An Esterel Extension Toolkit for Developing Software Radio Applications

Hahnsang Kim and Thierry Turletti
INRIA -Sophia Antipolis
2004, Route des Lucioles BP 93
06902 Sophia Antipolis Cedex France
{Hahnsang.Kim, Thierry.Turletti}@sophia.inria.fr

ABSTRACT

Designing DSP (Digital Signal Processing) applications is a complex task. It requires a software development environment capable of putting together DSP modules and providing facilities to debug, verify and validate the code. PSPECTRA was one of the first toolkits available to design basic software radio applications on standard PC workstations. In this report, we present the ESTEREL extensions we have added to this toolkit in order to make the debugging and verification phases of development easier. These phases are known to be the most time consuming operations. In order to improve the performance of these applications, we have also implemented a new scheduling model which is well suited to the specification of the control part of the applications in ESTEREL. The performance results reported in the report are promising.

Keywords

Digital Signal Processing, Software Radio, Data-Pull Model, Data-Reactive Model, ESTEREL

1. INTRODUCTION

Software radios are wireless communication devices in which some or all of the physical layer functions are implemented in software. The software approach brings many advantages: the price and performance of the applications track the technology curve, the utility of existing hardware is maximised, previously designed functions can be re-used to design other functions because the hardware is the same. The greatest advantage is the tremendous flexibility of software applications. It allows us to implement applications in which any aspect of the signal processing can be dynamically changed to adapt to varying channel conditions, traffic constraints, user requirements and infrastructure limitations. However, the design of such DSP applications is very complex and requires multi-disciplinary knowledge: software architecture, signal processing (modulation, channel coding, etc.), real-time scheduling, networking protocols (error control, congestion control, etc.), verification, validation, etc.

The aim of our work is to make the implementation of such applications easier by accelerating the development time, especially the debugging and verification phases. In order not to reinvent the wheel, we have used the PSPECTRA [1, 2] development environment designed by the SpectrumWare project[1]. This toolkit provides a real-time signal-processing environment used to implement portable DSP applications on general-purpose workstations. PSPECTRA is especially targeted to develop software radios and provides an API for developers to make DSP applications on general-purpose workstations.

In this report, we present the ESTEREL extensions we have added to PSPECTRA. Implementing the control part of such applications with a formal language such as ESTEREL is likely to reduce the most time consuming development phases, the testing and the verification for correctness. We give the performance obtained with the new ESTEREL-based PSPECTRA environment, and compare it with the original version of the toolkit. In addition, we present a new scheduling model of the control part called the DRM (*Data-Reactive Model*) and compare its performance with the DPM (*Data-Pull Model*) [1] originally implemented in PSPECTRA.

The formal language ESTEREL [3, 4] is an imperative synchronous parallel programming language dedicated to reactive systems. It is well suited to the programming of hardware or software synchronous controllers for which the control-handling aspects are predominant. ESTEREL programs are input-driven: they wait for inputs and compute corresponding outputs in a cyclic manner. An input-output computation is called a reaction. Conceptually, synchrony means that reactions take no time, *i.e.*, output signals are available as soon as input signals are available.

Using ESTEREL, the following advantages are expected. Firstly, it will be easier not only to write control-handling functions, but also to minimise mistakes that may happen during the specification phase. The limited set of expressions provided by ESTEREL is enough to write specified properties and it reduces errors that may be generated using general-purpose programming languages such as C or C++. ESTEREL supports the strictness for control-handling functions as well as the flexibility for data-handling functions which makes it compatible with the C programming language. Secondly, it is possible to use optimisation, simulation and verification techniques commonly used in areas such as functional process or hardware design, and to extend them to software applications. XES [?] is a graphical simulator for ESTEREL programs which supports source code symbolic debugging. XEVE [5] is a graphical interface environment for the analysis and verification of ESTEREL programs modeled as Finite State Machines; it allows one to check output status, to verify fairness, and to detect dead-

---

[1]See URL: http://www.sds.lcs.mit.edu/SpectrumWare/

locks or livelocks.

The structure of this report is as follows. Section 2 describes the original Pspectra toolkit. In particular, we provide an overview of the Pspectra architecture and the range of DSP applications that could be designed with it. Section 3 presents the time constraint features of such applications, and two different models of scheduling: the DPM, originally implemented within Pspectra, and a new model called the DRM. Section 4 describes the Esterel implementation of the two scheduling models we have designed within the control part of Pspectra. Section 5 presents a comparison of the performance of the the EDRM (*Esterel-based Data-Reactive Model*), the EDPM (*Esterel-based Data-Pull Model*) and the ODPM (*Original C++-based Data-Pull Model* of Pspectra) on two different machines. In addition, the number of lines of code is analysed for the three models. Finally, the last section concludes the report and introduces future work.

2. SOFTWARE ARCHITECTURE

Pspectra is a real-time signal-processing programming environment used to implement portable DSP applications such as software radios on general-purpose workstations. This environment includes a library of portable (across platforms) DSP computing functions and a performing I/O subsystem. With Pspectra the hardware part is minimal and the boundary between software and hardware is shifted right up to the A/D converter. This increases flexibility by bringing more functions under software control [1, 2].

