HAL
open science

The SANDRA project: cooperative architecture/compiler technology for embedded real-time streaming applications

Zbigniew Chamski, Albert Cohen, Marc Duranton, Christine Eisenbeis, Paul Feautrier, Daniela Genius, Laurent Pasquier, Valérie Rivierre-Vier, François Thomasset, Qin Zhao

To cite this version:

Zbigniew Chamski, Albert Cohen, Marc Duranton, Christine Eisenbeis, Paul Feautrier, et al.. The SANDRA project: cooperative architecture/compiler technology for embedded real-time streaming applications. [Research Report] RR-4773, INRIA. 2003. inria-00071813

HAL Id: inria-00071813

https://inria.hal.science/inria-00071813v1

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The SANDRA *project: cooperative architecture/compiler technology for embedded real-time streaming applications*

Zbigniew Chamski — Albert Cohen — Marc Duranton — Christine Eisenbeis — Paul Feautrier — Daniela Genius — Laurent Pasquier — Valérie Rivierre-Vier — François Thomasset — Qin Zhao

N° 4773

Mars 2003

THÈME 1

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--4773--FR+ENG

ROCQUENCOURT

The SANDRA project: cooperative architecture/compiler technology for embedded real-time streaming applications

Zbigniew Chamski*, Albert Cohen†, Marc Duranton*, Christine Eisenbeis†, Paul Feautrier†, Daniela Genius*†, Laurent Pasquier*, Valérie Rivierre-Vier*, François Thomasset†, Qin Zhao*

Thème 1 —Réseaux et systèmes
Projet A3

Rapport de recherche n° 4773 —Mars 2003 —13 pages

Abstract: The convergence of digital television, Internet access, gaming, and digital media capture and playback stresses the importance of high-quality and high-performance video and graphics processing. The SANDRA project, a collaboration between Philips Research and INRIA, develops a consistent and efficient system design approach for regular, real-time constrained stream processing. The project aims at providing a system template with its associated compiler chain and application development framework, enabling an early validation of both the functional and the non-functional requirements of the application at every system design stage.

Key-words: Embedded system, stream processing, real time, programmable coprocessor, hierarchical parallelism and control, domain-specific language, domain-specific optimization, static scheduling, static resource allocation

* Philips Research
† A3 group, INRIA Rocquencourt

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 —Télécopie : +33 1 39 63 53 30

Le projet SANDRA : une coopération architecture/compilation pour les applications enfouies de traitement de flux en temps-réel

Résumé : Dans un contexte de convergence entre télévision numérique, accès à Internet, jeux vidéo, acquisition et diffusion de média, les traitements vidéo et graphiques de haute qualité à hautes performances sont propulsés sur le devant de la scène. Le projet SANDRA, une collaboration entre Philips Research et l'INRIA, développe une approche cohérente et pratique de la conception de systèmes temps-réel de traitement de flux réguliers et contraints. Le projet consiste à proposer un modèle du système complet, accompagné de sa chaîne de compilation et de son environnement de développement, autorisant la validation précoce des spécifications fonctionnelles et non fonctionnelles des applications, et ce à chaque étape de la conception.

Mots-clés : Système embarqué, traitement de flux, temps réel, coprocesseur programmable, parallélisme et contrôle hiérarchique, langage spécifique, optimisation spécifique, ordonnancement statique, allocation de ressources statique

1 Introduction

The market of embedded processing for digital media is growing at a regular pace, stressing the importance of a fast and efficient development process for new products and system solutions. At the same time, growing customer expectations and new media standards (e.g., in the area of digital video) set new performance challenges to system designers. Among the most demanding application areas are:

- video rendering and composition, including picture quality improvement;
- on-the-fly front-end video processing (feature retrieval, segmentation, depth reconstruction);
- back-end of a 3D graphics rendering pipeline, from the rasterization stage onwards (gaming, MPEG-4 graphics).

