fps $= 25/d = 10$. So, we need a speedup of 2.5. The sequential implementation does not satisfy the

constraints, so the application is executed again to gather profiling information. The Motion-JPEG requires 6 steps, as illustrated Figure 3. Table I presents the percentages of CPU time spent in the main functions of the decoder. Table II presents the average amount of bytes exchanged

TABLE I
MOTION-JPEG SEQUENTIAL PROFILE

VLD	ZZ + IQ	IDCT	others
23%	22%	46%	9%

TABLE II
AMOUNT OF EXCHANGED DATA PER FRAME

input	→	DEMUX :	3024 bytes
DEMUX	→	VLD :	2691 bytes
VLD	→	ZZ+IQ :	65536 bytes
ZZ+IQ	→	IDCT :	65536 bytes
IDCT	→	LIBU :	16384 bytes
LIBU	→	output :	16384 bytes

per frame between the different functions. The DEMUX and VLD functions work on Huffman encoded data, and their size depends on the size of the image.
The entries in the table give the average size for the example used in the experiments of Section VI. ZZ, IQ and IDCT are working on macroblocks in the frequency domain, the size is constant (A 16×16 grid of 8×8 macroblocks with 32 bit coefficients). LIBU processes 128×128 pixel images, with each pixel represented by an 8 bit value.

C. Parallel specification

Goals To increase the performance, the application must be parallelized and/or pipelined. The Disydent approach advocates a MIMD solution using Kahn Process Networks, as formalized in [13]. To describe the KPN, Disydent provides the DPN library that implements the KPN communications, with the restriction that the FIFOs have a finite depth.

Designer work
The user defines a KPN of the application using profiling information gathered during the sequential execution and his knowledge of the application. He then describes the KPN in C. This last point consists firstly of restructuring the sequential source. Each Kahn process is described by a C function that has DPN FIFOs as parameters and interprocess communication is performed using solely the DPN I/O primitives (using global variables for this purpose is not allowed). Secondly, it consists of writing the main function that creates the DPN FIFOs and then runs the Kahn processes as POSIX threads.

Disydent run
The KPN application is compiled with the DPN library on the host machine, and executed on the host to check its behavior. Once the parallel specification is functionally validated, it must be embedded on the target platform. This is done by cross-compiling the KPN application, and by generating several platforms. The platforms differ by their number of processors. Due to the limited amount of parallelism, the execution time reaches a plateau for a given number of processors. Either the timing and area/power constraints fit, and the design is finished, or the constraints are violated, and the application is executed again and profiling information is gathered. The profiling information consists of the number of cycles spent in each process, the number of cycles spent in the system (context switches of the process and interprocess communications), and the bus load.

Example

Obtaining a KPN description from a sequential one is not an easy task. The main difficulty is due to the large number of possible solutions. For the *Motion-JPEG triplet*, one can search for parallelism and use an image as granularity (Figure 4), one can search for pipeline using this time the 8×8 pixel block as granularity (Figure 5). Of course, every combination of the former approaches is also possible (Figure 6). The parallelism at frame level is costly in memory, since each VLD+ZZ+IQ+IDCT process must be able to store at least an entire image and the FIFOs size must be around the image size. For a design that is pipelined at macroblock level, far less memory is required because the FIFOs size is around the block size, but the communication cost is much higher because there is a constant overhead in starting a communication and process wait more often for a communication to occur, resulting in more process context switches.
A full software solution based on Figure 4 needs at least three processors to meet the 25 images per second constraint. In a solution with three processors, 80% of each CPU processing power is required for decoding, so only 20% is available for synchronization and data exchange. This seems too tight to meet the timing constraint. A 4 processors solution would be more feasible, but the area constraint is violated ($3 \times 15 > 35$ mm^2). A solution based on Figure 5 at the frame level would not be much more efficient than the solution at the block level (due to the IDCT bottleneck) but would require 64 times more memory for the FIFOs. For these reasons, we will focus in the rest of the paper on solutions based on the KPN of Figure 5 at the macroblock level. The KPN of Figure 6 that is more promising will be used if no solution is found. We execute this graph on the host using DPN for functional validation. We simulate the execution of the KPN without modification of the ASIM0 system using CASS, then we enhance the ASIM0 system by adding processors and simulate it again. For each simulation CASS provides the duration in cycle, resulting in the frames per second figures presented in Figure 7.
These results show that with 1 processor, we need a speedup of 5 and with more than 1 processor we need a speedup of 3. From 2 to 7 processors, there is a 1.3 ratio between the performances of the sequential and KPN implementations. That is unexpected with such a KPN. With a processor devoted to each process, one can expect that the KPN implementation runs at least at equal speed and even faster than the sequential implementation. So these results need some investigation. The bus load is 31% for the sequential implementation and 82% (or 87%) for the KPN one on a 2 (or 7) processors board. For busses on which masters request ownership randomly, a bus load of more than 80% means bus saturation that induces large latencies. Concerning the cache misses, we have a miss rate of 7% for the sequential implementation and of 2% per processor for the KPN implementation on a 6 processors board. The amount of data transfered due to the misses in the 6 processors board is 2.15 ($\frac{2\times6}{7} \times \frac{10}{8}$, where $\frac{10}{8}$ is the ratio of execution times) times greater than in the uniprocessor board. The KPN communications increase the number of bus accesses and the number of cache misses, so the caches fight to get control of the bus and often wait and lock their associated processors. To reduce the number of cache misses, we simulate the execution of the application with larger caches. Figure 8 shows the results and the performance conforms to our expectations.

Legend:

- The bold arcs represent the decompression flow,
- The dotted arcs represent configuration parameters, described as global variables in the initial specification.

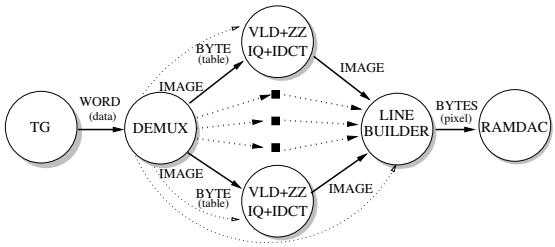

Fig. 4. Parallel KPN at the image level

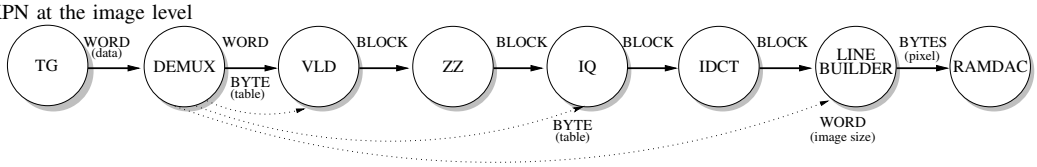

Fig. 5. Pipelined KPN at the block level

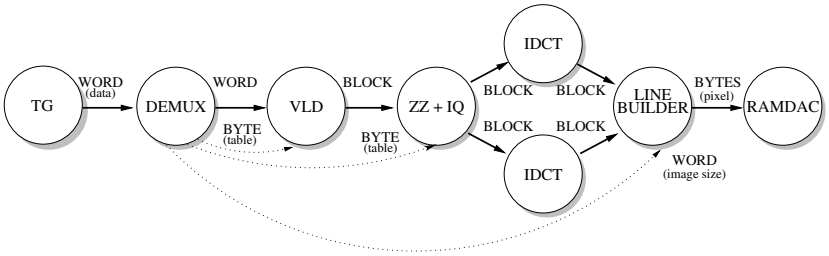

Fig. 6. Pipelined/Parallel IDCT KPN at the block level

The conclusion of these experiments is that hardware accelerators are still needed.
The previous results are obtained using cycle-accurate simulation that take into account communication, computation, system costs and cache misses. As shown by these measures, this level of accuracy is important in identifying bottlenecks that would be difficult to diagnose without it. For instance, for the Motion-JPEG triplet, a designer that uses a simulator that doesn't take accurately the cache misses in account would get results similar to those of Figure 8 instead of those of Figure 7. If the constraint was 20 frames per second, he would guess that optimizing the software would be enough to reach the constraint.
All the experiments presented in the rest of the paper are done with 2 Kbyte data caches.