Pspectra architecture is partitioned into a control part (*out of band components*) and a data part (*in-band components*). This partitioning allows for a maximal re-use of the computationally intensive DSP modules. The data part is the place where the temporally sensitive and computationally intensive work takes place. All code that is related to scheduling processing modules is contained in the control part.

2.1 Data Part

The data part contains the code required to perform specific signal processing tasks, access functions used by the control part to configure and monitor the DSP tasks, and I/O functions that read data from and write data into buffer. The data part consists of two components: processing modules and connectors. The processing modules perform the signal processing tasks and communicate with the control part by the access functions. Connectors correspond to the buffer which data is read from and written into. The processing modules are classified as follows:

- *Sources* are specialized modules that have one or more output ports and no input ports.

- *Sinks* are specialized modules that have one or more input ports and no output ports.

- Other intermediate modules have one or more input ports and one or more output ports.

Each port must be connected to exactly one connector. A connector can be thought of as a wire that carries signals from the output of one processing module to the input of another processing module. Each signal processing path has at least one source and at least one sink, and starts from at least one source and ends with at least one sink.

2.2 Control Part

The control part is responsible for creating topology and modifying current data flow according to needs in the system, controlling the communications between processing modules, handling user interaction, and monitoring the data computation on each processing module. The data manipulated by the processing modules flow from sources to sinks. A processing module (from which the *downstream* modules receive input sample data) reads input data from other processing modules (to which the *upstream* modules send output sample data) directly connected by connectors, and performs some computation on it.

To refer to the input and output data in the buffer, the *SampleRange* parameter is used on the processing modules. This parameter keeps track of the position of the data that each processing module accesses. As shown in Figure 1, a SampleRange contains two pieces of information: an *index* identifying a starting point from which to read data in the buffer and a *size* identifying the amount of data to read.
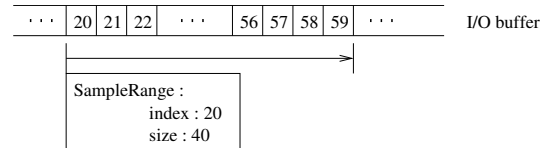

Figure 1: SampleRange: Each data block is referenced with an index and a size

All processing modules include an *estimating method* and a *computing method*. Estimating methods specify the SampleRange used by computing methods with reference to the SampleRange parameter of the upstream modules and inform the downstream modules of their SampleRange parameter. In addition, estimating methods have to ensure that the same data are not computed more than once. Computing methods start when estimating methods successfully return, and they manipulate the data that estimating methods have selected.

2.3 A New Architecture

Even though Pspectra provides features such as dynamic flexibility, portability, and re-usability by software implementations, it has some drawbacks compared to general software tools: it lacks the functionality of simulation or testing, and formal models accessible to developers. Data-intensive and control-intensive handling activities require different programming techniques, respectively.

In the Esterel-based approach, the data part (including core DSP computing functions and connectors) is described in C++, and the control part, in which the scheduling models are implemented, is described in Esterel. The control part contains DSP scheduling modules, as extra components, in order for the control part to access DSP computing functions of the processing modules in the data part. This allows one to simplify the data part since the access functions in the data part are now a part of the control part. Each DSP scheduling module in the control part is associated with a processing module in the data part.

Therefore, in the Esterel-based Pspectra, the data part contains signal processing-intensive functions and data buffering, and the control part contains all functions related to perform the scheduling model and functions that access to

the data part, including all tasks in the control part of the original PSPECTRA.

Figure 2 illustrates the ESTEREL-based PSPECTRA software environment. The control part is written in ESTEREL and the data part is written in C/C++. The component package is a package that provides for the data part a library where computational functions are described. The General Purpose PCI Interface (GuPPI) developed at MIT along with the Linux operating system allows the sampled signal data to be directly transferred in and out of memory of the workstation via Direct Memory Access (DMA).
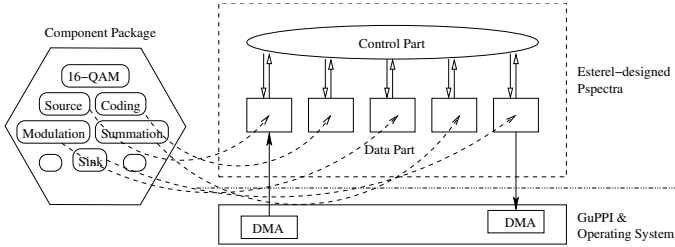

Figure 2: The ESTEREL**-based** PSPECTRA **environment**

3. DIFFERENT TECHNIQUES OF SCHEDULING

Before describing the statistical real-time model, it is useful to review existing definitions of real-time systems. Although there are many different definitions of real-time constraints in the literature, we can generally classify them into *hard* real-time and *soft* real-time constraints [6]. In hard real-time systems, the overall time consumption of all DSP modules is strictly limited. In other words, all the time critical functions have deadlines which must always be met in order for the system to function properly. Safety-critical real-time applications are used in domains including space rockets, aircraft automatic pilots, air traffic control, car vital systems, and some medical equipment. On the other hand, soft real-time systems are not well defined. They are generally thought of as real-time systems that can still function reasonably well even if deadlines are occasionally missed. Indeed, the reliability of a system relies on the accuracy of the estimates.

