ComPAS'2013 : RenPar'21/ SympA'15/ CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

A QoS Monitoring System for Dataflow Programs

Manuel Selva, Lionel Morel, Kevin Marquet, Stéphane Frénot

Bull Échirolles
1, rue de Provence
38432 Échirolles Cedex - France
manuel.selva@ext.bull.net

Université de Lyon,
INSA-Lyon, CITI-INRIA F-69621, Villeurbanne, France
prenom.nom@insa-lyon.fr

Abstract
With the generalization of multi-core processors, dataflow programming is regaining a strong interest, especially in the context of compute intensive interactive applications such as video decoding. However, most studies focus on static approaches to the compilation and placement problems. We build the first step towards dynamic adaptation of dataflow applications, namely a monitoring mechanism for observing quality-of-service properties of programs at run-time. We propose a language extension for expressing simple QoS properties over dataflow programs together with a run-time mechanism for the observation of events meaningful to the QoS establishment. We show the limited impact of such mechanisms on the application overall performances.

Keywords: dataflow, quality of service, dynamic adaptation, multi-core

---

1. Introduction

Motivations

It is now widely accepted [4, 9] that the future of computing lies in the efficient use of multi-core architectures. These are already predominent in mainstream computing devices. The traditional gap between desktop computers and portable devices such as cell phones is reducing dramatically and multi-core are becoming a *de facto* standard.
At the same time, the question of which programming model to adopt remains open. The thread-like approach has been prevalent over uni-processor for now, but strong arguments go against its use even in this setting [20] and many alternatives emerge in the context of multi-core platforms [25, 5, 30, 22]. In particular, many classes of programs may benefit from other programming models, both from the expressiveness and the optimization point-of-views. Our work focuses on the stream oriented programming paradigm.
A stream program is a set of computation actors that consume and produce elements from streams of data. Dataflow programming models are well-suited for addressing this setting for two reasons. One is that the dataflow programming style fits these applications well: they are dataflow by nature. The other one is that it allows to exploit multi-core processor advantages much more easily than other programming models. Indeed, as it separates communications from computations, it allows to schedule one independently of the other; hence, the latency of memory accesses can be hidden. Moreover, streaming languages expose rather than hide data and computation parallelism, making it easier for compilers to cope with this parallelism in order to benefit from multi-core processors.

In this work we particularly focus our attention on compute intensive applications. These applications may imply dynamic variations of their computation load. In addition, due to their high interactivity, it is important that they maintain a given *quality-of-service* (QoS) during their execution.

Long-term Proposition

We aim at designing a solution where a QoS can be defined and attached to dataflow programs, observed during execution, and where dynamic adaptation mechanisms can be used to maintain this expected QoS. Parts of these goals have already been addressed in the literature but have never been considered, to the best of our knowledge, for dataflow applications. However, this programming model raises different problems and has huge impact from the source code to the execution engine and the compiler.
First, such a scenario requires to investigate on how a QoS can be expressed and attached to dataflow programs. This QoS can be defined using various criteria that may be user-, application- or platform-dependant.
Second, the run-time system must be able to collect information about applications in order to verify that the desired QoS is reached or not. We propose to decompose this collecting into two parts. On one hand, a *reporting* mechanism allows applications to record meaningful information. On the other hand, this information is used by a system-wide *monitoring* component to compute an observed QoS. This monitor then compares it to the expected QoS.
Finally, the run-time system must be able to dynamically adapt itself in the case where an application does not reach its desired QoS anymore. Dynamic adaptations include migrating processes from one core to another, placing actors on the cores in order to optimize memory accesses, or disabling an actor.

Contributions

In this paper, we make the following contributions:

1. we define the notion of QoS for dataflow programs, with a focus on throughput,
2. we detail how dataflow applications can be monitored in order to verify that the QoS expressed by the programmer is respected at run-time,
3. we present a prototype system able to dynamically adapt in order to make real dataflow applications respect the desired QoS. We report on the impact of the reporting/monitoring activity on the performances of the system based on two realistic examples: a gzip decoder and an mpeg decoder,
4. we present a number of future works opened by this work.