Most of these applications can be modeled as transformations of data streams with tight real-time constraints; these constraints require performance levels exceeding today's general-purpose processors capabilities by orders of magnitude. Therefore, such applications are usually mapped onto dedicated, highly concurrent, hardwired blocks. However, the cost of a new custom design in upcoming silicon technologies becomes disproportionately high, calling for more programmable, yet still efficient solutions which can be reused throughout an entire application domain. Such programmable solutions have additional benefits, such as fast adaptation to new standards, support for new "killer applications" within product lifetimes, and easy field upgrades having a coherent hardware family suitable for various kinds of devices (mobile, PDA, HDTV).

With the advances in IC technology, large scale system-on-a-chip architectures have become possible; the design bottleneck shifts now towards concurrency issues: e.g., how to assemble computing elements, store and transfer data, control the complete system efficiently, express applications for such systems, map the applications to a specific target system, etc. Co-design is often advocated as the best way to build *application-specific* systems. However, since our approach is *domain-specific*, cooperation between compilation and architecture moves back to a higher, more generic level: designing an architecture template (with multiple instances) and a compilation tool-chain tailored to this template. This work describes such a cooperative compilation/architecture approach called SANDRA: the *Stream Architecture eNgine Dedicated to Real-time Applications*. Our contributions address the following issues:

1. managing the system complexity;
2. managing explicit timing requirements;
3. mapping and validating applications.

The paper is further organized as follows: Section 2 presents an overview of our approach; Section 3 describes the organization of the SANDRA hardware template, followed by application representations in Section 5, and the compilation chain in Section 4. We conclude by a discussion of related work and an overview of the current status of the project.

2 SANDRA: a Global Approach

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 the context of:

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

2.1 Managing the Complexity of the System

To achieve the required system flexibility, all elements must 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 tens of Hertz (e.g., between video frames)

to several tens of MegaHertz (position-dependent filter coefficients at pixel level). 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.

In our scheme, the coordination of tasks operating in the same subsystem 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 synchronisations.

Finer-grain (thus, higher-frequency) tasks have a strict latency and bandwidth requirements: they require fast access to data and a high storage bandwidth. Conversely, coarse-grain tasks can tolerate long latencies and a lower bandwidth to 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, lower-throughput 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.

In practice, video processing tasks have between three and four levels of nesting (e.g., pixels, tiles, stripes and frames), leading to a typical three- or four-level hardware control hierarchy.

2.2 Managing Explicit Timing Requirements

The presence of explicit 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 overcomes 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 media streaming applications, most events can in fact be predicted and anticipated: the arrival of the next video frame, the next horizontal synchronization pulse, etc. Whenever the execution latency of individual tasks can also be predicted, asynchronous control (interrupt-triggered) can be 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 therefore, directly increasing its efficiency. To exploit the predictability of both the application domain and the SANDRA hardware, we replace the traditional “best effort” compilation strategy by a two-step static scheduling:

1. check that there exists a mapping of the application to the SANDRA system that satisfies every timing constraint;
2. optimize resource usage within the limits enforced by the timing constraints, and generate a static schedule and mapping accordingly.

We actually chose to move away from sequential application descriptions towards *timed process networks*, more suitable to the application domain. Process networks directly capture concurrency, and temporal annotations attached to processes (or groups thereof) provide a natural way of representing the timing requirements of the application.

2.3 Mapping and Validating Applications

Both the mapping of an application to the SANDRA architecture and the validation of the resource constraints for this applications require a system model. In SANDRA, the target system description, a.k.a. *machine description file*, follows a hierarchical approach as well: it provides quantitative target system models at multiple levels of refinement and precision.

The machine description file is used for design space exploration, to select a hardware solution within the SANDRA family, selecting the custom functional units for a specific application, and tuning the application parameters (such as, the filtering phases, the texture-mapping algorithm, etc.).

Optimization, scheduling and mapping algorithms of the compilation chain are based on the same model. For example, bandwidth and connectivity parameters guide the folding of concurrent processes to controllers and the allocation of data streams on memory (or communication) structures.

3 Hardware Structure

We propose a programmable architecture matching the requirements of regular media stream processing. Given the power limitations of sequential (or rather, sequentially controlled) processors, we concentrate on exploiting the high concurrency available in the applications through the use of hierarchically layered, communicating, concurrent execution units.