The PSPECTRA system as well as the ESTEREL-based PSPECTRA runs on general-purpose workstations without explicit real-time support in the operating system (Linux OS). By taking advantage of the ability to sometimes process data faster than in real-time, jitter in the computation time of some functions can be absorbed. This provides a mechanism for dealing with the frequent, small scale time variability. Resource unpredictability may result in the processing time occasionally exceeding the real-time rate, but the average processing rate can still be well below the real-time threshold. Thus, there is a trade-off between higher average throughput and jitter in the computation time. In order to deal with the larger variations, the concept of *statistical real-time performance* is introduced, in which an application is characterised by:

- the cumulative distribution of the number of cycles required to complete the task.

- a desired real-time bound.

- a specification of the action that must be performed when the deadline is not met.

This is a kind of soft real-time constraint, since deadlines can be missed without disastrous consequences. The probability that the task will be completed within the desired time bound can be expressed from the cumulative distribution of cycles required by a given application. This is possible since the statistics associated with the execution time are consistent. Note that if the task completes with a probability of one, then the system can provide hard real-time constraints. Memory buffers are used to support temporal decoupling by allowing information to flow between DSP modules operating at different rates.

Different actions are possible when a deadline is missed. For example, we can abort computation and drop the remaining data, replace the remaining data by a special value or partially estimated data from the result, or start processing the next slice of data while the current processing job continues in parallel.

Instead of extending the real-time paradigm across the whole system, PSPECTRA extends the boundaries of the virtual time environment by (i) time-stamping and temporally decoupling sampled information at the edge of the system and (ii) providing a virtual time programming environment in which it is possible to implement applications that process temporally sensitive information.

3.1 DPM: Data Pull Model

The control part of PSPECTRA is based on the DPM [1]. Within this model, the execution is driven by the sink which requests data as it is needed. the DPM is implemented according to a "lazy evaluation approach" [7]. Lazy evaluation (call by need) has been proposed as a method for executing functional programs. The advantages of this are, among others, that unbounded data structures, e.g. infinite lists, can be handled easily, and furthermore it makes interactive input/output possible in functional programs. In the DPM, callers correspond to sinks which select the DSP scheduling modules required for computation.
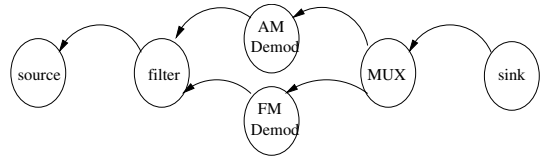

Figure 3: Diagram of DSP scheduling modules

In Figure 3, the sink calls the computing function of the MUX module and then the MUX module recursively calls the computing function of either the AM Demodulation module or the FM Demodulation module depending on the data-dependency [2] between the Demodulation and the MUX module. That is, if the data requested by the sink is associated with the AM Demodulation module, there exists data-dependency between the AM Demodulation and the MUX module. Otherwise, the data-dependency exists between the FM Demodulation and the MUX module. In this case, the MUX module is not able to have the data-dependency of both the AM Demodulation and the FM Demodulation module. Data-dependency relies on both the topology and the property of a given application. Therefore, the DPM dynamically adapts to changes in the processing topology.

In addition, the DPM effectively avoids computing the unnecessary intermediate data. For example, in Figure 3, the MUX module requests sample data from the AM Demodulation module and applies DSP functions to the returned sample data from the AM Demodulation module. After the computation of some sample data, when the MUX module requests the next sample data from the FM Demodulation module, the FM Demodulation module skips the corresponding sample data computed by the AM Demodulation module and computes only the sample data requested by the MUX module. This is hard to implement on the traditional DFM (*Data-Flow Model*) [8].

However, this model also has some drawbacks: with this approach, it is not possible to take advantage of the parallelism between the DSP scheduling modules. When the data-dependency between the DSP modules on the DPM has been decided, other modules not related to the data-dependency can do nothing. For example, suppose that there is an application with the following topology: it consists of two sources, two sinks, and several intermediate modules where there exists an intermediate module shared by two sinks and there is no relation between the two sinks. According to the DPM, one of the two sinks makes the data-dependency by requesting sample data from the intermediate module and performs computations. The other sink does nothing regardless of the lack of association with the current performed sink. In addition, when the sequential processing chain is created according to the data-dependency, sample data is processed by passing through this chain and the next sample data will be processed after the computation of the corresponding sample data is completed. Namely, it is not possible to interleave the computation chains of the current sample data and those of the next sample data.

3.2 DRM: Data-Reactive Model

In this section, we describe the DRM which is based on DFM. Within this model, the computation of sample data is triggered by the availability of data.
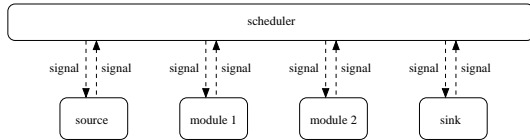

Figure 4: Data-Reactive Model

Figure 4 shows the architecture of the DRM specified in ESTEREL, which is especially well suited to reactive systems. We benefit from the well-formed semantic properties of ESTEREL such as parallel composition and hierarchical automata, which is introduced in [4].