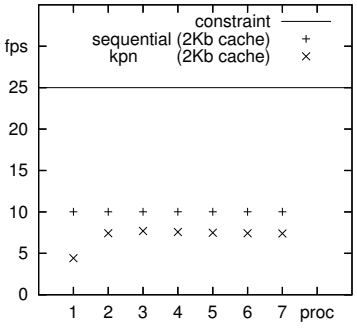

Fig. 7. Performance of the software implementations with small caches.

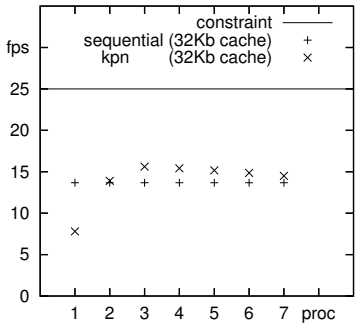

Fig. 8. Performance of the software implementations with large caches.

D. Hardware/software mapping

Goals Hardware design has a high manpower cost, so prior to starting a new design, it is necessary 1) to be sure that it will be useful and 2) to estimate the speedup necessary to satisfy the constraints. This second point is crucial because achieving a speedup of 5 or of 100 has different costs in manpower and in area. The problem is to find one or more groups of processes that are good candidates for hardware. To estimate the appropriateness of a selection, Disydent proposes an exploratory migration approach. It consists of hiding the C description of each process in the system within a FIFO interface. This is done through a wrapper that translates the DPN communication primitives into CASS primitives. Concretely, the migrated process performs at most only one read or only one write of a single item in each cycle. All computations between two I/Os

TABLE III
BUS LOAD OF GROUPS USING THE EXPLORATORY MIGRATION FOR NW SET TO 15 ON A UNIPROCESSOR BOARD

ALL	(VLD IQ IDCT)	(VLD IDCT)	(IDCT)
40%	33%	34%	40%

take NW (Number of Wait) cycles, NW being fixed by the user independently for each process. So, a NW of zero gives the upper bound on system performance assuming infinitely fast computation and communication performance. This NW is a user's estimate that will become a constraint for the high level synthesis tool.

Designer work
The user has to select a subset of the Kahn processes, based on profiling and intuition of fitness of a particular function for HW implementation. Furthermore, to get a system with these processes migrated to hardware, he must generate the wrappers for each process by running a utility that takes as parameters the FIFO names and access modes. He then extends the board description by attaching each process wrapper to a FIFO interface. Finally, the user modifies the main function of the parallel description by parameterizing the FIFOs that communicate with the hardware and suppressing the creation of the threads corresponding to the processes migrated to hardware.

Disydent run
The extended board running the application is simulated using CASS with different values of NW. Either the execution time seems to give enough margin to realize hardware that will respect the constraints (this decision can be taken only by a RTL designer) and the HW design can start, or the constraints are violated or very closely matched, and then a new group of processes must be selected. If no more groups are available, the overall system design should be restarted at the "parallel specification" step. If the user finds no new KPN, then the problem represented by the *Motion-JPEG triplet* has no solution.

Example
The former profilings indicate that the IDCT process must be hardwired, and that the VLD and IQ processes are also good candidates. Furthermore LIBU should not be hardwired because it needs a pixmap of eight times the maximum image width, which is costly in area. So we select four groups: ALL=(VLD IQ ZZ IDCT), (VLD IQ IDCT), (VLD IDCT), (IQ IDCT), (IDCT).
We simulate using CASS each group with different values of NW on a uniprocessor board. The results of the bus load for NW set to 15 are presented in Table III. They clearly show that the system bus is not a bottleneck. The results, in frames per second as a function of NW, are plotted Figure 9. Strangely enough, the number of frames per second tends to slightly increase when the time between two I/Os increases. This is due to changes in the scheduling of software processes that result in less context switches for these intermediate values. The almost constant curves begin to go downwards when NW is around 100. Figure 10 presents the number of frames per second as a function of the number of processors with NW sets to 0. These results imply that:

(IDCT), (IQ IDCT) no hardware latency can satisfy the constraint.
(VLD IDCT) no hardware latency can satisfy the constraint with 1 processor (we tried with 2 processors and we reached about 29 frames/second).
(VLD IQ IDCT) we almost satisfy the constraint of 25 frames per second, so this solution can be considered.
ALL we get more than 40 frames/second, so this solution can be considered too.

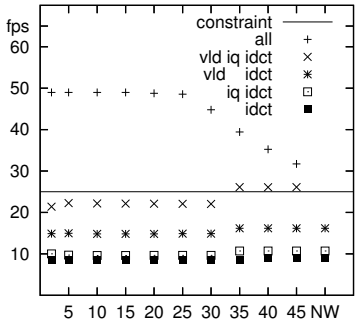

Fig. 9. Performance of groups using the exploratory migration on a uniprocessor.

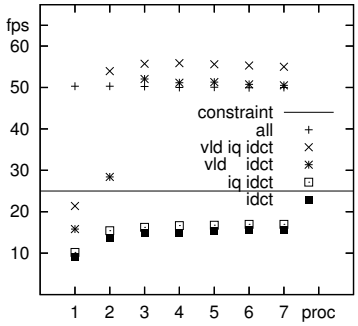

Fig. 10. Performance of groups using the exploratory migration with $NW = 0$.

A RTL designer knows that these three designs are easy because Figure 9 shows that he has approximately 50 cycles to perform a computation between two I/O operations. This is largely enough because either the complexity of the computation is low (VLD[27] IQ) or its parallelism is such that it can be exploited at the hardware level (IDCT[28]). To conclude, there are three potential solutions: (VLD IDCT) with 2 or more processors, (VLD IQ IDCT) with 1 processor and ALL with 1 processor. For these designs, the emphasis should be more on area optimization than on delay optimization.

E. Process synthesis

Goals Now it is time to design the coprocessors selected previously. Note that at this point the time constraint for each process is well known. To help the designer, Disydent provides the synthesis tool UGH.