3.1 Hierarchical Execution

The SANDRA architecture is organized in a hierarchical manner that reflects the application domain, see Figure 1. Just as information hiding in object-oriented languages, only the relevant information is made visible for a given level of the SANDRA hierarchy. From the programmer's point of view, this is understood as computations on different levels of data structures, making the temporal and spatial locality explicit to the architecture. Data structures and coarse-grain operations occurring in the higher layers can be complex, dynamic, and possibly irregular; whereas basic data elements, regular data streams and statically-scheduled operations are found at the lower levels.

The lowest layer deals with the effective computations. Its associated storage units have a very low access time and high communication bandwidth, with a limited storage capacity. They may be seen as register banks —with customizable register indexing modes —in a classical microprocessor architecture.

The second layer has a larger storage capacity, but data transfers consist of structured aggregates (sets of elementary data). This allows to decrease the penalty for a higher latency, just like the use of cache lines in the first level cache of a mainstream processor.

The third and last layer handles large, structured data; its latency is bound to the DRAM access time and inter-cluster communication delays. Ideally, this layer should be able to store all the data required for the internal SANDRA computations, requesting data to the main system memory only for input/output data.

3.2 Functional Structures

Spread across the hierarchical layers, the SANDRA hardware consists of four distinct structures dedicated to the different functions of the programmable system.

1. A *control* structure 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* structure gathering the functional units that operate on the contents of the data streams.
3. A heterogeneous *communication* structure tuned to the activity of each level: low latency, high bandwidth and connectivity for the lower levels, higher latency and throughput achieved through larger data blocks for the higher levels.
4. A *parameter* structure to customize the dedicated functional units of the execution structure, providing seldom modified values that are not directly linked to the results of the main computation flow.

The execution and parameter structures are tightly coupled but handle separate data with disjoint types and operations; they are thus assigned to different computation units and communication schemes. A typical example of parameter unit is dedicated to address generation: in most stream-processing algorithms, irregularities can be moved towards generating addresses, the remainder of the computation (e.g., pixel processing) follows a regular flow. Another example is a unit providing filter coefficients that do not need to be modified at every pixel.

This model also distinguishes what is related to stream-processing computations from what is needed to run the SANDRA system. Therefore, the SANDRA programming model splits codes in two parts, compiled from the application source and support libraries: 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; whereas the SANDRA control code adapts the execution to a given instance.

3.3 Control structure

The control structure of SANDRA has three hierarchical layers. It illustrates the current trend in system design: systems are composed of components (software or hardware) that are linked together by a common interface for communi-

cation and control. From the software point of view, this represents an evolution towards component based software engineering. The hierarchical control system also allows to distribute the control units near the functional units, hence to have a scalable 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 some time constraints. Therefore, the role of the top level consists in decomposing its tasks into subtasks, providing the time constraints of each subtask, managing memory allocation, and transferring the inputs/results to/from the subtasks. All other information —how to perform the subtask —are handled locally by the lower level controllers. In the abovementioned example, the lower level controllers can also decompose the mono-dimensional filters into more elementary vector operations, that can also be decomposed 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 (see Section 6).

The controller structure is the same for each level of the hierarchy, and every controller satisfies the following conditions:

- it is independent of the hierarchical level;
- it can be slave of a higher level controller, or master of a lower level controller;
- the instruction set is identical for each level of the hierarchy;
- the code is reentrant and extracted at loading time from the executable binary.

To cope with these requirements, the controller model 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 code section that triggers a (lower level) controller action (in this case, it is equivalent to a subroutine call) and a code section 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 applications.

The SANDRA controllers are organized in a tree-like structure, with the master at the top and several low level controllers at the bottom. The control program for each level is loaded at boot time into the local program memory of each controller, however, a mechanism allowing to reload code at runtime is possible. Each bottom controller is in charge of a *functional unit* and directly manages its effective computation units. When triggered by an instruction issued by its master controller, a bottom controller executes a (programmable) microcode memory storing activation bits for the computation units, memories and communication links.