The DRM acts on available data and performs an appropriate behavior corresponding to the property of the data, and then it waits for the next piece of data. The computed data flows from sources to sinks, processed from a DSP scheduling module to the following DSP scheduling modules, which are assumed to perform computations successively. All DSP scheduling modules react on available data. The relation among DSP scheduling modules is determined by a scheduler that decides which DSP scheduling module starts and stops its computation. The scheduling approach is as follows:

- data computation starts on the sources.

- whenever data on DSP scheduling modules are available, they start computing it.

- the corresponding data are completed on the sinks.

All DSP scheduling modules communicate with signals controlled by the scheduler. As soon as sources finish computing the data, they emit some signals that trigger computation of corresponding data on the downstream modules and then wait for ack (acknowledgement) signals from them. The DSP scheduling modules wait for two events: available data from the upstream modules and ack signals from the downstream modules that indicate completion of computation of the previous data. Having received both, the DSP scheduling modules compute the available data, and then transmit the computed data for the downstream modules, and emit ack signals to the upstream modules simultaneously. The corresponding data are finally consumed on the sinks. This scheduling mechanism is similar to pipeline methods, which allow one to interleave a data-computing iteration and a subsequent iteration, thus enhance the performance of data-computation.

4. IMPLEMENTING CONTROL PART OF PSPECTRA IN ESTEREL

We have implemented the control part in ESTEREL using two different models of scheduling: the DPM and the DFM. To illustrate the description, we use the *udp_tx* example of PSPECTRA applications included in the PSPECTRA toolkit. The udp_tx application has two functionalities: one is to modulate [2] input sample data and then transfer them through the network, and the other is to display the corresponding modulated sample data through a software oscilloscope.

4.1 Example of Pspectra application: udp_tx

Firstly, let us describe udp_tx, as shown in Figure 5. This application is composed of different modules: *source*, *coding*, *modulation*, *summation*, and two sinks (*udpsink* and *scopesink*). It gets input sample data from the source and performs the series of signal processing functions such as coding, modulation, and summation. Then, it displays the corresponding waveform on the scopesink and sends the output sample data onto the network through the udpsink. The source module continuously emits input sample data from a file through its output port until all sample data are processed. The coding module transforms "bits" from the upstream source module into "symbols", whereas the modulation module performs a modulation algorithm. The scopesink module displays a waveform on the screen. The summation module adds input sample data to history data. Then, the udpsink module sends the corresponding sample data as a UDP packet to the destination that will visualise the constellation diagram.

4.2 Implementation of the DPM in Esterel

Figure 6 shows that the architecture contains three different parts. The first part written in ESTEREL creates the DSP scheduling modules, initializes them and performs the

---

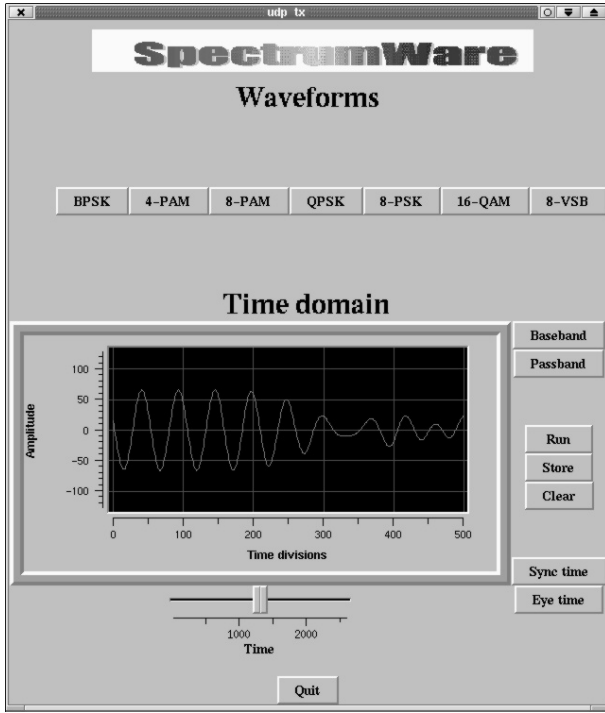[2]Possible modulations are: BPSK, 4-PAM, 8-PAM, QPSK, 8-PSK, 16-QAM

Figure 5: A screen shot of the udp_tx application

scheduling. The second part written in C/C++ is used to link ESTEREL-designed components to C++ objects. The last part written in C++ is the one in which DSP algorithms are performed.
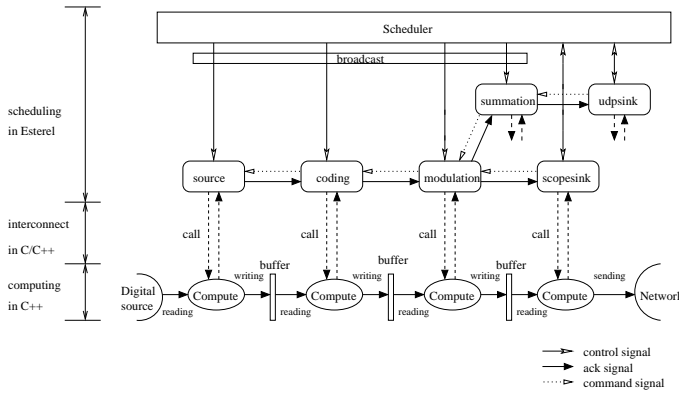

Figure 6: Scheduling of the control part with the DPM.

