Application Domain-Driven System Design for Pervasive Video Processing

Zbigniew Chamski, Marc Duranton
Philips Research, Eindhoven, The Netherlands
{zbigniew.chamski,marc.duranton}@philips.com

Albert Cohen, Christine Eisenbeis, Paul Feautrier
INRIA, Rocquencourt, France
{albert.cohen,christine.eisenbeis,paul.feautrier}@inria.fr

Daniela Genius
Université Paris 6, Paris, France
daniela.genius@lip6.fr

Abstract Pervasive video processing in future Ambient Intelligence environments sets new challenges in embedded system design. In particular, very high performance requirements have to be combined with the constraints of deeply embedded systems, frequently changing operating modes, and low-cost, high-volume production. By leveraging upon the key properties of the application domain, we devised a computation model, a hardware template, and a programming approach which provide a natural mapping from application requirements to a complete system solution. Our approach enables the direct exploitation of concurrency and regularity in achieving the combined challenge of adaptability, performance, and efficiency.

Keywords: SANDRA, video processing, timed process networks, hierarchical architecture, piecewise static control.

1. Introduction

Vision plays a dominant role in human perception, placing pervasive visualization and video processing at the heart of the Ambient Intelligence concept. The underlying properties of adaptability, anticipation, and ubiquity make the video processing sub-systems operate in a changing environment, and require run-time flexibility. While video processing is essentially regular, user interactions and communications with other devices introduce variability in operation modes, system loads, and quality-of-service requirements.

In media streaming applications of un-encoded data (before encoding/after decoding), most events can be predicted and anticipated. Whenever the execution latency of individual tasks can also be predicted (or imposed), asynchronous (interrupt-triggered) control can be entirely eliminated, leading to a fully predictable, real-time system. This in turn enables a tighter dimensioning of the system, reducing the difference between average and peak performance, and thus, directly increasing its efficiency.

At the same time, upcoming display technologies and ever improving compression standards enable a dramatic increase in content and display resolutions. The corresponding performance requirements are beyond the reach of general-purpose processor architectures, implying the use of domain-specific, multi-processing solutions and an increased system complexity. Yet to become commercially viable, these solutions must additionally satisfy the criteria of silicon efficiency (area and device utilization, power dissipation), affordable system design effort, and low manufacturing costs.

In the SANDRA (Stream Architecture eNgine Dedicated to Real-time Applications) project, we tackled this challenge using a global approach driven by the key characteristics of the application domains (video pre and post-processing): massive amounts of parallelism, piecewise regular processing of structured data, predictability of events, multiple processing rates, and explicit temporal requirements in applications. These characteristics were used to identify a suitable computation model, which in turn determined many aspects of system hardware and software. Ultimately, this led to the definition of a system template and a programming flow in which the requirements of the applications are driving the entire design process. The requirements of embedded systems are also taken into account: the SANDRA system is designed to be silicon efficient, satifying hard real-time constraints and having the lowest possible power consumption and memory bandwidth.

The scalability of the architecture is also important to cope with various applications and instances of embedded systems: a base SANDRA sub-system can be easily extended with new blocks using an intra- or inter-chip network, while still using the same model and representation of applications: it is seen as a single entity, with higher performances. The “connected” nature of Ambient Intelligence systems enables to think of even more sophisticated systems: for example a SANDRA system inside a camera could use the resources of another SANDRA system in the TV set for increasing its computational power during, e.g., video segmentation or depth reconstruction (if a right communication channel is available).

To address the complexity of programming such inherently concurrent systems, we propose to move away from sequential application descriptions towards *timed process networks* [5], which are much more suitable to our target application domain. Process networks directly capture the concurrency available in the applications, and temporal annotations attached to processes (or groups thereof) provide a natural way of representing the timing requirements of the applications. It also helps in distributing tasks onto separate instances of the system, allowing networking at the SoC level and at the multi-chip/multi-device level. The introduction of the quantitative time representation, important for real-time guarantees, also helps in characterizing the communication links (bandwidth, latency, buffer requirements). The resulting system can process data ”on-time” and not necessarily as fast as possible, allowing to determine the slowest possible clock required for performing each function, and thus reducing the global power consumption.

The hierarchical organization of the SANDRA architecture reflects the structure of both applications and data they manipulate. From the programmer's point of view, applications are seen as computations on different levels of data structures. From the system design point of view, the hierarchy of control and communications exploits temporal and spatial locality to enforce storage and bandwidth requirements. It also helps addressing the issue of on-chip signal propagation delays.

Another challenge of the project was to support the hard real-time requirements of "live" stream processing in combination with the dynamic reconfiguration inherent to most Ambient Intelligence applications. We propose to address this issue through piecewise static control: the scheduling and mapping of computations and communications in SANDRA is made statically for a range of "scenarios" defined by the different levels of maximum load guarantees and throughput/latency requirements. Within each scenario, the schedules and resource allocations guarantee the respect of performance and resource *requirements* while ensuring the best possible *efficiency*.

This paper is further organized as follows: section 2 introduces the application domain and the representation of applications used in SANDRA. Section 3 presents the overall structure of the SANDRA compilation chain. Key issues in code generation are presented in Section 4. Section 5 describes the key system architecture concepts of SANDRA.

