A Monitoring System for Runtime Adaptations of Streaming Applications

Manuel Selva, Lionel Morel, Kevin Marquet, Stephane Frenot

To cite this version:

Manuel Selva, Lionel Morel, Kevin Marquet, Stephane Frenot. A Monitoring System for Runtime Adaptations of Streaming Applications. International Conference on Parallel, Distributed and Network-based Processing, Mar 2015, Turku, Finland. 10.1109/PDP.2015.53 . hal-01229025

HAL Id: hal-01229025

https://hal.inria.fr/hal-01229025

Submitted on 16 Nov 2015

A Monitoring System for Runtime Adaptations of Streaming Applications

Manuel Selva*†, Lionel Morel†, Kevin Marquet†, Stephane Frenot†
*Bull Echirolles, 1 rue de Provence 38342 Echirolles Cedex, France
†Universite de Lyon
Inria INSA-Lyon, CITI
F-69621, France
Email: firstname.lastname@insa-lyon.fr

***Abstract*—Streaming languages are adequate for expressing many applications quite naturally and have been proven to be a good approach for taking advantage of the intrinsic parallelism of modern CPU architectures. While numerous works focus on improving the throughput of streaming programs, we rather focus on satisfying quality-of-service requirements of streaming applications executed along-side non-streaming processes. We monitor *synchronous dataflow* (SDF) programs at runtime both at the application and system levels in order to identify violations of quality-of-service requirements. Our monitoring requires the programmer to provide the expected throughput of its application (*e.g* 25 frames per second for a video decoder), then takes full benefit from the compilation of the SDF graph to detect bottlenecks in this graph and identify causes among processor or memory overloading. It can then be used to perform dynamic adaptations of the applications in order to optimize the use of computing and memory resources.**

I. Introduction

The traditional threading model exposes little parallelism and does not suit the programmer needs to program multi- and many-core architectures. In order to address this issue, one mean is to use a streaming programming paradigm, the advantage being that different levels of parallelism are made explicit in the program.

This paradigm has been proven useful in a large number of domains including networking, video, sound, graphics, cryptographic tools, or digital signal processing. With the advent of mobile computing, such applications are increasingly useful and, consequently, a lot effort has been put in the past decade to maximize the throughput of dataflow applications. However, for a majority of these applications, maximizing the throughput is not what the end user wants. Rather, it is usually not useful to go far beyond a minimum throughput requirement. For instance, if a video decoder satisfies a throughput of 25 frames per second, the goal is reached. In some cases, it can even be counter-productive to run very fast as it can result in energy waste. In this paper, we propose solutions to maintain, at runtime, a throughput close to the requirements for these dataflow applications, while standard applications (*i.e.* non described in a dataflow way) start and stop dynamically.

In order to optimize the use of computing resources, two types of approaches are traditionally investigated depending on the nature of the programming model. First, lots of optimizations have been proposed to take advantage of the Synchronous Dataflow (SDF) model of computation [17] (aka Static Dataflow). In SDF graphs, not only the communication channels – defining which actors communicate together – are explicit, but the communication rates – how many items are read and written each time an actor is executed – are known. This information can be used by the compiler to perform static optimizations. Typically, a balanced distribution of the actors on available processors or cores can be computed [14]. However, this requires to compute average execution times, which is impossible in a context where other, non-dataflow, applications start and stop dynamically.

Second, in the case of dynamic dataflow models of computation [18], actors consumption and production rates are not known statically. Runtime mechanisms need to be developed to maximize the application's throughput [20].

In this paper, we combine SDF static optimizations with properties of dynamic runtimes. We propose to take advantage of information included in static dataflow graphs and to use it at runtime to maintain quality-of-service requirements expressed by the programmer.

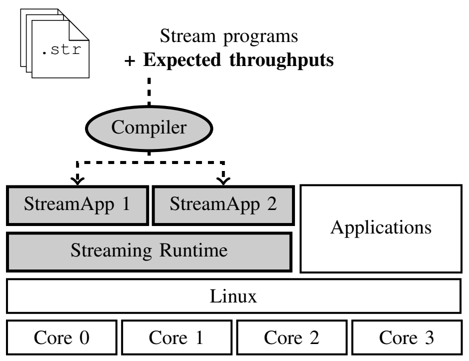


Fig. 1: Overview of the proposal

Our proposal is sketched in Figure 1. The contributions we make are:

- An extension of the StreamIt language [31] allowing a programmer to specify an expected throughput for its applications(s);
- An extension of the StreamIt compiler able to propagate the expected throughput to *each* actor of the graph. This is possible only in SDF programs;

- An *application* monitor able to check at runtime whether the throughput of each streaming application (denoted *StreamApp* in Figure 1) is respected or not;
- A *system* monitor able, in case of non-respect, to identify the origin of the bottleneck: CPU, memory latencies;
- An adaptation mechanism able, with regards to previous observations, to change the placement of actors on the cores/processors, change the placement of FIFOs in memory, and adapt the batching;
- A performance evaluation of the mechanisms described above.