The rest of the paper is organized as follows. Section 2 presents the notion of QoS for dataflow programs. Section 3 details our solution for monitoring programs. Section 3.4 briefly introduces potential dynamic adaptations. Section 4 evaluates the performances of the whole system on a multi-core platform. Section 6 concludes and details the future works of this proposal.

2. QoS for Dataflow programs

We consider programs implemented in a general dataflow language, directly inspired by [21]. These are a generalization of Khan Process Networks [18] and of many of the commonly used dataflow programming models or languages found in the literature e.g. Synchronous DataFlow[19], Cyclo-Static DataFlow[8], CAL[13], synchronous languages like Lustre or Signal[6], etc.
We first briefly recall the foundation of this model, before introducing QoS information.

2.1. Dataflow processes

A dataflow *process*, or actor, $\mathcal{P}$ represent a computation unit. Processes are connected with each other through *channels*. (see FIG. 1). They consume tokens on their input channels and produces tokens onto their output channels.
A channel has exactly one process that can write to it and may be read by exactly one process. It is manipulated in a First-In-First-Out (FIFO) manner. At a given time, a channel contains all the tokens produced by the writing process but not yet consumed by the reading process. These FIFO channels

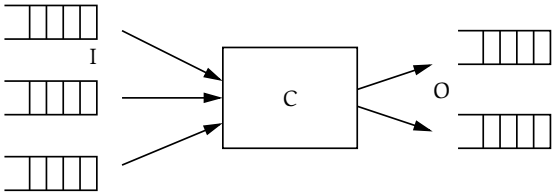

Figure 1: A dataflow component, with its input and output FIFO channels.

are the only way through which processes can communicate, which makes this a functional model well-suited for compositional reasoning. Processes and channels form dataflow networks that can be hierarchically composed to form complex systems.

The execution of a process is driven by the availability of tokens on its input channels. From an external point of view, a process can perform two operations: *reading* tokens from its inputs channels and *writing* tokens to its output channels. This essentially defines the model of communication of programs. The complementary model of computation expresses how new values are computed before they are communicated on output channels. We do not detail it here.

Original Dataflow Process Networks [21] assume possibly infinite channels. From an implementation point of view, this is clearly non realistic and bounded FIFOs need to be used. For now, we do not say whether this should be syntactically decided or checked by the compiler. We need only to decide what happens when a channel is full or empty. When an input channel is empty, the component that reads it is blocked until a new token appears in the FIFO. From the output point-of-view, an output channel being full leads the component writing to it to block until the component reading from the channel consumes at least a token.

A component is activated whenever tokens are available on any of its input channels, depending on firing conditions. In the synchronous dataflow model, the number of tokens needed by the component is known *a priori* and the actor fires when all its input channels contain the specified number of tokens.

2.2. Example program

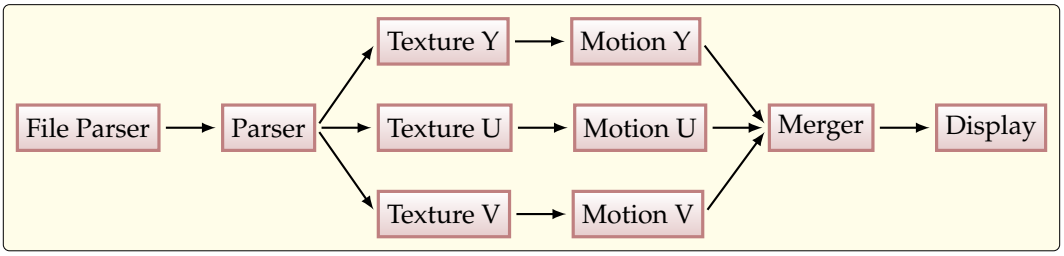

Figure 2: Graphical representation of the MPEG 4 part-2 Simple Profile Decoder

MPEG 4 part-2 simple profile is a standard describing a way to encode video streams. It is a typical dataflow application in which video streams are compressed using two distinct properties. The first one, called texture encoding, consists in compressing individual frames. The second one exploits commonalities between successive frames and is named motion — or temporal — compression. These two decoding steps must be applied to the three chromatic components (YUV) used to represent pixels in the video. The dataflow representation of this application clearly expresses the underlying parallelism. As shown in 2, Y, U and V decoding can be performed in parallel.