With the DPM, the scheduler uses two different phases: *data requesting phase* and *data computing phase*. Estimating methods are performed in the data requesting phase and DSP algorithms are performed in the data computing phase. The scheduler compares the index of SampleRange on the sinks to select the sink that has the lowest sample index. Then, in the data requesting phase, it estimates the SampleRange of the selected sink and requests the corresponding SampleRange from the upstream modules. When the requested SampleRange recursively reaches the source from the sink corresponding to the *command signal* flow,

the source, in the data computing phase, starts computing the requested SampleRange (actual data-computation starts by calling a process in C++). Then the computed data are transmitted to the downstream modules corresponding to the *ack signal* flow. At the following iteration the scheduler determines which DSP scheduling module will perform computations. For example, when the modulation module has enough for remaining sample data to be provided for the summation module, it does not request input data from the coding module. The upstream modules of the modulation module including itself are not activated by broadcasting *control signal*s. Therefore, computation will start from the summation module.

The data request is performed from the sinks to the source and the data computation is performed from the source to the sinks. Note that while a SampleRange is passed through from request to complete computation, all the corresponding modules are serialised. In other words, during the computation in the sinks, other modules either not related to the computation or already finished the computation are not able to manipulate the next SampleRange.

4.3 Implementation of the DRM in Esterel

In Figure 7, each module communicates with other modules by receiving *control signal*s from and emitting *ack signal*s to the scheduler. The scheduler first manipulates sample data on the source. The sample data computed by the source are then transmitted to the downstream modules. Intermediate modules (i.e. coding, modulation and summation modules) compute the sample data according to their own signal processing algorithm. Finally, the corresponding sample data are completed on the sink. When the sample data are being processed, synchronization between a module and adjacent modules is controlled by the scheduler. Therefore, sample data flows from the source into the sinks through intermediate modules.
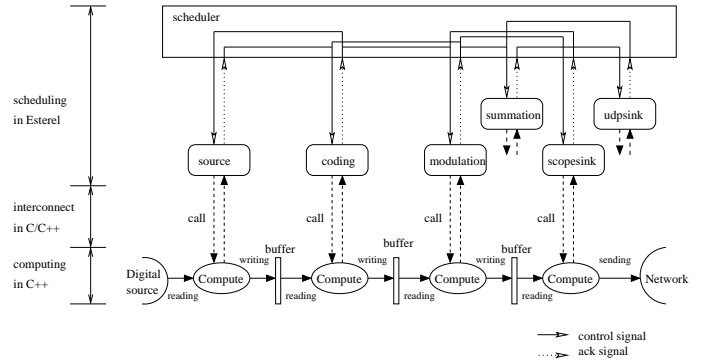

Figure 7: Scheduling of the control part with the DRM

The scheduler works as follows. Firstly, it triggers an estimating method on the source by sending a control signal. When the estimating method terminates, the source emits an ack signal to the scheduler in order to trigger the estimating method on the modulation module. Then, the source runs the computing method "instantaneously". After computation, the source needs to pause until the corresponding data on the coding module is ready to be computed, at which point it is expected that the coding module has just completed its computation and others are either active or

ready. The ack signal from the coding module triggers the next iteration on the source.

When each intermediate module gets a control signal from the upstream modules, it starts computation. It then transmits the computed sample data to the downstream modules and sends ack signals to the upstream modules simultaneously. The sinks perform the same operation as intermediate modules except that there is no ack signal sent to the downstream modules.

4.4 Comparison between the DPM and the DRM

In the DPM, the data requesting phase is separated from the data computing phase. After all the estimating methods on the modules are completed, the computing methods are triggered. On the other hand, in the DRM, the requesting phase is not separated from the computing phase. As soon as an estimating method on a module is finished, a computing method can start and triggers an estimating method on the following module.

5. PERFORMANCE ANALYSIS

Our performance analysis is carried out on both a PIII 600MHz machine with 256MBytes RAM and a SMP machine with two PII 390MHz and 256MBytes RAM, on Linux version 2.2.15, respectively. We provide a performance comparison of the EDRM (*Esterel-designed Data-Reactive Model*), the EDPM (*Esterel-designed Data-Pull Model*) and the ODPM (*Original Data-Pull Model of Pspectra*). Note that udp_tx uses the 16-QAM modulation algorithm in the experiment. Two udp_tx applications based on the EDRM and the EDPM, respectively, are compiled by the version 6.03 of the ESTEREL-TECHNOLOGIES ESTEREL compiler and are optimized by REMLATCH [9] and SIS [10]. The REMLATCH processor is called to optimise the state encoding of the circuit and SIS is used to reduce the combinational logic introduced by the sequential optimisation of REMLATCH.

An ESTEREL code is compiled into a Blif code by ESTEREL compiler with -blif flag. The Blif code is optimized by REMLATCH, and once more re-optimized by SIS. The optimized Blif code is translated into standard C code by ESTEREL compiler. The executable code [3] is built up by integrating the C++ code of the data part into the C code.