The rest of this paper is organized as follows. Section II provides the reader with necessary background concerning SDF languages, their compilation and runtime support. Section III presents extensions we propose to the StreamIt language, its compiler and runtime. Section IV presents our runtime monitoring that includes both application and system monitoring facilities. Section V presents experimental results to validate our proposal. Finally, Section VI gives an overview of related work while Section VII concludes and gives perspectives.

II. Background

A. SDF

We consider programs to follow the SDF model of computation [17]. An SDF program is a graph of actors communicating only through FIFOs: an actor can *pop* data tokens from its input queues or *push* tokens to its output queues. But it can have no other side effect on its environment (*e.g.* no global variables). In contrast with more general Dataflow Process Networks [18] or Kahn Process Networks [15], the number of tokens read from input queues (resp. written to output queues) of SDF actors is fixed at compile-time. This information leads to static schedulability. Using the static consumption and production rates, a dataflow compiler first computes a *repetition vector*, denoted q in the remainder of this paper. This vector defines the number of times each actor must be executed so that the number of tokens in all the FIFOs of the application falls back to its initial state [17]. From this repetition vector, the compiler computes a *steady-state schedule* or *iteration* that satisfies actors' data dependencies. This schedule consists of a sequence of *components*; each component act^x indicating to execute actor act x times before the next component. The SDF example of Figure 2 would have the following repetition vector q and $s1$ and $s2$ as two valid steady-state schedules:

$$q = (1, 1, 1, 3, 4, 1)\ ;\ \text{for actors } (A, B, C, D, E, F)$$
$$s1 = A^1B^1C^1D^3E^4F^1, s2 = A^1B^1C^1D^1E^2D^2E^2F^1$$

In the remainder of this paper, we use the following notations : $q(act)$ denotes the entry corresponding to actor act in the repetition vector; $outRate(act, out)$ denotes the number of tokens that actor act produces on each activation on output FIFO out; $activ(comp)$ denotes the number of activation of a component $comp$ in the steady-state schedule, *i.e.* $activ(act^x) = x$. We consider only consistent SDF graphs as defined by Lee [17].

If several cores (or CPUs) are available, the compiler is likely to balance the execution of the components on different

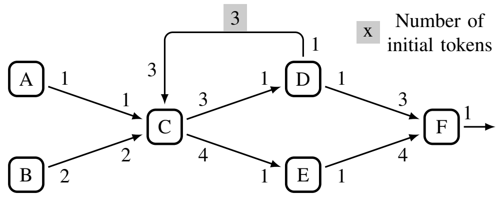


Fig. 2: Example SDF Graph.

cores. In this case, at runtime, components synchronize either on FIFOs or on a barrier as described in Section II-C. Also, note that actors can be executed in parallel by exploiting the inherent pipeline parallelism provided by the dataflow model: in our example of Figure 2 once actor A has produced a first token, C can start working with this token while A starts working on producing a second token.

B. Multi and many-core target platforms

In this work we target multi- and many-core processors with shared memory. These architectures are today standard in markets ranging from mobile systems to servers. Alongside the programability problem mentionned in the introduction, the complexity of these architectures, in particular of their memory hierarchy introduces tremendous difficulties for the evaluation of programs performances. As the number of cores keeps increasing, hardware providers are more and more using Non Uniform Memory Access (NUMA) architectures to alleviate the memory wall. Figure 3 shows the global architecture of an Intel Xeon X5650 dual socket platform released in 2011, that we use in our experiments.

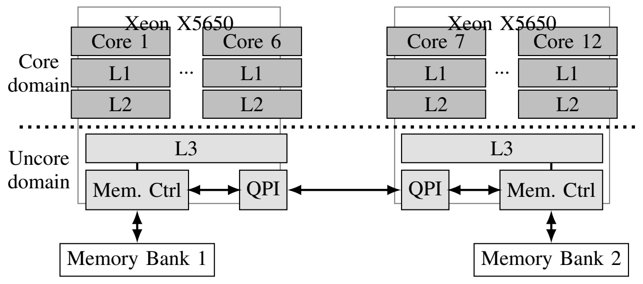


Fig. 3: Commodity NUMA hardware

On such NUMA hardware, improving the performance of programs is even more difficult. In the architecture of fig. 3, for a program that executes on core 1, accessing data stored in memory bank 2 is typically 25% slower than accessing data stored in memory bank 1. Additional care must thus be taken by the software to limit remote memory accesses [22], [9] in particular in the case of memory-intensive programs.

C. Execution of SDF programs over thread-based APIs

We assume that the underlying operating system provides a threading programming model. In this context, an efficient way of executing SDF programs, as done in StreamIt [14] consists of assigning one thread to each core and map actors to threads. As shown in Figure 4, the compiler initially balances actors among t threads where t is the number of cores of the targeted

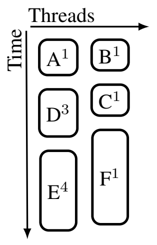


Fig. 4: Static load balancing with estimated execution times.