Designer work
The designer must adapt each process description to UGH. He must eliminate pointers, define a draft of the data-path (DDP) and modify the C Kahn Process description so that

so that each C variable has an associated register in the DDP, unless its assignment can be inlined in the statements that use it.
This adaptation is relatively easy from a syntactic point of view, unless the process description uses many functions with pointer parameters. However, the above adaptation is often inefficient, and a new hardware oriented algorithm needs to be defined. This induces a complete rewriting of the process. Given the state of the art in behavioral synthesis, this cannot be avoided. Furthermore, it is much easier to rewrite the process for UGH than to convert to synthesizable VHDL or SystemC.

Disydent run
The C description of the process for UGH is still compatible with the DPN primitives, so it can be executed on the host machine and simulated on the target platform using CASS. This allows one to check quickly the behavior prior to the actual synthesis (see loop 1 in Figure 2). The synthesis of the process is performed for the frequency of the platform to get a cycle-accurate CASS module. The platform is extended with the modules of the hardwired processes and the application is run on this new platform to get the real timings (see loop 2 in Figure 2).
If the timings fit the constraints, UGH is run again to get the structural VHDL description. This description must be synthesized using classical CAD tools to obtain the cost (area, power, ...). If both timing and cost constraints are met, the design is finished. Otherwise, the designer has to enhance either the micro-architecture of the hardware processes, choose different processes to implement in hardware, or even choose a better KPN partitioning of the application.

Example
We adapt the VLD, IQ and IDCT C descriptions, and validate them using DPN. Then we run UGH using a 50 MHz frequency to generate the cycle-accurate CASS modules. We then plug the modules into the board and simulate the application using CASS.
For the solution *"(VLD IQ IDCT) with 1 processor"*, we get **28 frames/second** and **32 frames/second** for the *"(VLD IDCT) solution with 2 processors"*.
We run UGH again to get the RTL VHDL description of the coprocessors and synthesize them with CAD tools using a $0.35\mu m$ technology. We get 13.2 mm^2 for the VLD, 1.2 mm^2 for the IQ and 10.9 mm^2 for the IDCT coprocessors.
For the solution *"(VLD IQ IDCT) with 1 processor"*, we need 3 hardware FIFOs (FIFO interfaces as shown in Figure 1) of 64 coefficients of 4 bytes and 1 hardware FIFO of 64 pixels of 1 byte, that is to say less than 1 Kbyte so around 1 mm^2. The area increase is 26.3 mm^2 $(13.2 + 1.2 + 10.9 + 1)$. Similarly, for the *"(VLD IDCT) solution with 2 processors"*, the area increase is 39.6 mm^2 $(13.2 + 10.9 + 0.5 + 15)$ that violates the area constraint.

F. Outputs

The outputs of Disydent are one or several solutions to a platform based design problem. More precisely, for each solution, this consists of the main software that initializes the hardware components and starts the software tasks, the C code of the software tasks and for the hardware components, the synthesizable VHDL descriptions and the cycle-accurate CASS models.

The choice of the solution and its implementation into the final product is done by the system integrator taking into account both technical parameters and commercial issues.

V. TOOLS

A. DPN

For the Kahn process network description, the user has on one hand to create the FIFOs and the processes and on the other hand to describe the behavior of each process.

The FIFOs are created using the `channel channelOpen(width, depth)` DPN primitive, where *width* is the item width in bytes and *depth* is the number of slots of the FIFO channel.
The process creation is performed using the `"pthread_create"` primitive of the POSIX threads. So from the user point of view, each behavior is a C function that takes a single array argument. Each element of the array is a channel descriptor.
The behavior of a process is written in C with 2 DPN primitives to exchange data through the FIFOs. Reading is done using `channelRead(channel, buffer, nItems)`, that reads *nItems* FIFO items from channel descriptor *channel* into the buffer starting at *buffer*, and writing is done with `channelWrite(channel, buffer, nItems)`, that writes *nItems* FIFO items to the channel referenced by the channel descriptor *channel* from the buffer starting at *buffer*. The size of the item is defined at channel opening time. The functions return only when *nItems* items have been transfered.

Using DPN is not difficult. However, not every application is suited to a parallel implementation (i.e a VLD cannot be decomposed in an efficient KPN). Also the time needed to transform a sequential description into a DPN network depends more on its coding style than on the intrinsic complexity of the application.

B. ASIM0 system

The ASIM0 system is composed of synthesizable VHDL models of the communication components and of the MUTEX micro-kernel [29]. The communication components are the PI-Bus[23] controller, the interrupt controller, the slave FIFO, the master FIFO, and the point to point hardware FIFO. The synthesizable VHDL models of the processor and its caches are not provided.

Most components comply with the VSI Alliance Virtual Chip Interconnect (VCI[30]) standard. Since the ASIM0 system uses a PI-Bus as an interconnect back-bone, Disydent also provides the VCI/PI-Bus wrappers.

1) Hardware: In ASIM0, all components share the same clock and the system is therefore fully synchronous.

Processor and caches
The processor is a MIPS R3000 as described in [31]. It has separate instruction (read-only) and data (read/write) caches. The caches are direct mapped, and use a write-through policy. The number of cache blocks and the size of the block can be set independently for both caches. The depth of the write buffer is also a parameter of the data cache. Since ASIM0 is based around a shared bus, the data cache maintains memory coherency by snooping the writing transfers on the bus, and invalidating the block in case of a hit. The memory consistency issue is solved by waiting for a transfer to be acknowledged prior to starting another transfer (as usual with buses, unless split transactions are allowed).

Interrupt controller
This module, pictured in Figure 11, groups up to 32 interrupt inputs lines into 1 output line. Each line consists of an interrupt request signal and of an interrupt acknowledge signal. At run time, it is possible to change the interrupt vector attached to a line, and to individually mask, set and clear an interrupt request.

FIFO As shown in Figure 12, the FIFO component is connected to the coprocessor and the system bus. Its parameters are the number of input and output FIFOs. An input FIFO allows the coprocessor to fetch a datum from the system

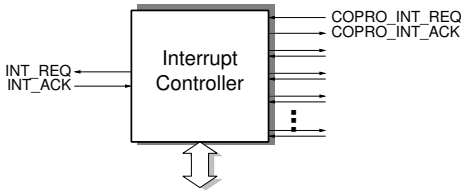

Fig. 11. Interrupt controller.

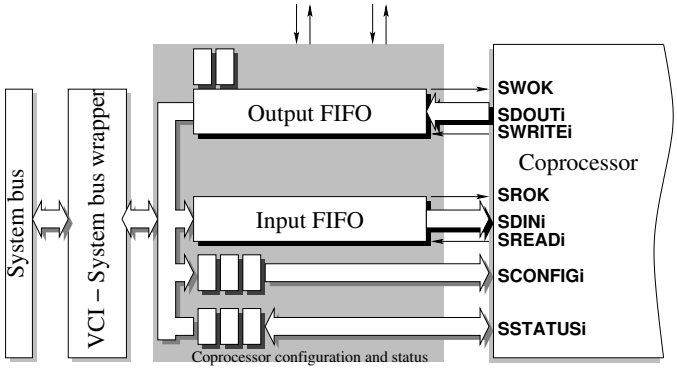

Fig. 12. FIFO interface overview.

bus. An output FIFO allows the coprocessor to emit a datum to the system bus.

