Implementation of an Esterel-based Toolkit for Designing DSP Software Applications

Hahnsang Kim and Thierry Turletti

INRIA -Sophia Antipolis
2004, Route des Lucioles BP 93
06902 Sophia Antipolis Cedex France
Tel: +33 4 92 38 75 77 Fax: +33 4 92 38 79 78
{Hahnsang.Kim, Thierry.Turletti}@sophia.inria.fr

Abstract

Designing real-time DSP (Digital Signal Processing) applications is a complex task. It requires a software programming 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 paper, we present* EPSPECTRA*: an* ESTEREL*-based extension of* PSPECTRA *that makes the design and implementation of portable DSP applications easier. It allows one to drastically reduce the testing/verification time while requiring relatively few expertise in formal verification methods. In addition, for the performance, the scheduling model in* EPSPECTRA *has been replaced by a scheduling model called Data-Reactive scheduling Model, which is well suited to describe complex control-paths in the control part of* PSPECTRA*. The performance analysis of the scheduling model applied to* EPSPECTRA *is finally presented and the performance results are promising.*

1 Introduction

Software radios are wireless communication applications in which all of the physical layer functions are implemented in software. The software approach brings many advantages: flexibility, re-usability of resources and easy upgrades of applications. The greatest advantage is the tremendous flexibility of software applications. It allows one 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 real-time 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 this work is to make the implementation of software DSP applications easier by accelerating the development time, especially by supporting the debugging and verification phases. In order not to reinvent the wheel, we have used the PSPECTRA [1, 2] programming environment developed by the SpectrumWare project [1]. PSPECTRA provides a signal processing programming environment to implement portable DSP applications on general-purpose workstations. It is especially targeted to develop software radios and provides the API for developers to make DSP applications on general-purpose workstations.

In this paper, we present EPSPECTRA: an ESTEREL extension we have built upon PSPECTRA, which has been developed in order to provide the debugging and verification phases by means of the ESTEREL [3] environment including XES and XEVE [4]. It allows the developing/testing time to drastically be reduced since the debugging and testing phases are known to be the most time consuming operations. For the performance, we would like to focus on the scheduling model and design a scheduling model called the Data-Reactive scheduling Model, while the real-time scheduling of PSPECTRA is based on Data-Pull Model [1].

The Data-Pull Model in the PSPECTRA programming environment was designed to overcome some of the limitations of a traditional Data-Flow Model implementation. In a typical data flow approach, the data is pushed from the place taking raw data to

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

the place producing the corresponding final data, but in the Data-Pull Model, the execution is driven by needs which request data. Even though the Data-Pull approach has significant advantages (see subsection 3.1), it is not well suitable for the implementation of scheduling in the synchronous ESTEREL language for implementing EPSPECTRA. The control part of EPSPECTRA is built on the basis of the Data-Reactive Model upon a data flow approach [5]. The Data-Reactive Model takes advantage of two features: a software pipelining scheduling method and the virtual sizing technique. These features will be discussed further in subsection 3.3.

The formal language ESTEREL [3] is a synchronous programming language dedicated to reactive systems. ESTEREL programs perform an input-driven computation: wait for inputs and compute corresponding outputs in a cyclic manner, referred to as a reaction. Using ESTEREL, the following advantages are expected. Firstly, it will be easier to write control-paths of handling computation functions. 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 simulation and verification techniques commonly used in such areas as functional process or hardware design, and to extend them to software/hardware applications.

The structure of this paper is as follows. Section 2 describes the PSPECTRA's software architecture which is divided into two parts: the data part and the control part. It also describes EPSPECTRA that we have added in which the control part is re-designed and implemented in ESTEREL. Section 3 compares two different scheduling models: the Data-Pull Model and the Data-Reactive Model. Section 4 shows an example of the application implemented using EPSPECTRA. Section 5 analyses the performance of two applications: one that is included in PSPECTRA and the other that has been re-implemented using EPSPECTRA. The experiment reflexes the features of the Data-Reactive Model. Finally, the last section concludes this paper and introduces future work.