2. Representation of Applications

When designing a domain-oriented system suitable for a range of applications, the characterization of the application domain is a key success factor: it makes possible to exploit application properties in an efficient way.

The target domain of SANDRA is real-time media stream processing. We provide a tentative solution to the system design issues in presence of:

- massive amounts of parallelism;
- piecewise regular processing of structured data;
- predictability of events;
- multiple processing rates;
- explicit temporal requirements in applications.

The application model must capture the concurrency and real-time attributes of the applications and of the SANDRA hardware. In addition, the applications operate on structured data whose size has to be taken into account in the model. When combining a machine description of the target system with the information of an application's degree of concurrency, data size, clock rates and hierarchy, it is possible to determine the peak and average bandwidth values, end-to-end and partial latencies, intermediate buffer sizes, utilization rates of target system elements, etc.

2.1 Multi-Periodic Process Networks

To enable fast retrieval of time and resource properties at every stage of the design process, we developed a process-based application model called Multi-Periodic Process Networks (MPPN; a detailed presentation and discussion of the model can be found in [5]). The MPPN model is inspired by Kahn process networks [10], Petri nets variants such as event graphs [2], and by Control/Data-Flow Graphs (CDFG, [15]). It also shares some motivations with the COMPAAN project [11] within the PTOLEMY environment [4].

The MPPN provides four distinctive concepts: (1) explicit synchronizations between processes, (2) bounded-size communication channels, (3) a quantitative notation for delays, latencies and periods of processes, and (4) a hierarchical composition mechanism for building aggregate processes from elementary ones. In particular, the two latter features of MPPN are fundamental when distributing applications onto networks of SANDRA instances. The hierarchical composition helps in partitioning, and the quantitative modeling of delays and latencies allows the MPPN network analysis tools to check if the communication channels are suited to the proposed partitioning. Figure 1 shows a simple MPPN for a two-dimensional polyphase filter, applied to the downscaling of video frames from a high definition (1920×1080) to a low definition

(720×480) screen; the filtering process is decomposed into a horizontal stage (sub-process P_5) and a vertical stage (sub-process P_6).

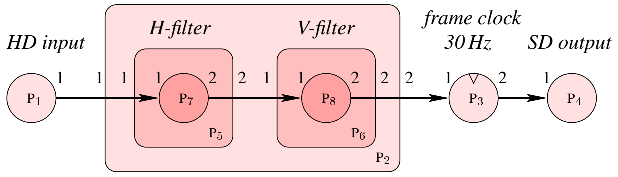


Figure 1. MPPN representation for a downscaler

The HARRY verifier tool that we have built checks the coherency of the input data and timing constraints and computes the required buffer sizes, process latencies, bandwidth and resource usage. It uses an XML representation of a MPPN (cf. Figure 2). This representation is designed such that an MPPN can easily be abstracted from a SALLY program (see 3.1). At present, it handles cyclic networks and clocked processes but not the splitter/selector extensions.

We tested MPPN models for five typical applications: downscaler with a polyphase filter, horizontal split-screen display (splitter and selector), picture-in-picture (full example with deep hierarchy), advanced anti-aliasing filter (pipelined execution and complex acyclic graph), and noise correction (cyclic graph). Using HARRY, we solved the network's equations to check for soundness and compute the missing parameters. Finally, we deduced resource requirements (memory sizes, bandwidths, functional unit counts).

The MPPN model was designed to leverage upon the key properties of the domain of pixel-stream processing (predictable behavior, regular processing, extended stream semantics with a steady state, timing and bandwidth constraints, few data dependent control loops). Its main limitations are the fixed token size and the restricted splitter/selector semantics. Token sizes are bounded to allow the design of predictable systems, possibly leading to a worst case design (this is also the case with ASIC design.) The splitter/selector model can hardly be extended without losing static schedulability, but applications that have non-deterministic (or data dependent) splits and selects also have an upper bound in their activation frequency, allowing for an approximate MPPN model. Extensions of the model are possible, at the expense of a big complexity increase. We therefore preferred to use simplifications and perhaps less optimal solutions to model the few applications that escape from the common characteristics of the domain.

3. Compilation Chain

The compilation process consists in mapping the timed process network representation of the application to the hierarchy of control, memory, and process-

```
<!DOCTYPE MPPN SYSTEM "MPPN.dtd">
<MPPN>
  <!-- The first process -->
  <Process id="1"
           Type="Normal">
     <Name>HDInput</Name>
     <OutPort ChannelID="1"
               Start="true"
               End="false"
               Q="2073600">
            <Bandwidth/>
            <Message/>
              <Access a="1"/>
              <Latency l="0"/>
              <Burstiness/>
     </OutPort>

     <!-- Process parameters -->
     <Period/>
     <Burstiness N="1"/>
     <Latency/>
     <PipelinedExecution/>
     <Activation/>
  </Process>

  <!-- ... -->
```