From the coprocessor side, the data read action from a FIFO is shown Figure 13. In the READ state, the coprocessor asserts the SREAD signal and loads the SDIN signals' data into an internal register. If SROK is not asserted, it means that the FIFO was empty, and the state must be run again because the loaded value is not significant. Otherwise the value loaded on SDIN is significant, and it is popped out of the FIFO (because SREAD is asserted). The writing action is similar to the

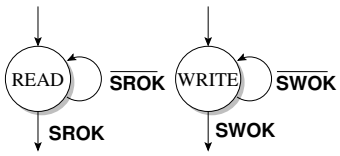

Fig. 13. Usual FSMs for reading and writing FIFOs.

reading action.

From the bus side, each FIFO may be either a slave or a master. The processors explicitly read or write the slave FIFOs. The processors must configure the master FIFOs by writing parameters into memory mapped registers, then the master FIFOs work like a DMA controller.

The latency is 2 cycles for the slave FIFO and 5 cycles for the master FIFO. For burst transfers, a throughput of 1 item per cycle is achieved in both cases.

Point-to-point FIFO communication

This interface allows direct communication between two coprocessors without using the bus. It implements FIFO queues and provides the same interface as the slave and master modules.

2) Software:

Multi-thread and multiprocessor kernel

Disydent provides a lightweight kernel that implements the POSIX threads API in kernel mode. This kernel comes in two kinds.

1) Symmetric multiprocessor, that allows the migration of processes from processor to processor, depending on the availability of the resources. This flavor assumes that the cache coherency is ensured by the hardware.
2) Asymmetric multiprocessor, for which each process is assigned to one processor. Local and shared memory are distinguished so as to allow for more optimized access (through the cache) to local data and still correct access (uncached) to shared data.

Both kinds assume relaxed memory consistency, i.e. a lock protecting a data will not be unlocked before the data is updated[32].

Communication services

A communication layer that performs the exchange of data between processes is available for processes implemented as either hardware or software. The SW/SW communication is basically the same as the DPN version. The HW/SW and SW/HW communications use semaphores and interrupts. The HW/HW communications that occur through the bus need little software intervention, usually only to set the transfer address, length and repetitive behavior at initialization time. In practice, several alternatives can be used for a given communication type, depending on the amount of data to be transfered. A precise explanation of these services can be found in [33].

C. CASS

CASS is a cycle-accurate simulator for register transfer level models. It allows the user to simulate an interconnection of components described as a VHDL structural netlist. Conceptually, a CASS model is a Finite State Machine (FSM) with a specially identified clock input. To be simulated by CASS, a component must be modeled by the three C functions detailed below. An example is provided Figure 14.

Instantiation and registration function

It is called once at the beginning of simulation to define the interface and the internal resources of the component. If the component is a Mealy FSM, it also declares the input ports that may modify the output values.

Sequential function

It is called at every cycle and it executes the behavior of the current cycle depending on the internal state of the component and the current input signals. It implements both the transition function and the output function that only depends on the state (Moore outputs). This function uses a fire-and-return modeling approach, in which the state is explicit and the function must return, as opposed to an approach in which wait operations are embedded in the code,

Combinational function

It is called at the end of a cycle to compute the Mealy outputs. It does not exist if the component has only Moore outputs.

The CASS simulation algorithm does not propagate events[34]. Instead, it uses a static scheduling technique that ensures firstly that the evaluation order of the simulated components is not data-dependent and secondly that no model will be executed more than once during a clock cycle, provided there are no combinational loops between models [35]. This restriction could in principle be avoided, leading to multiple evaluations of combinational parts of the components. However a compile time ordering strategy still allows one to minimize the number of re-evaluations [36]. This strategy is possible because the Mealy outputs are given by the combinational functions and the structural netlist indicates the relationships between the modules. There are actually few models that have Mealy dependencies, and even fewer belong to a combinational loop, so the time spent in the evaluation of the combinational part of a system is usually negligible.

This compile time approach is also usable to simulate systems that have multiple clock domains if and only if the ratios of all frequency pairs are rational numbers. This constraint is due to the fact that CASS must be able to compute the least common multiple of all clock frequencies to statically schedule the system.

The use of a precompiled schedule achieves high simulation performances for complex systems, and this is particularly useful when booting a kernel and debugging software.

```
#include <cass.h>
enum _state {GCD_READA,
             ...,
             GCD_WRITE};
typedef struct {
  enum _state  state;
  port         resetn;
  port         read, rok, din;
  port         write, wok, dout;
  unsigned int a, b;
} GCD;

GCD *CreateGcd(void)
{
    GCD *gcd = malloc(sizeof(*gcd));
    /* Registers the ports */
    ...
    return gcd;
}
```

```
#include "gcd.h"
void SequentialGcd(GCD *gcd)
{ /* Transition function */
  if (!CassRead(gcd, resetn)) {
     gcd->state = GCD_READA;
  } else
     switch (gcd->state) {
     case GCD_READA :
        if (CassRead(gcd, rok)) {
           gcd->a     = CassRead(gcd, din);
           gcd->state = GCD_READB;
        }
        break;
     ...
     case GCD_WRITE :
        if (CassRead(gcd, wok))
           gcd->state = GCD_READA;
        break;
     }
  /* Generation function */
  switch (gcd->state) {
  case GCD_READA :
  case GCD_READB :
     CassWrite(gcd, read,  1);
     CassWrite(gcd, write, 0);
     break;
  ...
  case GCD_WRITE :
     CassWrite(gcd, read,  0);
     CassWrite(gcd, write, 1);
     CassWrite(gcd, dout, gcd->ra);
     break;
  }
}
```

Fig. 14. CASS model of a GCD coprocessor.

D. UGH

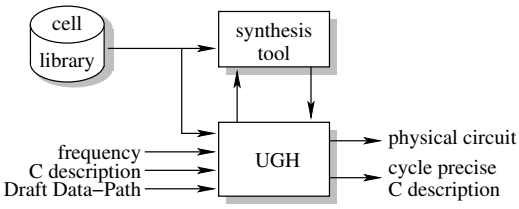

Fig. 15. UGH User view.

```
#include <ugh.h>
/*** the communication channels ***/
ugh_inChannel32   inFIFO;
ugh_outChannel32  outFIFO
/*** registers ***/
uint32 a,b;
/*** behavior ***/
void ugh_main(void)
{
 while (1) {
   ugh_read(inFIFO, &a);
   ugh_read(inFIFO, &b);
   while (a != b) {
     if (a < b) b = b-a;
     else       a = a-b;
   }
   ugh_write(outFIFO, &a);
 }
}
```

Fig. 16. UGH-C for Euclid's GCD algorithm.

Disydent provides the User Guided High Level Synthesis tool for the synthesis of control-dominated coprocessors[25].

1) User view: The user view of UGH is presented Figure 15, it shows that UGH needs to be tightly connected to a synthesis tool, (currently Synopsys) and that it also needs a library of characterized standard cells.

We take Euclid's GCD algorithm to illustrate the use of UGH.

a) Inputs: The first input is the frequency of the target coprocessor.