2 Software Architecture

PSPECTRA is a real-time signal processing programming environment used to implement portable DSP applications like software radios on general-purpose workstations. This environment includes a library of portable (across platforms), DSP functions and a 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.

The 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 and all code relating 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: DSP modules and connectors. The DSP modules perform the signal processing tasks and communicate with the control part via the access functions. A connector can be thought of as a wire that carries signals from the output of one processing module to the input of the following processing module. The DSP modules are classified as follows:

- *Sources* are specialised modules that have one or more output ports and no input ports.
- *Sinks* are specialised 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. Each signal processing path has at least one source beginning computation and at least one sink ending it.

2.2 Control Part

The control part is responsible for creating topology and modifying current data flow according to the system needs, controlling the communications between DSP modules, handling user interaction, and monitoring the data computation on each DSP module. The data manipulated by the DSP modules flow from sources to sinks. A DSP module reads input sample data from the preceding DSP modules directly connected by connectors, and performs some computation on it.

To refer to the input and output data in the buffer, a parameter called *SampleRange* is used by the DSP modules. This parameter keeps track of a position of the data that each DSP module's 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.

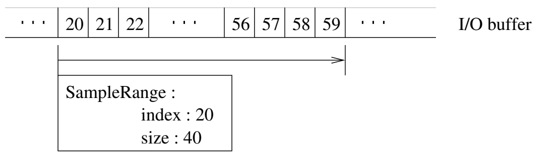

Figure 1: SampleRange: Each data block is referenced with an index and a size

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

2.3 Esterel-based Architecture

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

In an Esterel-based approach, as shown in Figure 2, the first part described in Esterel corresponds to the control part, which creates the components of DSP modules, initialises them and performs scheduling. The second part described in C/C++ is used as an interface to link Esterel-written control part to C++-written data part. The last part described in C++ is the data part in which DSP algorithms are performed.

Figure 3 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).

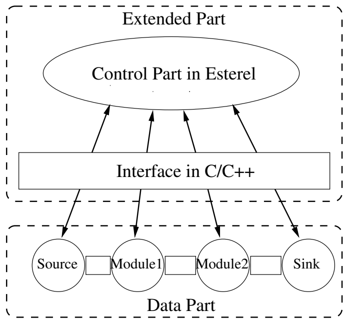

Figure 2: Architecture of EPspectra

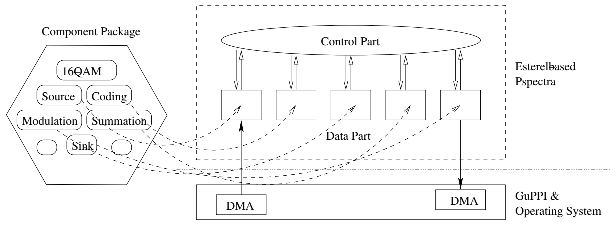

Figure 3: The Esterel-based Pspectra

3 Scheduling Techniques

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 EPspectra runs on general-purpose workstations in an operating system (Linux OS) without *explicit* real-time support. 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, and
- 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.

3.1 DPM: Data-Pull Model

Before looking into the Data-Reactive Model (DRM), let us account for the Data-Pull Model (DPM) on which the control part of PSPECTRA is based. 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 using the DPM in PSPECTRA include: improved computational efficiency resulting from the ability of lazy evaluation, rapid response to changes in the processing requirements, and the caching benefits with good locality of data reference by means of lazy evaluation. The more details of these advantages are described in [1].

However, the DPM fails to take advantage of parallel computing between the DSP scheduling modules. That is the fact that PSPECTRA makes use of parallel processing for data computation of DSP modules by means of multiple threads. Nevertheless, the overhead of synchronisation between threads which share the same data processed may degrade the performance of parallel processing. Suppose that there is an application as follows: it consists of two sources, two sinks, and several intermediate modules where there exists an intermediate module connected to two sinks and there is nothing to do in between the two sinks. According to the DPM, only one of the two sinks can be processed in an alternative manner, that is, the other sink does nothing regardless of no constraint on the current performing sink. In addition, when the sequential processing chain is created, a sample data is processed by passing it 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 chain of the current sample data and that of the next sample data.