5.1 Performance results on a PIII/600MHz workstation

Figure 8 shows the number of output sample data processed per second by the udp_tx applications based on the EDRM, the EDPM and the ODPM on PIII 600MHz machine. The output sample data produced by the EDRM version is about two times more than those produced by the EDPM and the ODPM version. The figure shows that the EDRM version enhances the performance about 45% over the ODPM version by means of a high degree of pipeline methods. At t=30s, the EDRM version produces 2.5Msps (4-bit samples per second) (i.e. more than twice as many as the EDPM and the ODPM which produce 1.1Msps and 1.3Msps, respectively). Therefore, the performance of the EDRM version is better, compared to that of the EDPM and the ODPM version, by means of pipeline computation

---

[3]The executable code is obtained by gcc version egcs-2.91.66 with the -O2 optimisation flag.

---

technique. However, note that this experiment was performed on a PC which had no other applications running so as to avoid interferences.
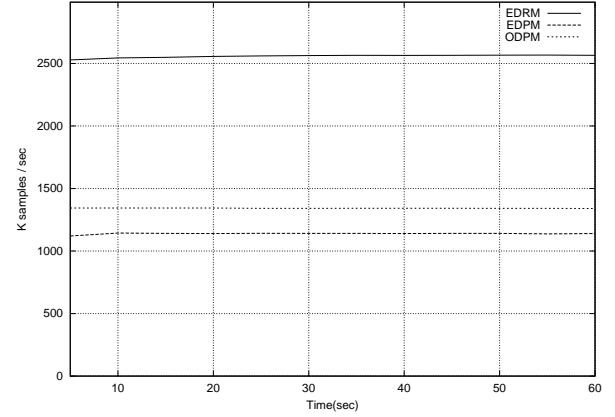

Figure 8: Number of processed samples per second without interfering process

Figure 9 shows the number of output sample data processed per second by the EDRM, the EDPM and the ODPM versions with interfering processes which correspond to both a *grep* command [4] and displaying functions. The grep process searches the entire file system running concurrently and the displaying process displays a waveform from the processed sample data. With the interfering grep process we ensure disk activity. The induced disk activity is expected to interfere with the ram disk access of the udp_tx application. We note that two cases which correspond to the experiments using the grep process and the displaying process, respectively, has been tested separately. In other words, the 3 curves that are above 500Ksps on the vertical axis correspond to experiment results from executing the udp_tx application in parallel with the grep process, and the others corresponds to ones in parallel with the the displaying process.
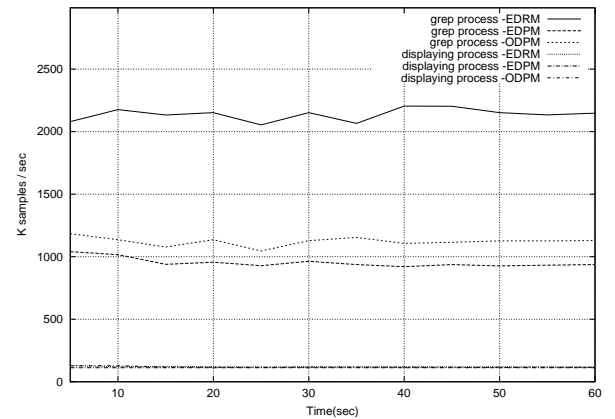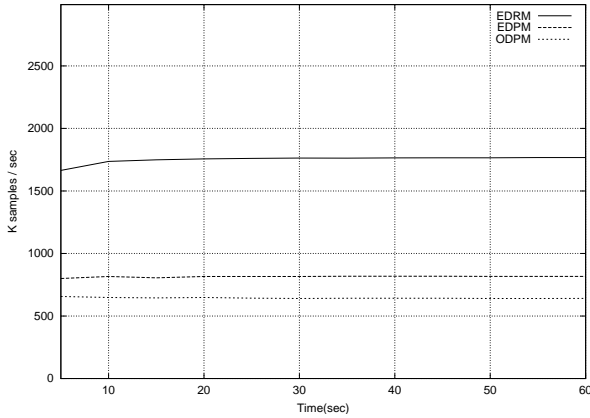

Figure 9: Number of processed samples per sec with interfering process

---

[4] A *grep* command that has a higher priority 10 than normal priority 8 was utilised for a stress scenario in the experiment settings of [11].

We see that the interfering grep process does not substantially influence all versions of udp_tx even though it adds some jitter. At t=30s, the EDRM version produces about 2.1Msps, whereas the EDPM and the ODPM version produce 960Ksps and 1.1Msps, respectively. However, the displaying process blocks the udp_tx application from running and substantially degrades the performance of the EDRM version. Therefore, we see that the displaying process (which also corresponds to any other functions related to X-Window process) is the primary bottleneck of running real-time softwares on a PC. At t=30s, the EDRM version generates 120Ksps, the EDPM and the ODPM version generate 112Ksps and 117Ksps, respectively. Compared to the results without the displaying function (as shown in Figure 8), the EDRM version loses up to 77% of its performance, whereas the EDPM and the ODPM version lose about 34% and 40% of their performance, respectively.

## 5.2 Performance result on bi-processor PII 392MHz SMP machine

Figure 10 shows the number of output sample data processed per second without interfering process on the SMP machine. At t=30s, the EDRM version produces about 1.7Msps. About 800Ksps and 600Ksps are produced by the EDPM and the ODPM version at this time, respectively. Taking into account the results from the PIII machine in Figure 8, even though the SMP machine has more computing power than the PIII machine, the number of produced sample data on the SMP machine unexpectedly decreases by up to 20%.