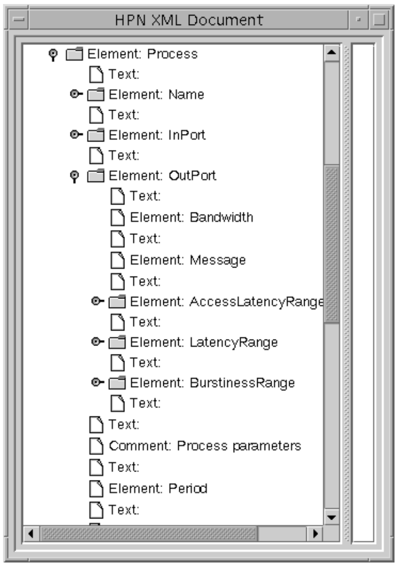


Figure 2. Sample XML representation and parse tree

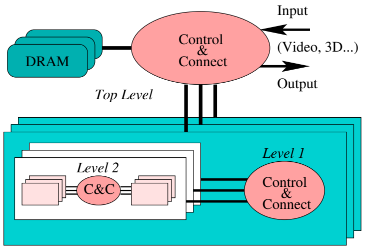


Figure 3. Hierarchical control and storage

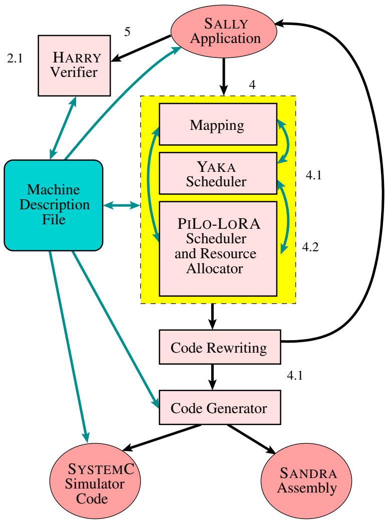


Figure 4. SANDRA compilation chain

ing units which form a SANDRA instance. Both the mapping of an application to the SANDRA architecture and the validation of the resource constraints for this application rely on a model of system. The model provides quantitative target system models at multiple levels of refinement and precision. By combining this information with the timed process network representation of the *requirements* of the application, it is possible to carry out performance estimations in a systematic way, even at early design stages of either applications or the target system.

The purpose of the compilation chain of SANDRA is threefold:

- providing timed simulation models to be used in design-space exploration and debugging of functional and temporal behavior of SANDRA applications;
- generating code and parameters for general-purpose and dedicated units, for each controller, at each layer of the architecture; this must be fully automatic because the execution model is too complex to be handled at the application level;
- optimizing the code such that the time constraints are satisfied while minimizing memory, computation and communication resources; tedious optimizations and transformations are automatic, but the engineer can still drive the design space exploration using an abstract process-network model of the application.

In the high-level synthesis community, Control/Data-Flow Graphs (CDFG) have been a successful representation for data-intensive applications with timing and resource constraints [15]. Indeed, CDFGs can be simulated for design-space exploration, they serve as a basis for optimizing transformations, and of course, they enable code or circuit generation. Well-known research tools such as HYPER [15] or PTOLEMY [4] (with alternative data-flow graph models) have been developed in this area.

Several compiler techniques developed for resource-constrained scheduling of loop nests can reconstruct the control and data structures completely through algebraic loop-nest transformations [9, 16]. These techniques can distinguish between each iteration of a loop or each value of a stream/array, enabling much more aggressive transformations.

However, the better efficiency of such techniques comes at a price: some of the versatility of CDFGs and other flow-graph approaches is lost, like the ability to handle arbitrary control flow or the natural integration of timing and resource constraints. But since our applications do not rely on arbitrary control flow, and despite the lower versatility and the higher computational complexity, we believe that only aggressive techniques can efficiently harness the resources of the highly parallel architecture.

The structure of the SANDRA compilation chain is sketched in Figure 4, where numbers link transformation phases and code representations to the relevant sections. Compilation starts with an application description specified in the high-level language SALLY (section 3.1) and checks real-time properties with the HARRY verifier (section 2.1). During the design space exploration, the YAKA multidimensional affine scheduler (section 4.1) and the PILO-LORA software-pipelining tool (section 4.2) produce one or several schedules and resource allocations of the concurrent program; the programmer may drive the exploration in suggesting a coarse-grain mapping of (some) processes to SANDRA controllers. The code rewriting phase (section 4.1) regenerates SALLY code from the abstract schedule, allowing for iterative refinement of the schedule. Finally, cycle-accurate simulation code and SANDRA assembler code are generated from the fully scheduled SALLY program. All communications between software modules are done via XML files, while the tools use their own internal formats.

To accommodate the flexibility of the hardware template, software tools supports parameterization by a machine description file that contains the information necessary for the code generation stage. It also seamlessly interacts with HARRY's evaluation of communication latencies, parallelism, buffer and bandwidth requirements, and the results may be fed back into machine descriptions of higher-level operations. Eventually, the machine description file feeds YAKA and PILO-LORA for resource allocation information, enables the automatic generation of simulation models, and provides a reference for regression testing of the actual hardware. This pervasive use of the machine description is a major governing principle in the SANDRA architecture and compilation chain.

3.1 The SALLY Language

In order to capture both functional and non-functional requirements of applications at the program level, we designed a small, domain-oriented language called SALLY. Rather than extending a sequential language such as C, C++, or Java, we propose to use a clean set of concepts tailored to the characteristics of the application domain, and we provide constructs which are familiar to domain specialists. This in turn makes it possible to formalize application requirements, while still allowing the compiler to perform domain-specific analyses, verifications, and optimizations.