3.2 DRM: Data-Reactive Model

In contrast to the DPM, the DRM makes the most of a software pipelining method [8], which allows one to reduce the idle time between the beginning and the end of computation operations. It leads to the speed up of computation operations and acceleration of computation-intensive scheduling. Figure 4 shows the architecture of the DRM specified in ESTEREL. All the modules wait for input signals and compute corresponding output signals. It allows us to benefit from the well-formed semantic properties of ESTEREL such as parallel composition and hierarchical automata, which is introduced in [3].

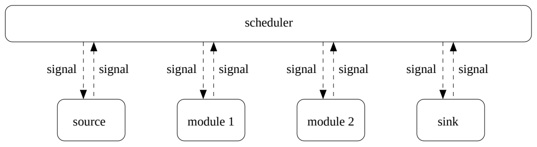

Figure 4: Data-Reactive Model

The data computed from the source is pushed into the sink through the operations of the intermediate modules. When all DSP modules react on available data, the relation among DSP modules is determined by a scheduler that decides which DSP module starts and stops its computation. The scheduling approach is as follows:

- data computation starts on the source.
- whenever data on DSP modules are available, they start computing it.
- the corresponding data is completed on the sink.

All DSP modules communicate with each other by passing signals controlled by the scheduler. As soon as sources finish computing the data, they emit some signals triggering the computation of the corresponding data on the following modules and then wait for ack (acknowledgment) signals from them. DSP modules wait for two events: available data from the preceding modules and ack signals from the following modules that indicate the completion of computation of the previous data. Having received both, the DSP modules compute the available data, and then transmit the computed data for the following modules, and emit ack signals to the preceding modules simultaneously. The corresponding data are finally consumed on the sinks.

3.3 Features of the DRM

Scheduling in the DRM starts from the source and ends upto the sink, while the DPM's scheduling starts from the sink to the source and turns back to the sink. The DRM has two features of scheduling: a software pipelining scheduling method and the virtual sizing technique of the sinks.

The software pipelining scheduling method utilises parallel processing among DSP modules *at the operation-scheduling level*, not at the instruction level. Let us look at the loop body of Figure 5(a). Each set **M** [2] of an iteration depends on the previous set of operations as well as the previous iteration. As shown by the execution schedule of Figure 5(b), the set of operations of the 2nd iteration of **M1** depends on the set of operations of the 1st iteration of **M2**. That is, the set of operations of the 2nd iteration of **M1** must follow the set of operations of the 1st iteration of **M2**. From this basic software pipelining scheduling method, it is expected to obtain the speed-up of the execution rate.

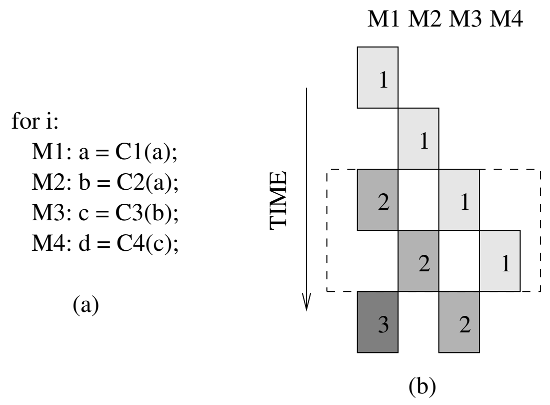

Figure 5: (a) Loop body code. (b) Execution schedule of iterations.

Each DSP module and the sink have a fixed unit size for computation. In terms of the sink, the scheduler makes a call to the sink in order to consume an available input data. Each call activated by the scheduler allows the sink to consume the same amount of the available data as a computing unit size of the sink. If the size of the available data is twice more than the computing unit size of the sink, at least two scheduling calls are required to compute it. The scheduling call overloads the scheduler which is in proportion to the difference between the size of the available data and the computing unit size of the sink.