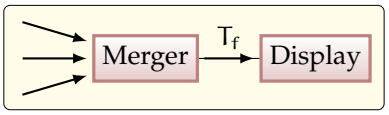

Figure 3: The mpeg example tagged with throughput information (in f/s).

2.3. Expressing QoS Properties

Quality of service covers a large set of properties of programs. The properties we consider can be expressed at application- or resource-level [16]. In the case of our video decoding example, these levels are illustrated as follows. Application-level QoS concerns properties that are hardware and OS independent. This concerns either quantitative aspects (e.g. video frame rates) or qualitative ones, e.g. inter or intra-stream synchronization schemes. Resource-level QoS properties are OS and hardware dependant level of application properties. They include throughput, delay or delay jitter for particular data streams in the program, constraints on memory size used.

Our QoS criteria concern throughput properties. In the following, we denote ETP the expected throughput. This can be expressed either on channels or inside actors. On a channel, it represents the number of tokens that enter the channel every time unit. Inside an actor, a throughput property needs to be defined by the programmer, e.g. as the number a given atomic action is performed every time unit. The latter is more intrusive and we shall prefer expressing throughput objectives on channels. In FIG. 3, the channel connecting the `Merger` and `Display` actors is now tagged with a throughput information T_f, expressed in numbers of frames per second.

3. Monitoring QoS Information

We now describe our approach for observing applications and building QoS information. We propose a method that:

- represents a minimal effort form the programmer's point of view. In particular, it should require only minimal changes to the programs, and certainly not introduce any shift in the programming model,
- has a minimal impact on the overall performances of the system. The global QoS of applications we wish to manage should not be impended by the QoS manager itself,
- is generic enough so we can build various kinds of QoS information on top of it.

In this section we first introduce the execution model we target. We then review how reporting and monitoring can be included in this execution model.

3.1. Targeted Execution Model

We focus on dataflow programs executed on general purpose operating systems using the thread execution model. While being not appropriate as a programming model [20], we believe that it should be kept as an execution model because of its wide support and its efficiency on commodity hardware. The underlying hardware is considered to be SMP.

Targeting this execution model, a dataflow compiler associated with a dataflow run-time is responsible for transforming and then executing programs using threads and allocating them to cores of the SMP platform. Existing dataflow compilers usually generate low-level source code (such as C or C++) that will need standard compilation tools to be effectively executed. The compiler is also responsible for generating the code orchestrating the execution of the dataflow actors.

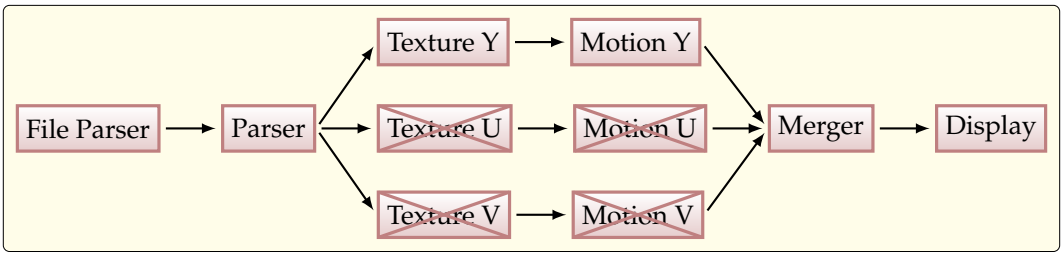

Figure 4: Optional Processes in MPEG 4 part 2 Simple Profile Decoder

3.2. Reporting

The reporting mechanism is the one responsible for gathering and storing information allowing to compute the effective QoS, denoted OPT (Observed ThroughPut). As stated in section 2.3, we use token counts to compute the OTP. This information is stored in a memory segment located in the process containing the threads implementing actors. This segment will be shared with the process implementing the monitoring and decision mechanisms.

For implementing the reporting mechanism, we adapt both the compiler and run-time support. For each token written on an observed channel, the instrumented version of the component will increment the token counter with the number of tokens written.

3.3. Monitoring

To compute the effective QoS, the *monitor* checks the token counts provided by the reporting system at regular time intervals and performs simple arithmetic to compute the OPT. Since we wish to manage a set of such monitored applications, the monitor must be system-wide.