SALLY is a declarative language. At the core of SALLY is the synergy between structured data types (arrays and records), iterators, and processes.

Variables in SALLY are *streams* of array or scalar values, indexed by iterator values. Each variable has an associated *index domain* (possibly unbounded), which identifies all index values for which this variable is defined [13]. SALLY variables correspond directly to channels in MPPN.

Iterators are a uniform concept for expressing loops (either parallel or sequential) and event-based processing. Three types of iterators are available: *indices* correspond to unordered, potentially concurrent iterations; *counters* correspond to ordered, i.e., serial iterations; *clocks* are counters with quantitative time distribution, they are used to capture the real-time requirements of the application. Processes are sets of equations or networks of other processes that are evaluated in response to a change of value of a special input, called the trigger, mapped to an iterator defined outside the process.

The basic statements of SALLY are equations and process activations. Statements may be explicitly associated with an iterator. An *equation* defines the value of a variable as the result of evaluating an expression in the current context of iterator values. A process definition consists of an *interface definition* and a *body* : the body lists the local variables of the process followed by its equations and subprocesses, whereas the interface definition provides type signatures and names for the input/output ports, along with per-invocation parameters of the process. A process activation instantiates the process, binds the ports and parameters of the process with actual variables, and maps the *trigger* of the process to an actual iterator.

SALLY programs can express parallelism in three ways:

- by triggering multiple statements/processes on the same iterator (there is no explicit sequential ordering; instead, the dependencies are extracted and checked at compile time);
- through unordered iterators (`for all i do...`);
- through array-wide operators.

The first method provides a natural expression of control parallelism, while the last one is specifically directed at data parallelism. The unordered iterators provide a means of trading control parallelism for data parallelism.

Example: *Figure 5 shows a short excerpt from a* SALLY *implementation of a two-dimensional polyphase filter.*

```
extern clock frameStart       30Hz
clock        output_line_clk  660 @ FrameStart (* visible + Vsync *)
clock        visible_line_clk output_line_clk[100 .. 579]

node Vstage(param float VFL_coefs[64][6], param int VFL_off[64][6],
          input  pixel frame_after_HFL[1080][720])
{
 decls
   (* after Vfilter: 480x720 pixels *)
   pixel frame_after_VFL[480][720]

 code
   (* frame-level V filter invocation *)
   frame_after_HFL -> VFL_stage(VFL_coefs, VFL_off) -> frame_after_VFL
      every frameStart

   (* output ctrl at line level *)
   frame_after_VFL[visible_line_clk - 100] -> OUTPUT -> VOID
      every visible_line_clk
}
```

Figure 5. SALLY application example: filtering and output

The first three lines define iterators (clocks) used by the `main` *process:* `frameStart` *runs at 30Hz and is provided by the environment;* `output_line_clk` *is a clock running* *at 660 times faster than `FrameStart` and is reset to zero at every tick of `FrameStart`; `visible_line_clk` is a sub-sampling of `output_line_clk` and is only active when `output_line_clk` value is between 100 and 579 inclusive.*

The process `Vstage` takes one input value (`frame_after_HFL` array) per activation, using `VFL_coefs` and `VFL_offset` as per-activation parameters.

When triggered, process `Vstage` will activate process `VFL_stage` at every tick of clock `FrameStart`, and will activate `OUTPUT` for every tick of `visible_line_clk`. Each activation of `VFL_stage` consumes the current value of `frame_after_HFL`, produces a new value of `frame_after_VFL` and uses the current value of `VFL_coefs` and `VFL_off` as per-invocation parameters. Each activation of `OUTPUT` selects an appropriate line from the latest value of `frame_after_VFL`, and acts as a sink node (output to `VOID`). ■

SALLY programs form a concrete representation of MPPN, with the addition of complete information on process internals. This information is critical to the precise evaluation of MPPN parameters such as process and channel latencies, based on a machine description of the underlying SANDRA architecture. In this way, SALLY program analysis and transformation can leverage on all techniques developed for MPPN.

The actual computation of bandwidth, buffer, latency and resource usage properties is done by the HARRY verifier. The MPPN abstraction of a SALLY source code enables fast estimation of these properties, which is critical to the design-space exploration of application and the target system. HARRY output is also used to find the necessary sequential and timing constraints to be included in the SALLY program, and to identify over-constrained applications not amenable to parallelization. It gives also a quick go/no-go answer in case of distributed application onto an network of resources: the available bandwidth and latency of the communication channels are checked against the current application partitioning.

4. Scheduling and Code Generation

The scheduling phase benefits of the domain-specific semantics of SALLY: array and iterator structures are constrained such that memory dependences (i.e., causality constraints) can easily be captured at the level of each iteration using classical *array dependence analysis* techniques [8]. Dependences are described by systems of *affine constraints* enforcing sufficient conditions to make a schedule valid. In addition, SALLY processes explicitly communicate through FIFO channels following the semantics of MPPNs. Extending array dependence analysis to communicating processes requires to match every send with its corresponding receive, i.e., to count the number of sends and receives; this may lead to polynomial expressions when communications are nested within multiple loops. To get back to a classical array dependence analysis problem, we convert each send/receive statement into a store/load reference into a cyclic buffer. This corresponds to a candidate implementation for the channel, assuming that the buffer is bounded and that the bounds are known at compile-time, which is easily checked on the MPPN model.