[2]Note that M represents a set of operations of each module, not an operation itself.

The virtual sizing technique allows the available data given to the sink to be consumed with a scheduling call by the scheduler. It is based on the difference between the computing unit size of the sink and the preceding modules in that the size of the available data produced by the preceding modules corresponds to the maximal computing unit size of what they have. The virtual sizing technique is particularly applied to the sink. So, the sink has two different units: one is the virtual unit that the scheduler refers to for making calls to the sink and the other is the unit that the sink initially contains for computation. Therefore, the scheduler make a call to the sink in reference with the virtual unit size of the sink and the sink performs to make *implicitly* repeated computations according to the unit size of its own until the given available data ends up to consume. The definition of `Compute_virtual_unit` function involved in this technique is as follows:

```
Compute_virtual_unit(){
p_u: the computing unit size of the preceding
    module
u: the computing unit size of the sink
u': the virtual unit size of the sink

if (u < p_u)
  for (i=2; u*i =< p_u; i++);
u' = u*(--i);
}
```

The function provides a virtual unit size of the corresponding sink for the scheduler and The scheduler regards the virtual unit size as the computing unit size of the sink. Let us look at Figure 6 showing the comparison of a different data measurement depending on the DPM and the DRM. The preceding module has the computation unit `p_u`=80000 and the computation unit of the sink is 600. Scheduler on the DPM, as shown in Figure 6(a), performs `i`=133 iterations of processing loop to produce the output data amount to 79800 because of having the unit of 600. In contrast, Figure 6(b) shows that performing only an iteration of processing loop on the DRM allows the sink to produce an amount to 79800 with a virtual unit `u'`=79800 calculated using the above `Compute_virtual_unit` function.

3.4 Data Dependencies

Two features of the DRM are *dependent on data* related to DSP modules of which a DSP application is composed. A *dependence* [8] exists between two operations if interchanging their order changes the results.

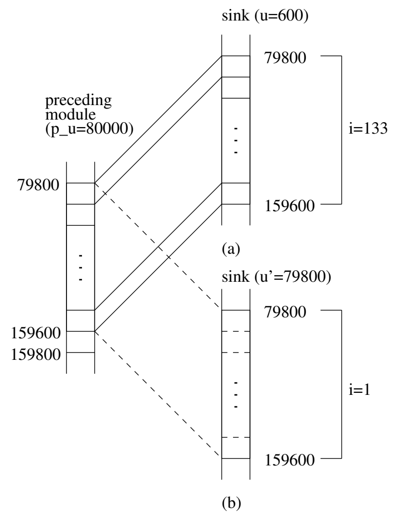

Figure 6: (a) data measurement on the DPM. (b) data measurement on the DRM.

Dependencies constrain what can be done in parallel. Let O_1 and O_2 be operations such that O_1 precedes O_2. O_2 must follow O_1 if O_2 reads data written by O_1. O_2 is said to be *data dependent* on O_1. *Data* dependence between two operations is extended to data dependence between two operational modules. There is another reason that one operation must wait for another operation. A *control* dependence exists between S_1 and S_2 if the execution of statement S_1 determines whether or not statement S_2 is executed. Therefore, even though S_2 is able to execute because of the available data, it may not execute because it is not known whether it is needed.

The current scheduling model based on the DRM considers data dependencies, not control dependencies. Look at an example of Figure 7. This is part of such audio receiver application that switches between AM and FM demodulators. It has data dependencies represented as (1), (2), (3), (4), and (5) and all the statement pertaining to the execution of all modules is able to execute as soon as all the data is available. The control program is required to change the execution topology with the establishment of either (1) and (3) or (2) and (4) after Channel Filter is done. Thus, it is necessary to have control dependencies as well as data dependencies between Channel Filter and AM demodulator or between Channel Filter and FM demodulator. It requires the *dynamic* reconfiguration that enables the execution topology to be adapted to the changeable environment.