When targeting standard operating systems, we implement this mechanism in a dedicated process. The communications between the applications and the monitoring process is implemented by a shared memory segment.

An important parameter is the frequency of the monitoring because of its potential impact on performances : the higher the frequency, the bigger the overhead. This frequency is directly linked to the expected throughput rate: in the case of a QoS of 25 frames per second, we don't need to monitor every millisecond. This frequency is strongly application-dependant.

3.4. Decision making and dynamic adaptation

The monitoring system aims at comparing the OTP and the ETP. If the OTP is lower than the ETP, dynamic adaptations must be performed. We do not focus on the possible adaptation mechanisms and leave it to future works.

However, we validate this proposal through a simple application-specific adaptation: deactivation of actors. FIG. 4 shows an example of optional actors in the mpeg decoder. When the frame per second rate becomes too low, the decision making process may decide to switch to black-and-white decoding by deactivating U and V decoding actors.

Other adaptation mechanisms are application-agnostic such as load balancing by process migration or process placement for optimizing memory accesses. We plan to investigate these in the near future.

4. Experiments

We instantiate the concepts introduced above (QoS expression, reporting, monitoring and adaptation) on the RVC-CAL language and associated run-time [7]. We run our experiments on the mpeg simple decoder and the gzip decoder mentioned earlier. The gzip application is depicted in FIG. 5.

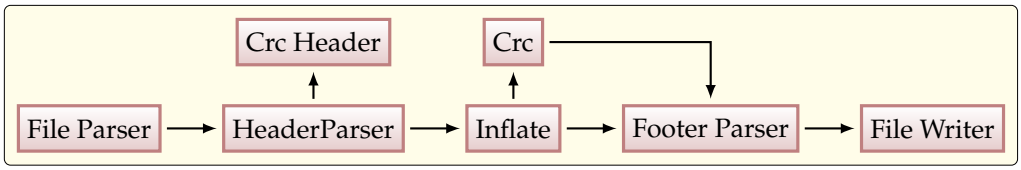

Figure 5: Top level view of RVC-CAL Gzip Decoder

4.1. RVC-CAL

RVC-CAL is a dataflow programming language defined by the MPEG group in the Reconfigurable Video Coding (RVC) initiative.

RVC-CAL is an extension of the CAL actor language [13, 15] that provides a framework for video codec development. It deals with fully typed actors only and adds some restrictions on the CAL language constructs to have efficient hardware and software code generation without tampering the expressiveness of the language. The underlying model of computation is dataflow process network in its most general sens: non-determinism can be expressed and dynamic scheduling is required in the general case. Nevertheless, the language lets static tools easily analyze programs to know into which model of computation they fall. For instance, in the case of a pure synchronous dataflow graph, a static scheduling can be computed to avoid dynamic scheduling overhead.

RVC-CAL actors may have several input and output ports. Actors may have one or more actions associated to firing conditions. These might depend on the presence, and possibly the value of input tokens and the actor's internal state. Priorities between actions can be defined.

The complete dataflow graphs are described using an XML-based language called XDF in which actors are instantiated and input and output ports connected to one another.

The compiler and run-time support for the RVC-CAL language is called Orcc [1]. It provides several back-ends to generate code for different software and hardware platforms. As already stated we focus on the C back-end generating either single or multi-threaded C programs to be compiled and run on POSIX platforms. The *main* function generated from an RVC-CAL application execute an infinite loop that schedules actors. Orcc provides two strategies for actor scheduling. The first one implements a round-robin policy. The second one[32] applies either data-driven or demand-driven policies. The compiler is also able to generate a threaded version of this application. The number of threads and the partitioning of actors into these threads must be specified by the programmer and passed to the compiler as an optional file. Scheduling of this multi-threaded version is left to the underlying operating system.

4.2. Extending RVC-CAL

We extend the XDF language to let the programmer express the ETP either at the application level or at the communication channels level. In both cases, we modify the compiler to take into account this information and send it at initialization to the monitoring process described below. At run-time, this information is stored, in the monitoring process, in a simple C data structure containing the ETP's *volume* and *time unit* info.