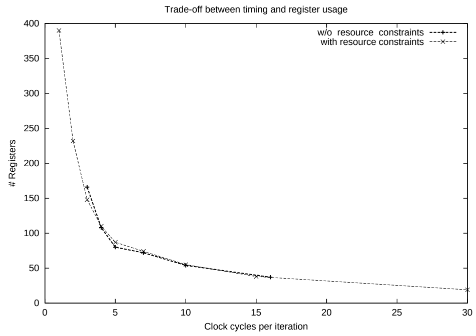
The resulting affine constraints can be handled by Feautrier's scheduling algorithm for "static-control" loop nests [9] (a class that includes most streaming algorithms), proven optimal in terms of asymptotic parallelism extraction [17]. This method uses an efficient constraint solver based on Parametric Integer Programming, PIP [7]. In theory, the result should be a multidimensional affine schedule for the whole program, telling when each iteration, operation or communication should occur. In practice, PIP may not scale to large systems generated from real-world streaming applications (its complexity is exponential in the worst case). Instead, we can benefit of the hierarchical decomposition of the SALLY program to cut down the scheduling problem to tractable pieces. This approach has already been studied and implemented in the context of the Alpha language [6].

4.1 The YAKA scheduler

The YAKA scheduler is the first step in the construction of the target program. It is invoked as soon as a first sketch of the architecture (number and type of the operators, size of the memory) is available. This information may be given, e.g., by a preliminary analysis using MPPN or be generated by the designer. It has two input interfaces. First, a C-like programming language (sYAPI) with process, channel, port and send/receive extensions; its ease of use makes it suitable as a development tool. The second interface is an intermediate representation in XML format (a convenient way of representing syntax trees) and do not impose any semantics on the designer. There is a DTD for this representation, which is primarily intended as documentation. In a future version, it is intended that the SALLY parser will generate an instance of such XML representation.

Eventually, the present version generates C code through standard polyhedra scanning techniques [1, 14, 3]. But the hierarchical control structure of SANDRA will require more work to generate low-level code (code compaction, code partitioning for different controlers, explicit generation of communication patterns, memory management).

In its standard version, YAKA does not address operation latency, real-time constraints, allocation of computations to the SANDRA controllers and low-level operators, and memory/register allocation. Theoretically, these additional constraints and tasks do fit into the YAKA model thanks to linear encodings, see e.g. [16]. Latencies and real-time deadlines are captured through additional affine constraints, and YAKA automatically converts resource constraints into artificial dependences (based on a cyclic allocation of resources to competing operations).




```
node HFL_elem(param float coef[64][6],
              in    pixel iline[1920],
              out   pixel oline[720])
{  ...
   (* polyphase FIR filter *)
   oline[t] =
       SUM(i[0..5],
           tmpline[src_idx(t,720,1920)+i-3]
           * coef[HFL_Phase[t]][i])}
```

Figure 6. Software pipelining of one HFL step

Most of the practical work around YAKA has to do with a better integration in the SANDRA compilation chain, both at the input and output sides. On the theoretical side, the problem of taking into account resource constraints (e.g., a limited number of adder-multipliers) has only partial solutions. In the present version, this is mainly obtained by adjusting the size of circular buffer (since the degree of parallelism cannot be higher that the amount of writable memory). This is unsatisfactory, as it needs manual arbitration between phases of the application. Another point is that the present scheduler is not modular, and the subroutines in the source have to be inlined.

4.2 Software Pipelining and Hierarchy

We are working on two possible solutions to improve the scheduling quality. Both of them are based on PILO-LORA, an existing software pipelining tool developed at INRIA that performs loop instruction scheduling (PILO) as well as loop cyclic register allocation (LORA). Unlike usual modulo scheduling algorithms, PILO implements the non iterative-DESP [18] software pipelining algorithm: it handles fine-grain resource constraints, including register types, non-uniform instruction formats and arbitrary reservation tables. PILO provides heuristics for the control of register pressure. For instance PILO-LORA can software-pipeline an elementary step of HFL (Horizontal Filter) and give the Pareto curve drawn in figure 6 for trading-off register pressure against timing in a dimensioning phase of the lowest level of the SANDRA architecture. In this example, the analysis assumed a design with 4 multipliers, 4 adders, and 3 memory ports (2 for loads and 1 for store instructions), with latencies of 20 cycles for loads and stores, and 10 cycles for multipliers and adders.

We are also considering other approaches for exploiting PILO-LORA. The first approach consists in regarding PILO-LORA as an alternative to YAKA's affine scheduling. Based on array dependences and reservation tables for every subtask involved in a process, the DESP algorithm can be applied to the innermost loops of the process. Then, application to the whole program requires a recursive application of PILO-LORA along the process hierarchy, much like hierarchical software pipelining techniques [12]. In practice, it requires an additional effort by the programmer since PILO is not able to automatically assign processes to SANDRA levels; hence a prior coarse-grain mapping has to be provided along with the SALLY source code.