4 Compilation Chain

First of all, we would like to stress that we neither propose an application-specific optimization framework nor an architecture-specific code generator. We will show that every tool in the software flow is applicable within the whole domain of regular and real-time stream-processing applications, and can be retargeted to a wide range of embedded architectures.[1]

[1] But hierarchical scheduling for concurrent processes may be an overkill for flat VLIW cores (e.g., TriMedia).

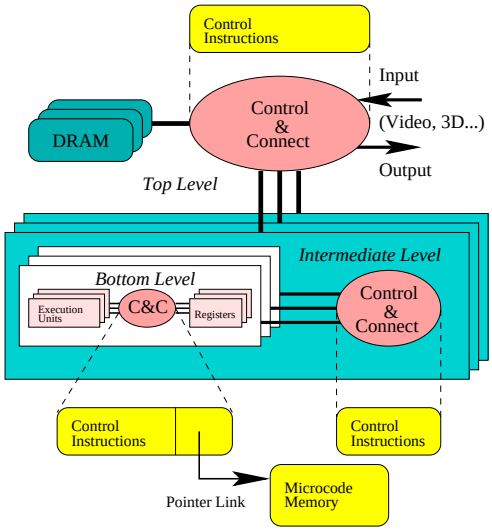

Figure 1: Hierarchical control and storage

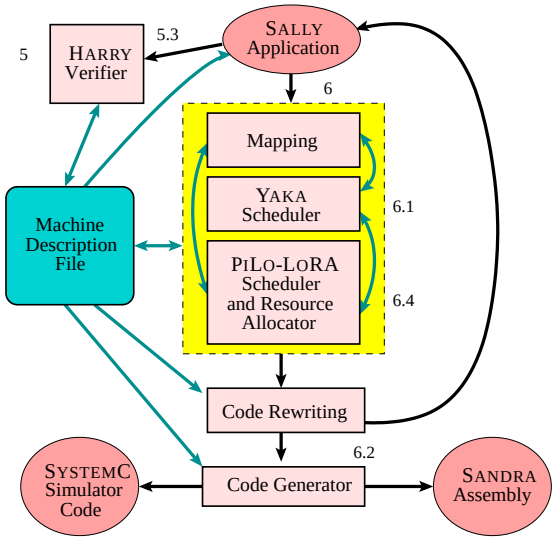

Figure 2: SANDRA compilation chain

The goal of the SANDRA compilation chain is to provide an automatic procedure for code generation on the complex SANDRA execution model, and to streamline the optimization of the generated code. More precisely, its task is twofold:

- generate 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 hard to handle at the application level;
- optimize 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 [1]. Indeed, CDFG 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 [1] or PTOLEMY [2] (with alternative data-flow graph models) have been developed in this area.

Conversely, compiler techniques for resource-constrained scheduling have been developed that are not limited to the context of a single loop [3] or may even reconstruct the control and data structures completely through algebraic loop-nest transformations [4, 5]. These techniques can distinguish between each iteration of a loop and each value of a stream/array, enabling more aggressive transformations. On the other hand, some of the versatility of CDFG and other flow-graph approaches lost, like the ability to handle arbitrary control flow or the natural integration of timing and resource constraints. Despite the lower versatility and the higher complexity, we believe that only such powerful techniques can efficiently harness the resources of the SANDRA architecture.

The SANDRA compilation chain is sketched in Figure 2, where numbers link transformation phases and code representations to the relevant sections. Compilation starts with an application description specified in high-level language called SALLY (to be described in the next section) and checks real-time properties with the HARRY verifier. During the design space exploration, the YAKA multidimensional affine scheduler and the PILO-LORA software-pipelining tool produce one or several schedules and resource allocations of the concurrent program; the programmer may drive the exploration in providing a coarse-grain mapping of (some) processes to SANDRA controllers. The code rewriting phase regenerates SALLY code from the abstract schedule, allowing for iterative refinement of the schedule. Finally, cycle-accurate simulation code and SANDRA assembler 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. Of course, this semantics-based description 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, 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.