In the current DRM, the topology requiring dynamic control dependencies among modules is not yet considered in that ESTEREL is much suited to describe complex and static fixed control-paths.

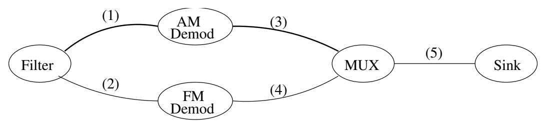

Figure 7: A diagram showing dependencies

4 Example of a DSP application: udp_tx

The udp_tx application has two functionalities: one is to modulate [3] 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 (see Figure 8.).

```
module UDP_TX_PROGRAM:
input <the program inputs>;
output <the program outputs>;
  [ ...
  run source/SOURCE
   ||
  run coding/P_MOD
   ||
  run modulation/P_MOD1to2
   ||
  run scopesink/MAINSINK
   ||
  run summation/P_MOD
   ||
  run udpsink/SINK
  ... ]
end module
```

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 is dispatched to the scopesink and the udpsink.

The source module continuously reads raw data until all of it is consumed. The coding module transforms "bits" from the source 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 sends the corresponding sample data as a UDP packet to

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

the destination that will visualise the constellation diagram.

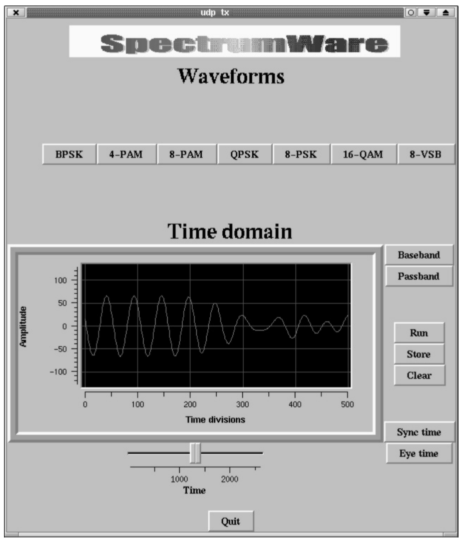

Figure 8: A screen-shot of the udp_tx application

4.1 Scenario of scheduling on the DRM

Figure 9 illustrates the computing procedure on the basis of the DRM. The scheduler triggers on the source the estimating method which sets a SampleRange with index=a and size=400 in reference with its input sampling frequency and the previously computed SampleRange, and its maximum output size. Then the SampleRange on the source starts to compute and at the same time the coding module is informed of the estimated SampleRange. A SampleRange $[a, 400]$ on the coding module is estimated according to the SampleRange received from the source and the information of the coding module (which corresponds the input sampling frequency, the last computed SampleRange, and the maximum output size.) and is computed. The completion of computation on the coding module allows the source to manipulate the following $[a + 400, 400]$ SampleRange and the modulation module to start computing the corresponding $[b, 80000]$ SampleRange. On the modulation module, the information being transfered to the coding module is delayed until both the scopesink and the summation module consume an input SampleRange corresponding to $[b, 80000]$ of the modulation module. The sinks finally consume the $[b, 80000]$ SampleRange corresponding to $[a, 400]$ of the source.

Hereby, the first iteration to obtain the first computed data block is done. According to the software pipelining scheduling method, it is expected that at the end of computing the first data block on the coding module, the source may begin to compute the second data block corresponding to $[a + 400, 400]$. Furthermore, when the modulation module is processing the second data block of the $[b + 80000, 80000]$ SampleRange, the source may process the third data block of the $[a + 800, 400]$ SampleRange.

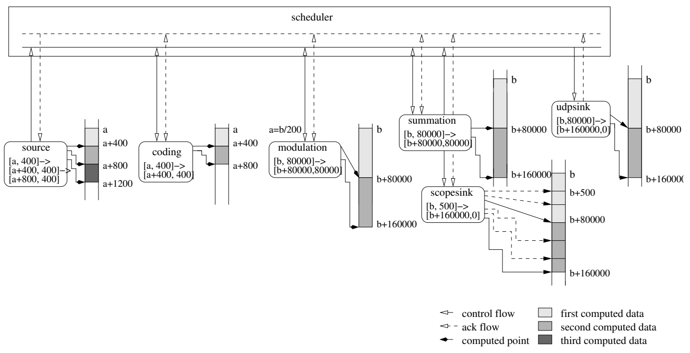

Figure 9: Scheduling Process with the DRM : scheduling starts from source.