The reporting mechanism is in charge of collecting information so that the monitor can compute the OTP for each observed channel. This information basically consists of a counter incremented every time a channel is written to. For minimal overhead, these counters are stored in shared memory.

For ETP expressed on channels, the reporting code is generated by the compiler. When the QoS is expressed at the application level, the programmer is responsible for instrumenting its RVC code to perform the increment. For this purpose we add a dedicated API in RVC-CAL allowing programmers to initialize the ETP *volume* counter and then to increment it.

The monitoring process is a dedicated operating system process, responsible for monitoring all registered applications. It is implemented as a server listening for application connections on a UDP socket. The monitoring frequency can be configured. When the monitoring process detects a difference between the observed and expected QoS the decision making mechanism is invoked. Section 4.3 evaluates the overhead for different frequencies.

4.3. Performance evaluation

The experiments were performed on an Intel Core-i5 with a dual core running at 2.50GHz, running a 64 bits SMP 3.2.0-30-generic Linux kernel. We report here the global cost of our mechanisms including both reporting and monitoring overheads. We compile each application to a single thread. To guarantee the monitor to execute frequently enough, we use Linux real-time priorities.

We present here two different overhead measurements. The first one measures application and monitoring execution time only, without taking interference from the operating system and other processes into account. It uses the Linux *clock_gettime()* function that is implemented using the CPU's Time Stamp Counter (TSC) register available on x86 and x86_64 architectures.

The second overhead measurement compares applications' execution wall-clock times. Contrary to the previous one, it accounts also for the time spent in kernel functions such as scheduling time and I/O waiting time.These measurements are done with the Linux *gettimeofday* function. To evaluate the worst possible overhead, we tie applications and monitor to same core.

All the execution times presented correspond to the average of 100 runs of the corresponding application. In both experiments, we used the gzip and mpeg decoders with arbitrary inputs. The gzip input file is an archive of a folder first transformed to a single file with the *tar* tool. It's size is 1.9MB resulting in 5.4 MB after decompression. The mpeg decoder input file is a video encoded with I-frames, P-frames and B-frames accounting for a total of 108 frames. The video decoding is done 3 times in each execution and thus results in 324 frame decodings.

For each application, we measure the overhead when adding an ETP 1) on only one channel and 2) on each channel of the dataflow graph (12 for the gzip decoder and 143 for the mpeg decoder). Another parameter in the experiments is the monitoring frequency. The chosen frequencies correspond to monitoring with a period of respectively 40ms (25 events per second), 100µs and 1µs.

In the following, we denote $UExT$, resp. $MexT$, the execution time of the un-monitored, resp. monitored, programs.

Results without OS interference

Tables 1 and 2 present monitoring overheads computed by comparing the execution time of the monitoring process to that of the initial application.

From these numbers, we see that the monitoring frequency is the crucial parameter impacting performances. For both applications, the performance overhead due to the monitoring becomes non negligible from 10KHz upwards; this corresponds to a check every 100µs.

The number of channels monitored has a limited impact. At 10KHz, the difference is only 1.1 point.

App.	*UExT (s)*	*MExT (s)*			*Overhead (%)*		
		0.25Hz	10KHz	1MHz	0.25Hz	10KHz	1MHz
gzip	6,832	0,001	0,111	8,309	0,010	1,620	121,631
mpeg	1,850	0,000	0,049	1,458	0,023	2,658	78,826

Table 1: 1 QoS Overhead measured with processes time

Application	*UExT (s)*	*MExT (s)*			*Overhead (%)*		
		0.25Hz	10KHz	1MHz	0.25Hz	10KHz	1MHz
gzip	6,832	0,001	0,117	9,477	0,010	1,712	138,719
mpeg	1,850	0,000	0,069	2,495	0,026	3,756	134,914

Table 2: All Channels QoS Overhead measured with processes time

Wall-clock time results

Tables 3, 4 present the overhead using wall-clock time and, as a consequence, accounts for interferences due to system tasks. Here, the overhead is computed by comparing the wall-clock time of the initial application with the wall-clock time of monitored applications. Recall that we pined the monitoring process to the same core than the application process. Not surprisingly the overhead is greater in that case because it accounts for system tasks. Scheduling between application and monitoring processes becomes more and more expensive as the monitoring frequency increases. The negative overhead value results from a benchmarking artifact.
FIG. 6 summarizes these results and shows the overhead as a function of the monitoring frequency. The plateau reached by the overhead values when approaching 10^6Hz shows the point from which the scheduling policy gives the minimum CPU time to the application and cannot schedule the monitoring process more frequently.