Another approach — currently in progress — consists in combining the YAKA scheduler with a software pipeline "microscheduling" phase, integrating resource allocation and fine-grain rescheduling. E.g., YAKA is appropriate for detecting (possibly unlimited) parallelism in loop nests and PILO-LORA is much better at allocating resources and scheduling the innermost loops in the code generated by YAKA. Artificial scheduling constraints may be added to YAKA in order to make the innermost loop code scheduling more efficient.

4.3 Simulation of Sally programs using SystemC

In parallel with the development of the compilation chain for native target programs, we developed a tool generating SYSTEMC simulation models directly from SALLY programs. The generation of SYSTEMC models leverages on the direct correspondence between core elements of SystemC and the SALLY process model, including time-related features (clocks, delays) and concurrency. Latency requirements information from the application source code is used to define run-time consistency checks related to deadline respect etc.

The SystemC models are generated by combining two sources of information: a SALLY program (used as a specification of application requirements, both functional and temporal), and a target system description containing the functional and temporal capabilities of elementary processes used in that SALLY program.

The granularity of the generated SYSTEMC model directly corresponds to the granularity of the processes described in the SALLY program. When the SALLY program is refined to use actual operations of the target system, the resulting SYSTEMC model will be equivalent to a compiled instruction set simulator of the target for the application specified by the SALLY program.

This approach to simulation model generation provides several advantages:

- direct representation of time (clocks, latencies) and concurrency represented in SALLY programs;

- elimination of verbosity in SystemC programming, particularly in class declarations; the SALLY process interface declarations are on average ten times smaller than the corresponding SystemC declarations;

- automatic generation of behavior cross-checking and reporting/profiling code, e.g., detection of missed deadlines, monitoring of process activity; the latter complements the analysis capabilities of HARRY, providing a dynamic feedback on the utilization rate of processes, on process network latencies, and on traffic shapes of inter-process communications.

- automatic generation of activity traces using SystemC trace generation facilities; a side-by-side analysis of the evolution of process states simplifies the analysis of synchronization errors and greatly simplifies application debugging.

5. SANDRA System Architecture

To achieve the required degree of flexibility, all elements of the system should be configurable: functional units, their interconnection, control mechanisms, and memory subsystem. The frequency of reconfiguration of the different system elements depends on the nature of the tasks being performed, and can vary from several Hertz (mode changes, transitions between video frames) to several tens of MegaHertz (sub-pixel filtering). Centralizing the reconfiguration decisions would lead to a severe control bottleneck in the system. Instead, we propose to distribute the control and organize the system using a hierarchical approach, driven by and adapted to the characteristics of the target application domain.

The presence of explicit timing/frequency requirements in targeted applications led to another fundamental decision: instead of executing the tasks as fast as possible (driven by the intrinsic speed of hardware modules), the tasks are triggered right-on-time, synchronized with specific events. This mechanism addresses a major shortcoming of conventional interrupt-triggered architectures, which maximize average performance, and tolerate latency on "low-probability" events, expected to arrive fully asynchronously with the operation of the processor.

In our approach, the coordination of tasks operating at the same rate is performed by a single controller, which delegates the control of individual tasks to the next level of the hierarchy. This mechanism is again used to control sub-tasks inside each of the top-level tasks, and can be recursively repeated for as many levels as required. Conversely, tasks with independent clock domains may be executed by different controllers without unnecessary synchronizations. Finer-grain (thus, higher-frequency) tasks have stricter latency and bandwidth requirements: they require fast access to data and a high storage bandwidth. Conversely, coarse-grain tasks can tolerate longer latencies than the rest of the system. This fact is reflected in the memory and communication structure of SANDRA. The lowest levels of the system hierarchy use small, fast memories fully interconnected with relatively simple operators (FIR filters, etc.). Higher levels of the hierarchy offer a lower number of larger memories, and communicate through a higher-latency network. In this way, both the locality of data references and the natural synchronization of tasks at each level can be fully exploited within a unified system organization.

5.1 Functional Structures

The SANDRA hardware (see figure 3) consists of four distinct, superposed architecture layers corresponding to the different functions of the programmable system:

1 A hierarchical *control* layer managing resource activity and enforcing data dependences and real-time constraints on the three other functional structures of the SANDRA hardware.
2 A clustered *execution* layer gathering the functional units that operate on the contents of the data streams. A the lower level, each functional unit has a structure of a dedicated VLIW (Very Long Instruction Word), allowing the various operative units such as ALU, multipliers, etc to work simultaneously.
3 A heterogeneous *communication* layer tuned to the activity of each level: low latency, high bandwidth and high connectivity for the lower levels (pixel processing), higher latency and throughput achieved through larger data blocks for the higher levels (line, image processing). Because of the multiplicity of compute cores, the increased need for communication bandwidth, and the ever increasing wire cost inside systems-on-a-chip (SoC), the higher level internal communications will be implemented by networks instead of classical busses.
4 A *parameter* layer to customize the dedicated functional units of the execution layer, providing parameters for the operations on the main (pixel) computation flow. It is composed of very small RISC processors.