Look at the data block computation of the scopesink. The computation unit of the scopesink is 500, but the 80000 amount of the data block coming from the modulation module is uninterruptedly computed since the virtual unit is set as 80000 by the `Compute_virtual_unit` function. Therefore, it allows one to decrease the overhead of repetitions of the function call required to consume all the data. 160 repetitions of call are reduced into one.

5 Performance analysis

Our performance analysis has been carried out on both a PIII 600MHz machine with 256MBytes RAM and a SMP machine with two PII 390MHz and with 256MBytes RAM, on Linux kernel 2.2.15, respectively. We provide a performance comparison of our EPspectra based on DRM (EDRM) and the Original Pspectra based on DPM (ODPM). Note that the udp_tx application uses the 16-QAM modulation algorithm in this experiment. The udp_tx application based on the EDRM is compiled by the version 6.03 of the Esterel compiler [4] and are optimised 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 optimised by Remlatch, and once more re-optimised by

[4]6.0 or later version of Esterel compiler is now developed and maintained by Esterel-Technologies Ltd.

SIS. The optimised Blif code is translated into standard C code by ESTEREL compiler. The executable code [5] is built up by integrating the C++ code of the data part into the C code.

5.1 Performance on a PIII/600MHz machine

Figure 10 shows the performance result of the udp_tx applications based on the EDRM and the ODPM on PIII 600MHz machine. The number of output sample data processed per second is compared with different situations: with and without interfering process using a *grep* command [6]. As of without interfering process, the output sample data produced by the EDRM version is about two times more compared to the ODPM version. At t=30s, the EDRM version produces 2.5Msps (4-bit samples per second) (i.e. more than twice as many as the ODPM which produce 1.3Msps). The EDRM version enhances the performance about 45% over the ODPM version by means of the software pipelining method and the virtual sizing technique.

The interfering process, *grep* searches the entire file system running concurrently. The disk activity induced by *grep* is expected to interfere with the ram disk access of the udp_tx application. We see that there is some jitter caused by *grep*. At t=25s, the EDRM version produces about 2Msps, whereas the ODPM version produces 1Msps. *Grep* degrades about 20% of the performance of both, which means that it is required to satisfy resource requirements of the applications running on standard workstations (see Figure 11.).

5.2 Performance on bi-PII/392MHz SMP machine

Figure 11 shows the performance result of the udp_tx applications on bi-PII 392MHz machine. The comparison of the number of output sample data processed per second is made according to with and without interfering process. At t=30s, the EDRM and ODPM versions produces about 1.7Msps and 600Ksps. We see that the *grep* process does not interfere the process of the udp_tx application because resource requirements of two processes is satisfied on the bi-processor machine.

5.3 Comparison of LoC (Lines of Code)

Table 1 gives the comparison of the loc of udp_tx based on two different versions for the control part.

Table 1: Comparison of loc

code line\model	EDRM	ODPM
ESTEREL code	978	*not used*
generated C (non-opt)	5253	*not used*
generated C (opt)	4749	*not used*
C/C++ for interface	2520	*not used*
C++ for control part	*not used*	8045
Sum Total (opt)	8247	8045

We note that this measurement includes the control part of udp_tx on the EDRM and the ODPM [7]. . The EDRM version has 978 lines of ESTEREL code which is translated into C code with 5253 lines. We have obtained about 9.6% code optimisation of the EDRM version using the REMLATCH and SIS optimisors.

In terms of the loc of the EDRM 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 and broadcast as well as synchrony, simulation, verification, etc.