Application	*UExT (s)*	*MExT (s)*			*Overhead (%)*		
		0.25Hz	10KHz	1MHz	0.25Hz	10KHz	1MHz
gzip	6,885	6,937	7,112	23,228	0,747	3,289	237,370
mpeg	1,946	1,979	2,077	4,712	1,698	6,704	142,132

Table 3: 1 QoS Overhead measured with wall-clock time

Application	*UExT (s)*	*MExT (s)*			*Overhead (%)*		
		0.25Hz	10KHz	1MHz	0.25Hz	10KHz	1MHz
gzip	6,885	6,870	7,138	24,684	−0,223	3,670	258,510
mpeg	1,946	1,991	2,148	5,816	2,289	10,359	198,864

Table 4: All Channels QoS Overhead measured with wall-clock time

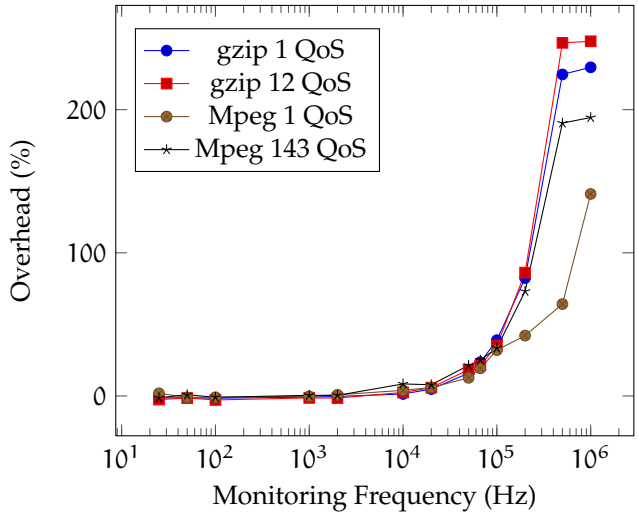

Figure 6: Monitoring overhead for different frequencies

Dynamic adaptations

We implemented an adaptation mechanism consisting in disabling some actors in the dataflow graph. In the gzip decoder, we are able to disable the `CRC` actor if need be. Running the mpeg decoder within an overloaded system, we measured an OTP of 23 f/s. The monitor detected that the ETP of 25 f/s is not met and decided to switch the mpeg decoder to black-and-white by deactivating U and V decoding actors, allowing to obtain 32 f/s. We obtain comparable results by migrating processes on cores with low computation load.
These simple mechanisms show the interest of our monitoring solution. We intend to identify more accurate dynamic adaptations. However, this is a separate concern that is left to future work.

5. Related Works

We divide related works into three categories : research efforts around dataflow models, dynamic optimizations for execution of standard programs on multi-core platforms, and definition of applicative QoS. We now review them and exhibit the originality of our work.
There has been a rich body of research on dataflow models of computation (MoC) [18, 19, 21, 8, 17]. We clearly do not define new models of this kind but rather base our formalization effort on existing, well-defined MoCs. Different execution layers for dataflow programs have also been set up [10], even for multi-core machines [23, 32] but they clearly do not concern dynamic optimizations nor the respect of QoS properties. [3] tackles dynamicity in an ad-hoc manner while we try to be as generic as possible.
Works attempting to optimize program performances are of course numerous. Among them, we are only interested dynamic techniques including thread migration [12, 27], dynamic scheduling and placement techniques such as affinity scheduling [24], exploitation of cache locality [31] or clustered-affinity scheduling [29], or processor frequency adaptation (DVFS). However, these works only base on thread execution model and do not consider dataflow actors. Recent work[32] reports two strategies for scheduling dataflow programs on a multi-core platform but does not address the respect of per-application quality of service.
The last category includes numerous contributions to the definition of quality of service. We already mentioned Jin and Nahrstedt's taxonomy [16]. In future works, we intend to draw from existing work such as [14] and [2] to build a QoS description language adapted to dataflow programs.