The second input (see Figure 16), is the C description of the coprocessor. The entry point is the `ugh_main` function. The `ugh_inChannelxx` and `ugh_outChannelxx` types define communication ports compatible with the hardware FIFO components. The `ugh_read` and `ugh_write` functions generate the automata shown in Figure 13. The standard C types are allowed but UGH also defines the `intxx` and `uintxx` to precisely size the variables and the registers that will contain them. Most C constructs are allowed, but pointers, recursive functions and the use of external functions (e.g. printf, strcat, ...) are forbidden. Furthermore, all variables must be either global or static unless their assignments can be inlined in the statements that use them.

The last input (see Figure 17.a) is a simplified structural description of the target data-path called Draft Data-Path (DDP). The DDP is a directed graph (Figure 17.b) whose nodes are functional or memorization operators and whose arcs indicate the authorized data-flow among the nodes. For instance, the 2 arcs that point to the `a` input of the `Subst` node indicate that in the final data-path, the bits of this input can be driven by: a) constants, b) bits of the `q` port of the `A` register, c) bits of the `q` port of the `B` register, d) any bitwise combination of the former cases. Furthermore, note that the DDP neither sets the bit size of the operators associated to the nodes, nor sets the bit size of the arcs.

The C input and the DDP are interdependent. A global or static variable (respectively: array) of the C input must correspond to a register (respectively: register file or static ram) of the DDP having the same name. For each statement of the C input there must be at least a sub-graph of the directed graph that can execute the statement.

b) Outputs: UGH generates a VHDL structural description of the circuit. The top level entity is composed of a Moore finite state machine and a data-path. The finite state machine is described in synthesizable VHDL, the data-path is a structural description whose leafs are standard cells of the input cell library.
UGH also generates the simulation module in C for the CASS simulator that exactly reproduces the cycle by cycle behavior of the circuit.

c) Options: UGH supports several synthesis directives. The two major ones are presented below.
UGH accepts a partially connected DDP. In this case if there is no sub-graph in the DDP graph to implement a statement of the C input, UGH will automatically create one. So with this option the minimal DDP only contains the memorizations and the functional cells. Our experiments show that using minimal DDPs produces circuits with an area (respectively: an execution speed) a few percent larger (respectively: slower) than using fully connected DDPs.
By default, an "if" statement of the C input is micro-controlled, and becomes a state with two transitions in the FSM of the target circuit. UGH provides a pragma to hard wire a "if" (speculative computation). In this case, the "if" is executed combinatorially within the data-path using added hardware. All the "if" statements can be wired knowing that wiring an "if" automatically wires all its nested "if"'s.

2) Synthesis steps: The synthesis process, presented in the Figure 18, is split into 3 main steps: First, the Coarse Grain Scheduling (CGS) is run, resulting in allocation and translation of C statements into RTL instructions; then the mapping is performed to get the physical data-path and the temporal characteristics; finally the Fine Grain Scheduling (FGS) is run, resulting in the scheduling of the RTL instructions taking as constraints the annotated timing delays of the data-path.

a) CGS: CGS starts with a consistency check. Enough registers must have been instantiated to store all the non-trivial variables. Each statement of the C description must correspond to at least one

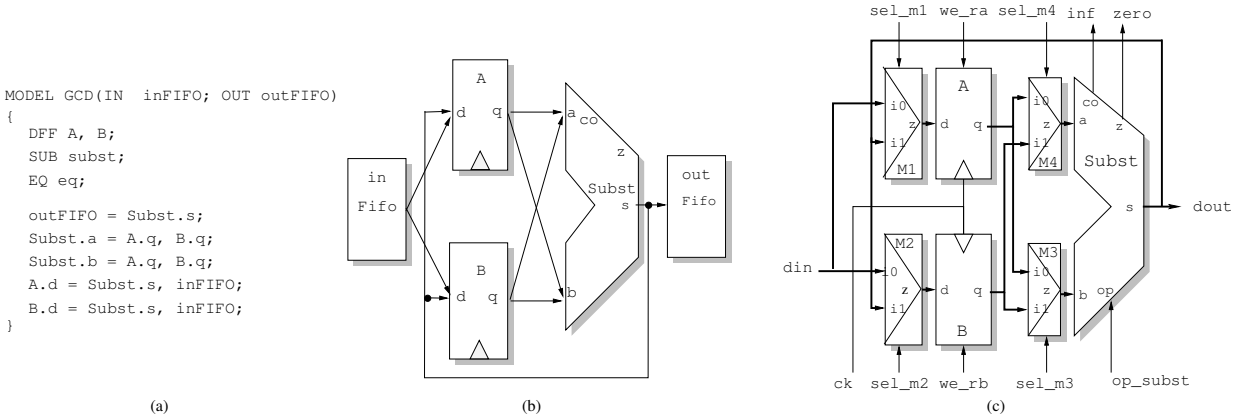

Fig. 17. Draft Data-Path of the GCD example.

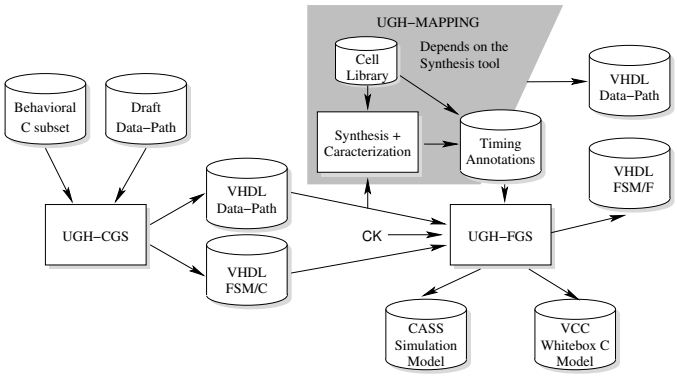

Fig. 18. Main scheme of UGH.

sub-graph of the DDP.
Then the binding begins: Each node of the DDP corresponds to a cell of the data-path, its bit size it deduced from the bit size of the C variables, the input connectors of the cells are connected to output connectors either directly or using a multiplexer when inputs are driven by different sources. The resulting data-path of the GCD example is shown in Figure 17.c.
Finally the *coarse* FSM is elaborated, where *coarse* means that the operations are only partially ordered like in soft scheduling[37].
The algorithm used in CGS must choose a DDP sub-graph for each C statement and then *coarsely* order them. These choices and this ordering are done by maximizing the intrinsic parallelism while trying to reduce the data-path area. The degrees of freedom for reducing the area are the minimization of the input numbers of the added multiplexers and the binding of operations of the same type and similar bit sizes to the same node. Its temporal constraints are: multipliers need 2 cycles, adders and subtracters need 1 cycle, and all other functional cells have negligible propagation times. This algorithm is detailed in [38].

b) Mapping: The mapping step is seldom described in the literature. The synthesis tools most often generate a VHDL standard cell netlist. The circuit is obtained by placing and routing the VHDL netlist. The generated circuit will probably not run at the expected frequency. The main reasons are that the FSM has been constructed with estimated operator and connection delays, and that often the FSM is a Mealy one and its commands may have long delays. Furthermore it is also possible that the circuit does not run at any frequency if it mixes short and long paths. This happens frequently in circuits having both registers and register files.
Of course, these problems also occur with designs done by hand: in that case the designer solves them by adding states to the FSM, adding buffers to speed up or down some paths. This is not easy, and it takes time, but it is possible because he has an intimate knowledge of the design. After high level synthesis, these problems can not be corrected because the designer has lost the knowledge of the design.