The execution and parameter layers are tightly coupled, but they perform radically different operations and process different kinds of data. A typical example of parameter unit is address generation: in most stream-processing algorithms, irregularities can be moved towards generating addresses, the remainder of the computation (e.g., pixel processing) following a regular flow. In the application domain considered, most of (pixel) compute kernels are similar in various algorithms; it is the way they are organized and how their parameters are computed that differ and gives the differenciating factor. Therefore, the parameter layer is the most flexible.

The architecture also distinguishes what is related to stream-processing computations from what is needed to run the SANDRA system: the execution code is split in two parts. The *application code* describes the computation on the data flow, and the *control code* schedules the application code over the hierarchical architecture. The application code is independent of the architecture instance and targets the communication, execution and parameter structures. The control code adapts the execution to a given SANDRA instance and to the dynamic part of applications.

5.2 Control Structure

The control structure of SANDRA is also composed of hierarchical layers. It illustrates the current trend in system design: systems are built by integrating components (often called IPs) (software or hardware) that are linked together by a common interface for communication and control. From the software point of view, this represents an evolution towards component-based software engineering. The hierarchical control system allows to distribute the control units near the functional units, hence to have a scalable and modular design. For example, if the higher-level controller implements a two-dimensional polyphase filter, it may decompose this task into separate horizontal and vertical filters, and delegate these subtasks to lower-level controllers. The top level does not need to know how the lower-level controllers perform the tasks, as long as they satisfy the specified time constraints. The lower level controllers can also decompose the mono-dimensional filters into blocks of vector operations, then into scalar products and additions, and so on. Each level of the computation is assigned to a controller, but several logical controllers can be folded into one physical controller. This scheme allows to have a ”logical” single control flow, but in fact not centralized and supporting some asynchrony (sub-level controllers can be independent, and they can feedback to the controller just above when they have finished their task).

The controller structure is the same for each level of the hierarchy, and is a stack-based virtual machine. Since multiple reentrant control codes should be executed on one physical controller, no explicit register allocation is done. Variables (used only for the control part of the application) are not explicitly allocated but remain on the stack. If a new task starts, it could start on top of the previous task stack as long as it eventually restores the right stack position. Using a stack-based virtual machine also eases portability across implementations of the SANDRA architecture and favors code compactness (factorization). As opposed to traditional stack based languages (Java, Forth, Postscript, OPL, etc.), we propose a *threaded* code structure where each instruction explicitly targets the next instruction to be executed; together with stacks, this improves factorization and eases reentrance and late binding. At each level (except for the lowest level), an instruction is composed of two main fields:

- the first one is dedicated to the control flow itself and its threading mechanism: instead of a “program counter”, the next instruction is indicated explicitly within the current instruction, in a similar manner as the linked-task structure of a real-time OS;

- the second field manages the lower level controllers; it is composed of several slots, one for each sub-controller; thus, there is no real distinction between a slot that triggers a (lower level) controller action (in this case, it is equivalent to a subroutine call) and a slot that controls a functional unit.

This structure allows to map a code onto various instances of SANDRA, with no recompilation and a minimum load during the instantiation of the code (binding). It gives some code expansion, but it is believed to be compensated by the code factorization present in our domain-specific applications.

5.3 Dynamic Reconfiguration and Application Switching

Let us now show how our static modeling, compilation and optimization framework can cope with the dynamic features of Ambient Intelligence media applications. Stream-oriented applications can be modified on request of the user (for example, having or moving a Picture In Picture), or due to the environment (new people entering a room, appearing in the vision field of a camera, etc.). However, these changes are slow compared to processing speed (user interactions are at the split second level, while video applications require changes at the frame rate, i.e., several milliseconds.)

The following paragraphs describe how dynamic reconfiguration and application switching can be mapped to the SANDRA system, while keeping the most important features of statically compiled code, such as guaranteed performances, predictability and high silicon efficiency.

In the considered application domain, there are always boundaries and limits that are given, either explicitly or implicitly: for example, giving a time limit for the execution time (e.g. a frame interval) and given the hardware resources, an implicit limit on the complexity of the application can be derived. We have developed the MPPN to help determining these constraints. If an application cannot fit within the hardware or time constraints, then it has to be modified (simplified). This led to the idea of "piecewise static control": an application is split into sub-applications, each of them specifies a sub-case for a certain range of parameters. Each sub-application can therefore be statically compiled with good efficiency. Dynamic behavior within a sub-application is handled by classical methods, such as worst case dimensioning and predicated instructions. Each sub-application has known characteristics, performances, and has a better efficiency because it covers only a sub-set of the variability of the original application. The activation of the relevant sub-application is done by one controller after the analysis of the input parameters.

Task or sub-task allocation in the system is done preferably in space rather than in time (using parallelism rather than a faster clock). Although the controller can implement time-sharing of functional units, the context saving needs to be explicitly expressed in the application, and it might be costly due to the large amount of data stored in the various pixel pipelines. This is why in SANDRA we prefer to use a ”space-sharing” mode: tasks are activated or deactivated at the level of the sub-controller directly in charge of the resources allocated to these tasks. In the SANDRA controller system, a task (or a subtask) is represented by the sub-tree with the root node being the controller that directly or indirectly (by the controllers that depend on it) ”covers” all the resources available for the task. Switching from one task to another one means simply deactivating one complete sub-branch of the tree and activating a new one. This is done in a very simple manner by changing the link field in the corresponding instruction of the root node.