5 Representation of Applications

The application model must capture the concurrency and real-time attributes of the applications and 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 application's information on 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.

5.1 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) [6]; a detailed presentation and discussion of the model can be found in [6]. The MPPN model is inspired by Kahn process networks [7] and Petri nets variants such as event graphs [8]. It also shares some motivations with the COMPAAN project [9] within the PTOLEMY environment [2]. Moreover, it provides four distinctive concepts, namely (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. Figure 3 sketches a 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:[2] the filtering process is decomposed into an horizontal phase (sub-process P_5) and a vertical phase (sub-process P_6).

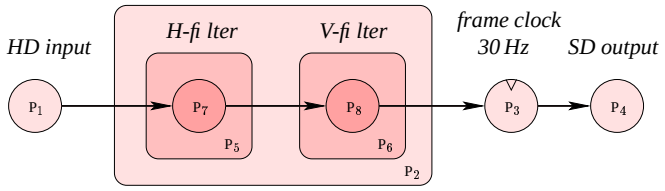

Figure 3: MPPN representation for a downscaler

5.2 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, it was decided to rely on MPPN to provide a clean set of concepts tailored to the application domain and familiar to domain specialists.

Using a formal representation of application requirement specifications as a direct input to the software chain brings three key benefits:

- direct availability of all application domain-related information to the software tools, enabling more powerful analyses and transformations;
- natural and implicit representation of parallelism, not bothering programmers with explicitly parallel constructs, and giving the compiler full freedom to exploit the available parallelism;[3]

[2] A similar filter is used for picture-in-picture and texture-mapping applications.
[3] This is a major difference with [9] whose input is a sequential MatLab program.

- simplification of the software production process, avoiding a manual translation of the specification into a general-purpose programming language.

SALLY is a declarative language relying on 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 [10]. SALLY variables correspond directly to channels (i.e., FIFO buffers) in MPPN.

Iterators are a uniform concept for expressing loops (either parallel or sequential) and event-based processing. Three types of iterators are available in SALLY: indices, counters, and clocks. *Indices* correspond to unordered, potentially concurrent iterations. *Counters* correspond to ordered, i.e., serialized iterations. *Clocks* are counters which change their values at specific moments in time, and are used to capture the real-time requirements of the application. All iterators are integer-valued, and their values can be used in statements.

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. *Processes* are sets of equations or networks of other processes that are evaluated for each value of a shared iterator, called the *trigger*. 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 off control parallelism against data parallelism.

Figure 4 shows an excerpt from a SALLY implementation of a two-dimensional polyphase filter. 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, `Vstage` activates process `VFL_stage` at every tick of `FrameStart` and activates process `OUTPUT` at every tick of `visible_line_clk`. Activation of `VFL_stage` consumes the current value of `frame_after_HFL` as input, produces a new value of `frame_after_VFL` as output, and uses the current value of `VFL_coefs` and `VFL_offsets` as per-invocation parameters. Each activation of `OUTPUT` selects a line from the latest value of `frame_after_VFL`, and acts as a sink node (output to `VOID`).

```
extern clock    frameStart          30Hz
clock           output_line_clk     660 @ FrameStart (* 660 lines including Vsync *)
clock           visible_line_clk    output_line_clk[100 .. 579] (* blanking = 100 *)

node Vstage(param float VFL_coefs[64][6], param int VFL_offsets[64][6],
            input pixel frame_after_HFL[1080][720])
{
  decls
    pixel frame_after_VFL[480][720]     (* after V filtering: 480x720 pixels *)
  code
    frame_after_HFL -> VFL_stage(VFL_coefs, VFL_offsets) -> frame_after_VFL
      every frameStart                  (* frame-level vertical filter invocation *)
    frame_after_VFL[visible_line_clk - 100] -> OUTPUT -> VOID
      every visible_line_clk            (* line-level control of video output *)
}
```

Figure 4: SALLY application example: filtering and output

5.3 Link With the Process Network Model

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.