From our point of view the mapping is an issue that must be dealt with, and not a minor one, because the generated circuit must run as it comes out of the tool. If it is not the case the synthesis tool is simply unusable.

In UGH, the mapping consists of generating the behavioral VHDL with its constraints (*i.e.* maximum fan out for the connectors) for each operator of the data-path, and shell scripts to automatically run the synthesis of the whole data-path using a cell library. The execution of these scripts invokes the Synopsys Design Compiler to generate structural VHDL files respecting the given constraints. UGH starts up again, reading all the structural VHDL files, parsing the file describing the cell library to get their timing equations and computes the propagation delays, the setup and the hold times of all the operators of the data-path.
Of course this step may be quite long, several hours for large processors. For this reason, UGH gives the possibility to bypass the mapping during design tuning and use pessimistic estimated delays.

c) FGS: FGS adapts the coarse FSM to the characterized data-path to ensure that the circuit will run at the given frequency. Once the FSM is computed, back-ends generate the cycle-accurate models for CASS and the VHDL FSM.

FGS extracts the register transfer instructions from the coarse FSM and then reschedules them taking into account the propagation delays, the setup and hold times of the cells and the intrinsic parallelism supported by the data-path. This algorithm is detailed in [39].

3) Main points: Like classic HLS algorithms UGH supports multi-cycle operators. For instance, if the statement "$A = B_1 \times B_2$" is part of the input and if the given clock period is smaller than the propagation times from the B_i to A plus the setup time of A, the statement is scheduled in several cycles. Similarly for the statement "$A = B_1 + B_2 + B_3$", the data-path contains a chain of two adders. If the clock period is greater than all of the propagation times from B_i to A plus the setup time of A, the statement is scheduled in one cycle.

Like classic HLS algorithms, operators chaining is supported too. Furthermore, for the same statement, for the given clock period T and the propagation times p_i from B_i to A plus the setup time of A, if $2\,T \leq p_i \leq 3\,T$, the statement is scheduled in 3 cycles. So UGH supports the multi-cycle chaining as opposed to classical HLS schedulers that would introduce a register to save the result of the first adder and schedule the statement in 4 cycles.

UGH correctly handles pipelines. For instance in the sequence "$A = 1; B = A + B; A = 2$", the "$A = 2$" assignment will be scheduled in the same cycle than "$B = A + B$" if the

performance constraints of the data-path allow it. Furthermore, even wave pipelining [40] is handled.

Finally, UGH also supports multifunction operators, for instance if there are the statements "$A = B + C; A = A - 1$" in the C inputs and only one adder/subtracter is given in the DDP, it is shared by the 2 instructions.

The HLS tools usually propose an iterative approach to explore the design space. The user runs the synthesis, the result being the FSM graph and various cross-reference tables (between states and source statements, between cell and source statements, ...). Then, using pragma in the source file, the user can force specific allocations. He runs again the HLS synthesis to get the new results and so on until he obtains the expected design. This iterative approach is difficult to use primarily because: 1) For large designs the time between iterations is too long, 2) The tables are difficult to interpret. The analysis of the results to set judicious pragma requires to rebuild the data-path from the cross-reference tables, and this is a very long and tedious work. 3) This latter work must be done again at each iteration. So the iterative approach is unsuitable for large designs.

UGH is only aimed at VLSI designers. The designer does not have to change his working habits. He provides a data-path and a FSM, the only difference is that for UGH a data-path draft is needed (DDP) and that the FSM is a C program. So designers can obtain designs very close to the designs they would do by the hand.

Most of the HLS tools let the low level synthesis adapt the data-path to the frequency. This approach neither ensures that the circuit can be generated (low level synthesis tools can not respect the clock frequency) nor ensures that the generated circuit runs at the given clock frequency (even at any) if the circuit mixes short and long paths. Furthermore, this approach generates very large circuits when the low level synthesis tools enter into speculative computation techniques.

Taking an opposite view, UGH adapts the FSM to the data-path. This has three advantages over the usual approach. Firstly, the data-path is small because we do not give specific constraints about the critical path to the low level synthesis tool. Secondly, any frequency is reachable if all the memorization cells of the data-path support it and if the FSM synthesis tool can synthesize the resulting FSM for this frequency. Thirdly, our approach only acts on the scheduling, therefore it only modifies the FSM by adding a few states, so the area cost is null or very low.

The only disadvantage is that communication with the external world must be synchronized with handshaking because the communication cycles are set by FGS and depend on the electrical characteristics of the data-path and the target frequency.

Finally the last point is the comparison of the execution speed of the resulting circuits. When UGH generates a circuit in which it adds states to the FSM, an execution needs more cycles, and thus the circuit is less efficient than the circuit generated by the tools that rely on logic synthesis to optimize the data-path performance. Nevertheless, if the added states are not in the main execution loop or if they are in the main loop but the main loop has a lot of states and/or many branches, the execution speeds will be very similar. Furthermore, if UGH added states, it is because there is at least one path in the data-path whose propagation time is greater than the given clock period. In this case, the usual approach will give this period as a constraint to the low level synthesis tool. The risk here is to compare the execution speed of a unsynthesized virtual circuit (the low level synthesis tool has not been able to satisfy the constraint or has generated too large of a circuit) with a working one.

VI. Results

Currently, Disydent is operational and allows one to solve actual platform based design problems. The benchmark used is the Motion-JPEG. The input stream contains 25 images. Experimentation was done on a 1.7 GHz PC running Linux.

Running the sequential Motion-JPEG application directly on the host takes 5 ms. Running the DPN Motion-JPEG application with the pthread linux implementation as backend takes 20 ms. The simulation times of the extended platform for several configurations are presented in Table IV. For these simulations, all CASS models of the bare ASIM0 platform are cycle-accurate and bit-accurate, the NW parameter of all the exploratory migration models is set to 0 and the models generated by UGH are cycle-accurate and bit-accurate. The simulation times range from 5 minutes to 15 minutes, and are acceptable for design space exploration. Thus, it is possible in one day to perform functional and temporal verifications for movies of around 100 seconds on a ASIM0 platform with UGH coprocessors. This times must be compared with the VHDL simulation of the same board that we estimate to 3000 minutes (3000 = 20 x 150: 20 is the number of components of the board; 150 is the time spend for the VHDL gate level simulation of the synthesized VLD coprocessor.).

TABLE IV
CASS cycle-accurate bit-accurate simulation performance

	nb proc.	CPU times	cycles/ second
sequential	1	5m 36s	268142
KPN software	1	17m 43s	268257
	5	11m 52s	234647
E M: ALL	1	05m 07s	95242
E M: VLD IDCT	1	11m 04s	130122
	2	06m 50s	119764
E M: VLD IQ IDCT	1	09m 23s	104659
UGH: VLD IDCT	1	16m 48s	71685
	2	9m 16s	70191
UGH: VLD IQ IDCT	1	15m 27s	60697

E M: simulation using the exploratory migration

UGH: simulation using the model synthesized by UGH

We synthesized 900 lines of C code and obtained coprocessors requiring 25 mm^2 and 952000 transistors in a 0.35μm technology. The synthesis times for the coprocessors are given in Table V. The 'UGH' column gives the time spent to generate the cycle accurate simulation model using default delays. These times are acceptable for micro-architectural design space exploration. The 'delay computation' column shows the CPU time needed by Synopsys to generate the data-path and the time spent by UGH to extract the delays.