6. Conclusions and Future Works

We presented a first step towards a run-time system for dynamic adaptation of dataflow applications based on quality-of-service requirements. We focused primarily on reporting and monitoring mechanisms both at the language level and at the run-time level. This approach has been validated on a couple of real-life applications based on the RVC-CAL programming environment. We have shown that the reporting/monitoring mechanisms have a very low overhead and can be efficiently set up for dataflow applications.
First, we want to build QoS management solutions independent from a particular dataflow language and run-time support. Although we believe the present work is not tightly coupled to RVC-CAL, we want to conduct comparable studies on different dataflow languages. The next step in that direction will be to adapt our proposal to the StreamIt [28] infrastructure.
Second, we ran our monitoring mechanism together with only one dataflow application. In the short term, we intend to adapt it so that it can monitor different applications. Moreover, the possibility to attach ETPs to different channels in dataflow graphs allows to imagine dynamic adaptations at the granularity of actors and not only of applications.
We then wish to study the dynamic adaptation mechanisms themselves. This opens to a wide research area that includes load balancing, cache optimization, routing, etc. Our primary goal here will be to study if the dataflow model can be of any use for making wiser decisions on some of these problems. Unplugging a given actor is rather simple. Plugging components back is more difficult in the general case : how to ensure that a new mapping of actors to core will not penalize other applications ? Operating systems and hardware mechanisms such as cache behavior need to be anticipated so as to guarantee QoS constraints of all applications.

In the long term, we will study precisely the influence of the underlying execution model. RVC-CAL actors are compiled by orcc to pthreads. The final behavior of our application thus depends on the scheduling policy used by the operating system. In the line of [11] or [26] we need to identify the adequacy between dataflow models and the scheduling policy. We also wish to investigate the mapping of our QoS framework to novel execution models for multi-core platforms, such as Helios [25], Barrelfish [5], fos [30] or Tessellation [22].

Bibliography

1. `http://orcc.sourceforge.net/`.
2. Aagedal (J.). – *Quality of Service Support in Development of Distributed Systems*. – Thèse de PhD, Departamento de Informática, Facultad de Matemáticas y Ciencias Naturales, Universidad de Oslo, 2001.
3. Albers (a. H. R.) et With (P. H. N.). – Task complexity analysis and QoS management for mapping dynamic video-processing tasks on a multi-core platform. *Journal of Real-Time Image Processing*, février 2011.
4. Asanovic (K.), Bodik (R.), Demmel (J.), Keaveny (T.), Keutzer (K.), Kubiatowicz (J.), Morgan (N.), Patterson (D.), Sen (K.), Wawrzynek (J.), Wessel (D.) et Yelick (K.). – A view of the parallel computing landscape. *Commun. ACM*, vol. 52, n10, octobre 2009, pp. 56–67.
5. Baumann (A.), Barham (P.), Dagand (P.-E.), Harris (T.), Isaacs (R.), Peter (S.), Roscoe (T.), Schüpbach (A.) et Singhania (A.). – The multikernel: a new os architecture for scalable multicore systems. *In : Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. pp. 29–44. – New York, NY, USA, 2009.
6. Benveniste (A.), Caspi (P.), Edwards (S. A.), Halbwachs (N.), Le Guernic (P.) et De Simone (R.). – The synchronous languages 12 years later. *Proceedings of the IEEE*, vol. 91, n1, January 2003.
7. Bhattacharyya (S. S.), Eker (J.), Janneck (J. W.), Lucarz (C.), Mattavelli (M.) et Raulet (M.). – Overview of the MPEG Reconfigurable Video Coding Framework. *Journal of Signal Processing Systems*, vol. 63, n2, juillet 2009, pp. 251–263.
8. Bilsen (G.), Engels (M.), Lauwereins (R.) et Peperstraete (J.). – Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, vol. 44(2), 1996, pp. 397–408.
9. Borkar (S.) et Chien (A. A.). – The future of microprocessors. *Commun. ACM*, vol. 54, n5, mai 2011, pp. 67–77.
10. Boutellier (J.), Sadhanala (V.), Lucarz (C.), Brisk (P.) et Mattavelli (M.). – Scheduling of dataflow models within the reconfigurable video coding framework. *In : SiPS*, pp. 182–187.
11. Caspi (P.), Scaife (N.), Sofronis (C.) et Tripakis (S.). – Semantics preserving multitask implementation of synchronous programs. *ACM TECS*, 2008.
12. Choffnes (D.), Astley (M.) et Ward (M. J.). – Migration policies for multi-core fair-share scheduling. *SIGOPS Oper. Syst. Rev.*, vol. 42, n1, janvier 2008, pp. 92–93.
13. Eker (J.). – CAL language report. *University of California at*, 2003.
14. Frolund (S.) et Koistinen (J.). – *QML: A Language for Quality of Service Specification*. – Rapport technique nHPL-98-10, Hewlett-Packard Software Technology Laboratory, February 1998.
15. Janneck (W.). – An introduction to the Caltrop actor language, 2001.
16. Jin (J.) et Nahrstedt (K.). – Qos specification languages for distributed multimedia applications: a survey and taxonomy. *MultiMedia, IEEE*, vol. 11, n3, july-sept. 2004, pp. 74 – 87.
17. Johnston (W.), Hanna (J.) et Millar (R.). – Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, vol. 36, n1, mars 2004, pp. 1–34.
18. Kahn (G.). – The semantics of simple language for parallel programming. *In : IFIP Congress*, pp. 471–475.
19. Lee (E.). – Synchronous data flow. *Proceedings of the IEEE*, vol. 75, n9, 1987, pp. 1235–1245.
20. Lee (E.). – The problem with threads. *Computer*, vol. 39, n5, 2006, pp. 33–42.
21. Lee (E.) et Parks (T.). – Dataflow process networks. *Proceedings of the IEEE*, vol. 83, n5, mai 1995, pp. 773–801.
22. Liu (R.), Klues (K.), Bird (S.), Hofmeyr† (S.), Asanović (K.) et Kubiatowicz (J.). – Tessellation: Space-time partitioning in a manycore client os. *In : HotPar '09: Proc. 1st Workshop on Hot Topics in Parallelism*.