Figure 10: Number of processed samples per second without interfering process

On Figure 11, the 3 curves that are above 500Ksps on the vertical axis and the others, respectively, correspond to the number of output sample data produced in parallel with a grep process and in parallel with a displaying process on the SMP machine. Compared to the experiment data of Figure 10, we see that the grep process does not influence the udp_tx application. This results from dispatching two parallel processes, the udp_tx process and the grep process, to each CPU. In contrast, for the displaying process, the number of output sample data produced by the EDRM version dramatically decreases. The reason for this is that even though the computations of the displaying process are performed on the other CPU, the udp_tx process should wait

until the called function related to the displaying process is completed before starting the next task. Therefore, it is hard for the displaying process not to completely block the udp_tx process.
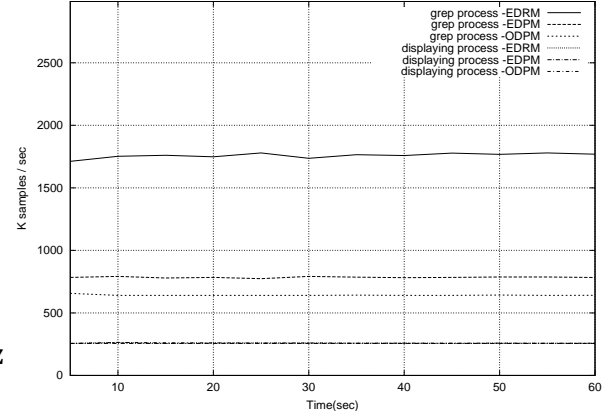
Figure 11: Number of processed samples per second with interfering process

Compared to the experiment results (which correspond to the curves that are below 500Ksps on the vertical axis)for the displaying process on a PIII machine, all versions on the SMP machine compute twice more sample data. This results from the diminishing influence of the displaying process on the udp_tx process due to the high degree of parallelism on the SMP machine. On the PIII machine, CPU cycles are spent in context switch due to assigning the CPU to either the displaying process or the udp_tx process.

5.3 Overhead of the displaying process

Figure 12 and 13 show the CPU load of the EDPM version on the PIII machine and on the SMP machine, respectively. The experiment is performed by monitoring the two main processes: the displaying process and the udp_tx process. We do not include in the experiment results other processes that has reached sleeping state or that utilise less than 1% CPU, most of which correspond to processes which are in a sleep state. The sum of the CPU load of such processes actually amounted to less than 1%.

In Figure 12, the CPU loads of the displaying process and the udp_tx process, between t=0s and t=14s, correspond to about 80% and 19%, respectively. When the displaying process, between t=14s and t=15s, reaches a sleeping state the CPU load of the udp_tx process increases: as the tasks that the displaying process performs are reduced, the CPU load of the displaying process decreases up to 42%, whereas the udp_tx process obtains a high CPU load of up to 57%. Therefore, we see that the displaying process is the main bottleneck in running the udp_tx process at least on a uni-processor machine.

In Figure 13, the displaying and the udp_tx process, by at t=13s, have almost 100% and 60% of the CPU load, respectively. The remaining 40% of the CPU load in the udp_tx process corresponds to the idle time that the udp_tx process waits for the return of a function call related to the displaying process. From t=14s, the main occupation of the CPU load is reversed.

From Figure 12 and 13, we note that the displaying pro-

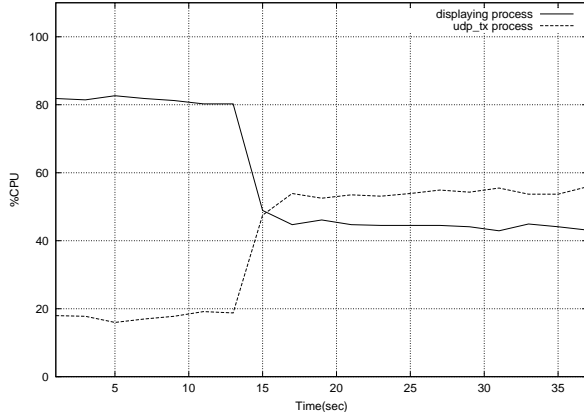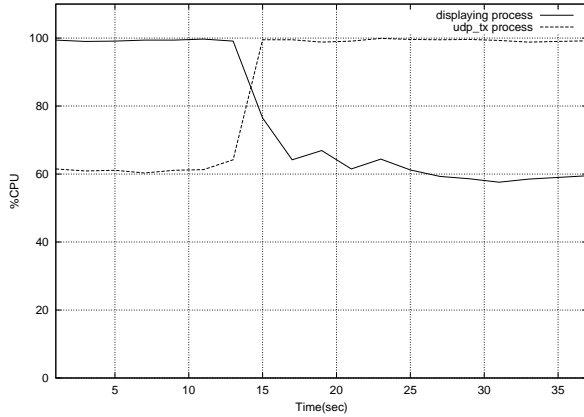**Figure 12: Evolution of the CPU load on the PIII machine**

Figure 13: Evolution of the CPU load on the SMP machine

cess hinders the udp_tx process from running on the uni-processor machine on the Linux OS, but the SMP machine to some extent allows us to overcome the overhead of the displaying process.