Then, practical computation of the 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.

6 Scheduling and Code Generation

We now describe our approach to achieve the *static* scheduling and mapping of a SALLY program and to generate SANDRA assembler. Most of the following techniques rely on classical algorithms and research tools, but their effective integration in a compilation chain is not fully understood (see [11] and [9] for alternative approaches). The rest of this section summarizes actual implementations and work in progress. It should also raise interest towards the development of more ambitious compilation chains for domain specific applications and architectures.

6.1 Affine Scheduling for Kahn Process Networks

The scheduling approach benefits of the domain-specific semantics of SALLY: the array and iterator structures are constrained such that memory dependences (i.e., causality constraints) can easily be captured at the level of each iteration, through the use of classical *array dependence analysis* techniques [12]. 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 Kahn Process Networks. 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 [4] (a class that includes most streaming algorithms), proven optimal in terms of asymptotic parallelism extraction [13]. This method uses an efficient constraint solver based on Parametric Integer Programming, PIP [14]. 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 [11].

6.2 Implementation and Code Generation

These techniques are implemented in the YAKA (Yet Another KAhn compiler) scheduling tool. Practically, YAKA works with an XML description of the SALLY program, whose main elements are loops, statements and channels. The attributes of a statement are its surrounding loops (given from outside inwards), the variables which are read and modified, and its duration. If the statement is a send or receive, the name of the channel must be given. The attributes of a channel are its size and transmission delay.

Systems of affine constraints are automatically extracted from the source program and the machine description file. Moreover, YAKA sticks to the suggested schedule and mapping wherever enforced in the SALLY program (this is useful when iteratively refining the schedule). However, the output of the scheduler consists of systems of linear constraints assigning a logical (vs. physical) execution date to every iteration of a statement. To produce annotated

SALLY code (with mapping, resource allocation and clocking information), YAKA implements a sophisticated code rewriting phase: the CLOOG [15] tool —an implementation of Quilleré's algorithm [16] —regenerates loops and conditional structures from the affine schedule.

Assembly code generation requires an additional step. Beyond the usual loop, array and expression "lowering", this step requires an architecture-specific phase, due to the decoupled control-application paradigm of the SANDRA programming model. Control code must be extracted from loops, conditionals and parameters, and gathered into separate files for each (virtual) controller; whereas basic blocks associated with each controller are assembled and encoded into the SANDRA instruction format to form the application code. The ongoing work targets code compaction and instruction reuse within the application code.

6.3 Adding Resource and Real-Time Constraints

The previous section did not address operation latency, real-time constraints, allocation of computations to the SANDRA controllers and low-level operators, and memory/register allocation. Many of these additional constraints do fit into the YAKA model thanks to linear encodings, see e.g. [5], and sometimes resorting to conservative simplifications. Latencies and real-time dead-lines 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).

Today, YAKA does not produce a fully satisfying code and resource allocation, especially at the lowest level of the SANDRA architecture. This optimization weakness is currently the price to pay for the genericity of the approach, due to the non-linearity of most resource constraints and cost functions. The next paragraph points out some directions to progress in this area.

6.4 Software Pipelining and Hierarchy

We conclude this section by 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 [17] software pipelining algorithm: it handles fine-grain resource constraints, including register types, non-uniform instruction formats and arbitrary reservation tables.

The first approach consists in regarding PILO-LORA as an alternative to the previous affine scheduling technique. From 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 [3]. 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 for 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.

7 Conclusion

This work addresses the development of embedded systems dedicated to real-time streaming applications. The computation and bandwidth constraints of these applications exceed today's *general-purpose* processors by order of magnitude, and conversely, 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. Although the project is still under development, it already 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 design-space exploration, validation of resource/time properties, and 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.

We are unable to provide experimental results at this stage: further work is required before demonstrating a running prototype. Larger examples should then 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.

Acknowledgments

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

References

[1] 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.

[2] 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.

[3] 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.

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

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

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

[7] 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.

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

[9] 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.

[10] 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.

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

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

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

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

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

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

[17] 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.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifi que
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399