TABLE V
UGH performance

	C lines	UGH	delay computation sum = *UGH* + *Synopsys*
VLD	200	56.1s	126m41s = 4m40 + 122m01s
IQ	100	0.9s	10m25s = 0m05 + 10m20s
IDCT	600	11.2s	80m50s = 0m54 + 79m56s

Donnet[38] has compared UGH with the CoCentric SystemC Compiler of Synopsys that, like UGH, is a control dominated HLS tool. The VLD and IDCT circuits generated by UGH are 2 times smaller and 1.3 and 1.5 times faster respectively.

He has also compared UGH to GAUT[41] (a data-dominated HLS compiler) on the IDCT description that is completely data-flow (Table VI). The IDCT circuit generated by UGH is 2 times smaller but the GAUT circuit is 2.5 times faster. In one day of designer work he rewrote both the C input and the DDP of the IDCT. The result of UGH with this new description is smaller and faster than the circuit generated by GAUT. This proves that UGH allows the designer to get precisely the expected design.

VII. Conclusion

The configuration of Disydent for a new platform is a complicated task. Firstly, it requires the definition of a basic system (hardware components, communication schemes, operating system, ...). Secondly, the design at the RTL level of the hardware components

TABLE VI
COMPARISON OF IDCT SYNTHESIS

	Execution time μs	Area mm^2
Cocentric	34.5	19.9
Gaut	9.2	19.0
Ugh	24.9	10.9
Ugh-revised	7.8	18.4

is needed. Thirdly, a CASS simulation model must be written for cycle-accurate simulation. However, once this is done, the application designer just extends the platform and tunes it for the application.
The SoC design approach we propose is efficient. The user input being a C program using the KPN primitives, the functional validation is performed on the host with very high performance: real time decoding of MPEG2 streams is possible for a small image size. This is several orders of magnitude faster than the performance of RTL-level simulators. The temporal validation is one to two orders of magnitude faster than classical event driven simulators (VHDL or SystemC).
The approach is also simple, because the functional validation at the KPN level is sufficient to ensure the functionality of the application on the target platform. This is due to the KPN property that ensures that the relative execution speed of the processes does not influence the global behavior[14].
Last but not least, the tight integration of the tools allows one to develop the Motion-JPEG example in a few weeks.

- The description of the Kahn Process Network is runnable as is on the local host, and also on the platform.
- The description of the Kahn Processes can be synthesized by the Disydent synthesis tool. It generates directly a cycle-accurate simulation module, as opposed to other synthesis tools that most often generate VHDL gate netlists. To simulate a synthesized coprocessor within the whole system, co-simulation between the cycle-accurate simulator and the VHDL simulator is needed. This is not very efficient and it requires engineering work.
- The exploratory migration does not require any engineering work. It allows the designer to quickly estimate the temporal constraints on a coprocessor, taking into account the communication delays.

The most negative aspect is the rewriting of the process following UGH style. This requires one to have two descriptions of the same object. Although the UGH-C may be used instead of the initial DPN description, its performance when running in software is significantly worse. So, both forms are really needed. Algorithms for hardware are very different from algorithms for software. The hardware form exhibits both hardware parallelism through a sequential description that complicates the code, and many bit level computations that are easily realized in hardware but are lengthy to perform in software.

ACKNOWLEDGMENTS

We would like to thank Denis Hommais for his participation at the beginning of this work. We also would like to thank the anonymous reviewers of the Transactions for their numerous and constructive remarks that have greatly improved the quality of this paper.

REFERENCES

[1] P. Quinton and V. van Dongen, "The mapping of linear recurrence equations on regular arrays," *Journal of VLSI Signal Processing*, vol. 1, no. 2, pp. 95–113, Oct. 1989.
[2] J. Xue and C. Lengauer, "The synthesis of control signals for one-dimensional systolic arrays," *Integration, the VLSI journal*, vol. 14, no. 1, pp. 1–32, 1992.
[3] P.-Y. Calland and T. Risset, "Precise tiling for uniform loop nests," in *Proc. of Int. Conf. on Application Specific Array Processors*, 1995, pp. 330–337.
[4] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere, "Laura: Leiden architecture research and exploration tool," in *Proc. of the 13th Int. Conf. on Field Programmable Logic and Applications*, Lisbon, Portugal, Sept. 2003, pp. 911–920.
[5] S. Derrien, A. C. Guillou, P. Quinton, T. Risset, and C. Wagner, "Automatic synthesis of efficient interfaces for compiled regular architectures," in *Proc. of Int. Samos Workshop on Systems, Architectures, Modeling and Simulation*, Samos, Greece, July 2002.
[6] P. Coussy, "Synthèse d'interface de communication pour les composants virtuels," Ph.D. dissertation, Université de Bretagne Sud, LESTER, Dec. 2003, (in french).
[7] G. Goossens, J. V. Praet, D. Lanneer, W. Geurts, and F. Thoen, *Programmable Chips in Consumer Electronics and Telecommuncations – Architectures and Design Technology*. Kluwer Academic Publishers, 1996, pp. 135–164.
[8] S. Aditya, B. R. Rau, and V. Kathail, "Automatic architectural synthesis of VLIW and EPIC processors," in *Proc. of the Int. Symp. on System Synthesis*, 1999, pp. 107–113.
[9] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood, "Lx: a technology platform for customizable VLIW embedded processing," in *Proc. of the 27th Int. Symp. on Computer architecture*. New York, NY, USA: ACM Press, 2000, pp. 203–213.
[10] P. H. Chou, R. B. Ortega, and G. Borriello, "The chinook Hardware/Software co-synthesis system," in *Proc. of the Int. Symp. on System Synthesis*, Cannes, France, Sept. 1995, pp. 22–27.
[11] S. Vercauteren, B. Lin, and H. D. Man, "Constructing application-specific heterogeneous embedded architectures from custom HW/SW applications," in *Proc. of the Design Automation Conf.*, Las Vegas, Nevada, June 1996, pp. 521–526.
[12] R. K. Gupta and G. D. Michelli, "Hardware-software cosynthesis for digital systems," *IEEE Design and Test of Computers*, vol. 10, no. 3, pp. 29–41, Sept. 1993.
[13] 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 signal processing systems," in *Proc. of the 37th Design Automation Conf.*, June 2000, pp. 402–405.
[14] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. of Information Processing 74*, Stockolm, Sweden, Aug. 1974, pp. 471–475.
[15] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis," *Proc. of the IEEE*, vol. 85, no. 3, pp. 366–390, Mar. 1997.
[16] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.
[17] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C.Passerone, A. Sangovnanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-software co-design of embedded systems: the Polis approach*. Kluwer Academic Publishers, 1997.
[18] M. Santarini, "Cadence adds system-level design tool to eda flow," *EE-Times*, Jan. 2000, http://www.cadence.com/technology/hwsw/ciertovcc/articles/.
[19] J.-Y. Brunel, W. M. Kruijtzer, H. J. H. N. Kenter, F. Pétrot, L. Pasquier, E. A. de Kock, and W. J. M. Smits, "Cosy communication ip's," in *Proc. of the 37th Design Automation Conf.*, Los Angeles, CA, June 2000, pp. 406–409.
[20] F. Balarin, H. Hsieh, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, and Y. Watanabe, "Metropolis: An integrated environment for electronic system design," *IEEE Computer*, vol. 36, no. 4, pp. 45–52, Apr. 2003.
[21] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proc. of the IEEE*, vol. vol. 83, no. 5, pp. 773–801, May 1995.
[22] T. M. Parks, "Bounded scheduling of process networks," Ph.D. dissertation, Electronics Research Laboratory, Berkeley, 1995.
[23] T. N. et al, "Draft Standard OMI 324: PI-Bus," Open Microprocessor Initiative, Tech. Rep., Dec. 1996, rev. 0.3d.
[24] F. Pétrot, D. Hommais, and A. Greiner, "Cycle precise core based hardware/software system simulation with predictable event propagation," in *Proc. of the* 23$^{\mathrm{rd}}$ *Euromicro Conf.*, Budapest, Hungary, Sept. 1997, pp. 182–187.
[25] I. Augé, R. K. Bawa, P. Guerrier, A. Greiner, L. Jacomme, and F. Pétrot, "User guided high level synthesis," in *VLSI: Integrated Systems on Silicon*, R. Reis and L. Claensen, Eds., IFIP. Gramado, Brazil: Chapman & Hall, Aug. 1997, pp. 464–475.
[26] J. committee, http://www.jpeg.org/, JPEG is standardized in ISO/IEC IS 10918-1/2.