5.4 Pros & Cons

EPSPECTRA provides a solution to provide the debugging and verification phases for designing and developing basic DSP software applications. In addition, the DRM provides good performance results for DSP software applications with a well-formed dependence topology as it fully utilises a basic software pipelining method and the virtual sizing technique of the sinks.

Nevertheless, since it is based on a basic technique, the software pipelining method applied to our scheduling model in ESTEREL is weak to applications which have the topology with *implicit* control dependencies that requires the dynamic reconfiguration of the execution topology. It has a hardness about the dynamic reconfiguration in that ESTEREL is much suited to describe complex and static fixed control-paths.

6 Conclusion

In this paper, we have presented EPSPECTRA in order to ease the work involved in the debugging and verification phases of developing DSP software applications. ESTEREL not only suites for the specification of control-paths, but also provides simulation and verification phases for the correctness properties of the system. The control part of EPSPEC-

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

[6] 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].

[7]The code corresponding to the data part of the EDRM is the same as is utilised in that of the ODPM

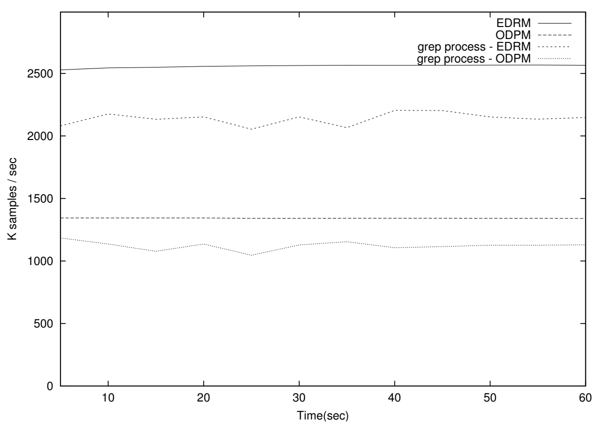

Figure 10: Comparison of number of processed samples per second

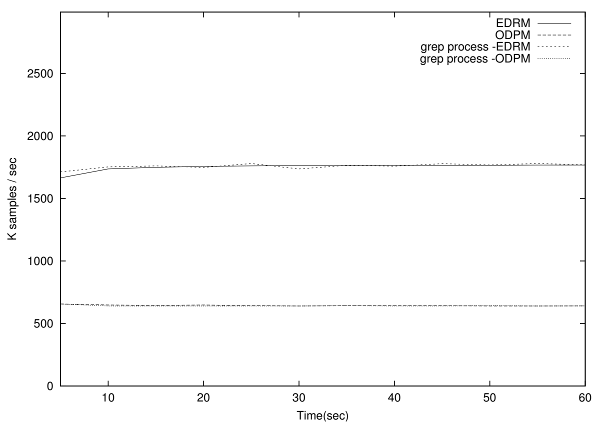

Figure 11: Number of processed samples per second without interfering process

TRA is based on a Data-Reactive scheduling Model (DRM) for scheduling DSP modules, instead of Data-Pull model on which the control part of PSPECTRA is based. The DRM utilises the software pipelining scheduling method and the virtual sizing technique of the sinks. These methods improve the performance of DSP applications with a static scheduling configuration since they have to obey strict scheduling rules.

In a separate article, we show the ESTEREL methodology, especially focusing on the verification issues: the safety property ("something bad will never happen.") and the *bounded* liveness property ("something good will eventually happen in a certain times unit."). Another aspect of verification that would be interesting to investigate is to check timing-constraints of scheduling DSP functions, which can be carried out using a toolkit for verifying real-time properties, TAXYS [12] provided by France Telecom R&D.

Acknowledgments

We gratefully acknowledge discussions about PSPECTRA with John C Ankcorn at MIT and about ESTEREL methodology with Robert de Simone and Amar Bouali at INRIA-Sophia Antipolis. This research was supported by the DESS project associated with Information Technology for European Advancement (ITEA).

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 constructive semantics of pure esterel, 1999.

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

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

[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] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipeling. *SIGPLAN Notices*, 27(3):367–432, September 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] 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.