5.4 Comparison of LOC (Lines of Code)

Table 1 gives a comparison of the number of loc for the different versions of the control part of PSPECTRA. The EDRM and the EDPM version have 978 lines and 694 lines of ESTEREL code, respectively. ESTEREL code is translated into C code of which the EDRM and the EDPM version corresponds to 5253 lines and 6267 lines, respectively. Even though the EDPM version has about 250 less loc than the EDRM version, it obtains about 1000 more loc than the EDRM version as a non-optimized generated C code. REM-LATCH and SIS are applied to the optimisation of both ES-TEREL programs. We have obtained about 9.6% code optimisation of the EDRM version and about 4.0% code optimisation of the EDPM version.

Table 1: Comparison of loc

code line\model	EDRM	EDPM	ODPM
ESTEREL code	978	694	*not used*
generated C (non-opt)	5253	6267	*not used*
generated C (opt)	4749	6018	*not used*
data-handling C	699	678	*not used*
C++ code	1821	1864	4387
Sum Total (opt)	8247	9254	4387

Even though the loc of the EDRM and the EDPM version are about twice as long as that of the ODPM version, taking into account that the advantage of a general-purpose system is to utilise the large amount of memory, loc is not an important issue for these applications, as opposed to embedded applications. Instead, the cost of extra loc can be considered as the benefit of ESTEREL: the easy expression of preemption [12] and broadcast as well as synchrony, the simulation, and the verification, etc.

5.5 Pros & Cons

The software architecture of the ESTEREL-designed PSPEC-TRA, whose features include a clear separation between the control part and the data part, and a component-based DSP library, makes easier the design and implementation of DSP applications. On top of that, it integrates verification functionalities which provide the debugging and verification phases of development.

On the other hand, signal processing is usually dedicated to a computation-intensive and time-sensitive process which requires pre-empted resources like CPU. As we have seen in the 5.3 Subsection, all resources of general-purpose Linux workstations are shared with a number of processes, which makes difficult the execution of time-critical computations without interfered by other processes.

To overcome this difficulty, possible methods are: to provide a DSP-computation process with a higher priority to pre-empt certain resources by means of a real-time kernel or to contribute all resources on a machine to the DSP-computation process, which means to remove all unnecessary processes in the machine.

6. CONCLUSION

In this report, we have described the ESTEREL extensions that we have added to the PSPECTRA toolkit in order to ease the work involved in the debugging and verification phases. ESTEREL not only is suited to the specification of control-driven functions, but also provides simulation and verification of the correctness properties of the system. We have also proposed a Data-Reactive scheduling Model to improve the performance of the system. We have implemented the control part of PSPECTRA in ESTEREL using two scheduling models. We have shown that the EDRM version performs better than the EDPM and the ODPM version using a basic PSPECTRA application and the displaying process is the primary bottleneck in running DSP applications on general-purpose workstations. In addition, auto-generated code is generally larger than hand-written code, but the resources available on general-purpose systems save the cost of this overhead.

In a separate paper, we will focus on the verification issues, in particular the safety property ("bad things will never occur") [13] and the *bounded* liveness property with XEVE. Another aspect of verification that would be interesting to investigate is the checking of the time constraints of the DSP functions which can be carried out using a tool for verifying real-time properties, TAXYS [14] provided by France Telecom R&D.

ESTEREL extension of PSPECTRA is and will be available to develop the following applications: basically, from a AM/FM radio receiver station, further to PAL/NTSC television reception applications, HDTV decoding, UMTS BTS and other applications to which DSPs are currently used.

Acknowledgements

7. REFERENCES

[1] Vanu G. Bose. *Design and Implementation of Software Radios Using a General Purpose Processor*. PhD thesis, MIT, June 1999.

[2] Brett W. Vasconcellos. Parallel signal-processing for everyone. MS thesis MIT, February 2000.

[3] Gérard Berry. The Esterel v5 language primer. http://www.esterel.org, 1999.

[4] Gérard Berry. The constructive semantics of pure esterel, 1999.

[5] Amar Bouali. Xeve: an esterel verification environment. In *the 10th International Conference on Computer Aided Verification*, volume 1427. LNCS, 1998.

[6] E. Jensen. Eliminating the hard/soft real-time dichotomy, 1994.

[7] T. Johnsson. Efficient compilation of lazy evaluation. *SIGPLAN Notices*, 19(6):58–69, June 1984.

[8] R. Jagannathan. Dataflow models, 1995.

[9] E. Sentovitch, H. Toma, and G. Berry. Latch optimization in circuits generated from high-level descriptions, 1996.

[10] E. M. Sentovich, K. J. Singh and *et al*. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, Univ. of California Berkeley, 1992.

[11] Michael B. Jones and Stefan Saroin. Predictability requirements of a soft modem. In *ACM SIGMETRICS Conference*, Cambridge USA, June 2001.

[12] Gérard Berry. Preemption in concurrent systems. In *Foundations of Software Technology and Theoretical Computer Science*, pages 72–93, 1993.

[13] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. ISBN 0-387-97664-7. Springer-Verlag, 1992.

[14] D. Weil, E. Closse and *et al*. Taxys: a tool for developing and verifying real-time properties of embedded systems. In *the 13th International Conference on Computer Aided Verification*, 2001.