[27] M. Bakhmutsky, "High-performance variable length decoder with two-word bit-stream segmentation," in *Proc. Digital Compression Technologies & Systems for Video Communications*. Berlin: SPIE Vol. 2952, Oct. 1996, pp. 634–640.
[28] C. Loeffler, A. Ligtenberg, and G. Moschytz, "Practical fast 1-D DCT algorithms with 11 multiplications," in *Proc. Int. Conf. on Acoustics, Speech, and Signal Processing*, 1989, pp. 988–991.
[29] F. Pétrot, P. Gomez, and D. Hommais, "Lightweight implementation of the posix threads api for an on-chip mips multiprocessor with vci interconnect," in *Embedded Software for SoC*, A. A. Jerraya, S. Yoo, D. Verkest, and N. Wehn, Eds. Kluwer Academic Publisher, Nov. 2003, part 1, chapter 3, pp. 25–38.
[30] O.-C. B. D. W. Group, "Virtual component interface standard version 2," VSI Alliance, Tech. Rep., Apr. 2001, oCB 2 2.x www.vsia.org.
[31] G. Kane and J. Heinrich, *MIPS RISC Architecture*. Prentice Hall, 1992.
[32] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proc. of the 17th Int. Symp. on Comp. Arch.* ACM, May 1990, pp. 15–26.
[33] D. Hommais, F. Pétrot, and I. Augé, "A practical toolbox for system level communication synthesis," in *Proc. of the 9th Int. Symp. on Hardware/Software Co-design*, Apr. 2001, pp. 48–53.
[34] G. Jennings, "A case against event driven simulation for digital system design," in *Proc. of the 24th Annual Simulation Symp.*, New Orleans, LA, Apr. 1991, pp. 170–175.
[35] F. Pétrot, D. Hommais, and A. Greiner, "A simulation environment for core based embedded systems," in *Proc. of the 30th Int. Simulation Symp.*, Atlanta, Georgia, Apr. 1997, pp. 86–91.
[36] D. Hommais and F. Pétrot, "Efficient combinational loops handling for cycle precise simulation of system on a chip," in *Proc. of the 24th Euromicro Conf.*, Vesteras, Sweden, Aug. 1998, pp. pages 51–54.
[37] J. Zhu and D. D. Gajski, "Soft scheduling in high level synthesis," in *Proc. ot the 37th Design Automation Conf.*, New Orleans, June 1999, pp. 154–157.
[38] F. Donnet, "Synthèse de haut niveau contrôlée par l'utilisateur," Ph.D. dissertation, Université Pierre et Marie Curie, LIP6, Jan. 2004, (in french).
[39] I. Augé, F. Donnet, and F. Pétrot, "Retiming finite state machines to control hardened data-paths," in *Proc. of the 16th Symp. on Integrated Circuits and Systems Design*, So Paulo, Brazil, Sept. 2003, pp. 41–47.
[40] C. T. Gray, W. Liu, and R. K. C. III, "Timing constraints for wave-pipelined systems," *IEEE Trans. Computer-Aided Design*, vol. 13, no. 8, pp. 987–1004, Aug. 1994.
[41] C. Jego, E. Casseau, and E. Martin, "Architectural synthesis of digital signal processing applications dedicated to submicron technologies," in *IEEE International Conference on Electronics Circuits and Systems, ICECS 01*, Malte, Sept. 2001, pp. 533–536.

Ivan Augé received the engineer degree in Computer Science from "Conservatoire National des Arts et Métiers", Paris, France, in 1983, and the PhD in computer science from the same university in 1990. He joined Philips research labs in Paris from 1990 to 1994 where he worked on the Alma compiler, a high-level synthesis tool and lossless data compression adapted to networks.

He joined the engineering school Institut d'Informatique d'Entreprise, Evry, France in 1995 as an Assistant Professor in computer science. His research interests include computer-aided design of integrated systems, high-level synthesis, operating systems and local area network configuration.

Frédéric Pétrot received the DEA degree in Computer Science and Electrical Engineering from Université Pierre et Marie Curie, Paris, France, in 1989, and the PhD in computer science from the same university in 1994. He has been Assistant Professor in Computer Science at Université Pierre et Marie Curie until September 2004. From 1989 to 1996, Prof. Ptrot was a main contributor of the open source Alliance VLSI CAD system. Since 1996, he works on the definition and implementation of the Disydent system-level design environment.

He is now professor of computer architecture at the Institut National Polytechique de Grenoble. His main research interests concern computer-aided design of VLSI circuits and system architecture, with a particular emphasis on system integration, kernels and multiprocessor SoCs.

François Donnet received the DEA degree in Computer Science and Electrical Engineering from Université Pierre et Marie Curie, Paris, France, in 1999 and the PhD for the same University in 2004. During all this period, he contributed to the Alliance VLSI CAD system and to the Disydent co-design tools. Dr. Donnet is currently research engineer at the m2000 company. His research interests are in high-level and logic synthesis for ASICs and FPGAs.

Pascal Gomez is a PhD candidate in the department of computer science at Université Pierre et Marie Curie, Paris, France. His research interests include system level design and operating systems for multi-processor SoC. Gomez received the DEA degree in Computer Science and Electrical Engineering from Université Pierre et Marie Curie, Paris, France, in 2000.