23. Lucarz (C.), Roquier (G.) et Mattavelli (M.). – Reconfigurable Video Coding on Multicore. *IEEE Signal Processing Magazine*, noNovember, 2009.
24. Markatos (E. P.) et LeBlanc (T. J.). – Using processor affinity in loop scheduling on shared-memory multiprocessors. *In : Proceedings of the 1992 ACM/IEEE conference on Supercomputing*. pp. 104–113. – Los Alamitos, CA, USA, 1992.
25. Nightingale (E. B.), Hodson (O.), McIlroy (R.), Hawblitzel (C.) et Hunt (G.). – Helios: heterogeneous multiprocessing with satellite kernels. *In : Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. pp. 221–234. – New York, NY, USA, 2009.
26. Potop-Butucaru (D.), Azim (A.) et Fischmeister (S.). – Semantics-preserving implementation of synchronous specifications over dynamic tdma distributed architectures. *In : EMSOFT*, pp. 199–208.
27. Sarkar (A.), Mueller (F.) et Ramaprasad (H.). – Predictable task migration for locked caches in multicore systems. *In : Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*. pp. 131–140. – New York, NY, USA, 2011.
28. Thies (W.) et Karczmarek (M.). – StreamIt: A language for streaming applications. *Compiler Construction*, 2002, pp. 179–196.
29. Wang (Y.-M.), Wang (H.-H.) et Chang (R.-C.). – Clustered affinity scheduling on large-scale numa multiprocessors. *Journal of Systems and Software*, vol. 39, n1, 1997, pp. 61 – 70.
30. Wentzlaff (D.) et Agarwal (A.). – Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, vol. 43, n2, avril 2009, pp. 76–85.
31. Yan (Y.), Society (I. C.), Zhang (X.), Member (S.) et Zhang (Z.). – Cacheminer: A runtime approach to exploit cache locality on smp. *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, 2000, pp. 357–374.
32. Yviquel (H.), Casseau (E.), Wipliez (M.) et Raulet (M.). – Efficient multicore scheduling of dataflow process networks. *In : Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, pp. 198 – 203. – Liban, décembre 2011.