The application domain enforces quality-of-service constraints including time requirements, and we are aiming at efficient and guaranteed usage of the architecture, not at the best effort processing (it is useless to go faster than required) Thus, the concept of hierarchical hardware and control provides a simple mechanism for tasks activation: if two sets of controllers, organized in a sub-tree, can control the same kind of compute elements, storage units and communication means, sub-tasks can execute indifferently on any controller. Hence, a simple compile-time scheduling and allocation is possible. The nearest controller that supervises the two tasks (i.e., the closest parent in control hierarchy) can adjust its own configuration to link the input/outputs of the new task to the rest of the system (this is made possible because all controller have an unique ID, therefore the code can know exactly where it runs). For any controller above the direct supervisor of the two tasks, the activation of one or the other task has no effect, as long as the communication schemes of the tasks are identical.

6. Conclusion

This work addresses the development of embedded systems dedicated to pervasive video applications in the Ambient Intelligence universe. The computation and bandwidth constraints of these applications exceed today's *general-purpose* processors by orders of magnitude, yet the cost of *application-specific* hardwired components becomes disproportionate with product lifetimes. To address these challenges, we stressed the need for a fast and efficient development process for *domain-specific* system solutions.

We surveyed the SANDRA approach to the architecture, compilation and language issues addressed by real-time streaming applications. The project led to promising results in four different aspects:

- The development of Multi-Periodic Process Networks — providing time and hierarchy to a restricted class of Kahn Process Networks — helps in design-space exploration, validation of resource/time properties, and in mapping onto distributed components.
- The design of SALLY, a domain-oriented language combining streams and implicit parallel constructs with non-functional properties such as time requirements and resource allocation.
- A compiler chain, using state-of-the-art algorithms for extracting parallelism, affine scheduling, software pipelining and code generation.
- A hierarchical architecture, easily tuned to the application requirements and allowing to run highly demanding algorithms at consumer price.

The proposed approach can also be applied when the resources are networked, mainly inside a SoC, and it allows to support dynamic behavior to some extend in an environment where hard time constraints are important, therefore demanding real-time streaming applications for Ambient Intelligence can be defined and efficiently mapped with our approach on embedded systems.

Further work is required before demonstrating a running prototype, and larger examples should be studied to explore the system's scalability. Nevertheless, we believe our model has matured enough to clearly state the most important directions towards a domain-specific approach to architecture and compilation development.

Acknowledgements

This project is supported by a Pierre et Marie Curie fellowship and a European Community project MEDEA+ A502 "MESA".

References

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proc. third SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 39–50. ACM Press, April 1991.

[2] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity*. Wiley, 1992.

[3] C. Bastoul. Generating loops for scanning polyhedra. Technical Report 23, PR*i*SM, University of Versailles, 2002.

[4] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *J. Comp. Simulation*, 4, 1992.

[5] A. Cohen, D. Genius, A. Kortebi, Z. Chamski, M. Duranton, and P. Feautrier. Multiperiodic process networks: prototyping and verifying stream-processing systems. In *Proceedings of Euro-Par 2002*, volume 2400 of LNCS, pages 299–308, Paderborn, Germany, August 2002.

[6] J. B. Crop and D. K. Wilde. Scheduling structured systems. In *EuroPar'99*, LNCS, pages 409–412, Toulouse, France, September 1999. Springer Verlag.

[7] P. Feautrier. Parametric integer programming. *RAIRO Recherche Oprationnelle*, 22:243–268, September 1988.

[8] P. Feautrier. Dataflow analysis of scalar and array references. *Intl. Journal of Parallel Programming*, 20(1):23–53, February 1991.

[9] P. Feautrier. Some efficient solutions to the affine scheduling problem. part II. multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, 1992.

[10] G. Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland Publishing Co., August 1974.

[11] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *Proc. 8th workshop CODES*, pages 13–17, NY, May 3–5 2000. ACM.

[12] M. S. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proc. ACM Conf. Programming Language Design and Implementation*, pages 318–328, 1988.

[13] H. Leverge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *J. of VLSI Signal Processing*, 3:173–182, 1991.

[14] F. Quiller, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Intl. J. of Parallel Programming*, 28(5):469–498, October 2000.

[15] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak. Fast prototyping of datapath intensive architectures. *IEEE Design and Test of Computers*, 8(2):40–51, 1991.

[16] L. Thiele. Resource constrained scheduling of uniform algorithms. *J. of VLSI Signal Processing*, 10:295–310, 1995.

[17] F. Vivien. On the optimality of Feautrier's scheduling algorithm. In *Proceedings of Euro-Par 2002*, volume 2400 of LNCS, pages 299–308, Paderborn, Germany, August 2002.

[18] J. Wang, C. Eisenbeis, M. Jourdan, and B. Su. DEcomposed Software Pipelining: a New Perspective and a New Approach. *Intl. J. on Parallel Processing*, 22(3):357–379, 1994. Special Issue on Compilers and Architectures for Instruction Level Parallel Processing.