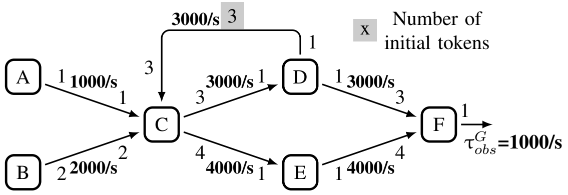


Fig. 5: τ^G_{exp} is given by the programmer; the compiler propagates it to compute $\tau^L_{exp}(act, out)$ for all channels and attaches the information to the application for the runtime.

platform. Here we have $t = 2$ on the example for readability. The runtime engine pins each thread on a specific core. Each one has an associated list of pointers to the execution function of each actor. A thread iterates over this list to execute the actors and then synchronizes on a barrier. Once all threads have completed one steady-state iteration, the runtime layer can safely start the next one, without loosing any data. FIFOs are implemented as fixed size arrays shared by all the threads of the application.

III. Language, compiler and runtime support

This section describes how a global expected throughput can be added to a stream application, and how the compiler can compute an expected throughput for each actor. Finally, it describes our extension to the aforementioned runtime to include monitoring and adaptation capabilities.

A. Language support

In the remainder, we consider SDF applications to have exactly one output flow. This is not necessarily the case in real-life applications. For programs with multiple output flows, the expected throughputs expressed on different output flows need to be consistent with each other. Due to the static nature of SDF progams, this is easily verified at compile-time.

Our proposal applies to all SDF languages, provided that it is possible to specify an expected throughput. In our case, we extend the StreamIt language. The developer specifies a *global expected throughput* for his application. This throughput for an application *app* is noted $\tau^G_{exp}(app)$. It is a simple integer specifying for the output edge of the graph, the number of tokens that must be produced on this edge per second.

B. Throughput propagation

From the throughput given by the programmer, the compiler computes, for each actor, the rate at which this actor must output items. Such a rate is called *local expected throughput*. Figure 5 illustrates this on our toy example of Figure 2. In the remainder of this article the *local expected throughput* of an actor act on its output arc out is noted $\tau^L_{exp}(act, out)$.

The propagation done by the compiler for an application *app* consists of computing the frequency at which all the actors have to be fired in order to satisfy the global throughput. For a given actor act, this *required frequency* is noted $f_r(act)$ and the local expected throughput is computed using equation 1, below. Equation 2 defines $f_r(sink)$, $sink$ being the actor from which starts the output channel e on which $\tau^G_{exp}(app)$ is defined.

$$\tau^L_{exp}(act, out) = f_r(act) * outRate(act, out) \tag{1}$$

$$f_r(sink) = \frac{\tau^G_{exp}(app)}{outRate(sink, e)} \tag{2}$$

From this, the required frequency of each actor act can be computed using the number of activation of act to the number of activation of $sink$ in the repetition vector as shown by equation 3.

$$f_r(act) = f_r(sink) * \frac{q(act)}{q(sink)} \tag{3}$$

From the activation frequency required for one actor, the compiler easily computes the frequency required for one component $comp$ of the steady-state schedule using equation 4.

$$f_r(comp) = f_r(act) * \frac{q(act)}{activ(comp)} \tag{4}$$

In the execution model we use, each thread executes several actors and synchronizes with others with a steady-state barrier. As a consequence, $\tau^L_{exp}(act, out)$ values can't be used as is by the runtime. Indeed, an actor being too slow will slow down the entire application, leading to the impossibility to identify the bottleneck actors by their throughput alone. Therefore, using the local expected throughput of each actor, we compute its *expected execution time*, noted $\kappa_{exp}(act)$ using equation 5. This expected execution time will be used at runtime to identify actors that are too slow.

$$\kappa_{exp}(act) = \frac{outRate(act, out)}{\tau^L_{exp}(act, out)} \tag{5}$$

C. Runtime Support

We extend the runtime of Section II-C so that it handles several streamit applications. The information obtained by the throughput propagation of Section III-B is attached to each application, and includes its global expected throughput and a local expected throughput for each actor.

Figure 6 shows an overview of our runtime. The streaming applications execute on top of a mainstream operating system (Linux in this instance) alongside non-dataflow applications. Streaming applications are compiled to $n - 1$ threads where n is the number of cores of the hardware. Two components, detailed in the next section are part of the runtime. The

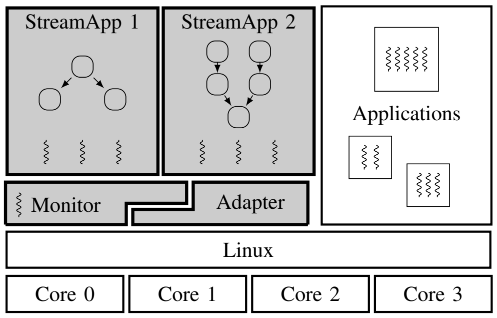


Fig. 6: Runtime Overview

Monitor running in a dedicated thread pinned on the first core keeps track of how much resources are allocated to dataflow applications and how these resources are used by the actors. We extend the StreamIt compiler in order to generate monitoring code that will be activated on demand by the runtime. The *Adapter* implements adaptation heuristics based on monitoring results. It is called on-demanded from the *Monitor* thread and also executes on the dedicated core. To support the *Adapter*, the runtime layer provides a way to suspend and resume dataflow applications either individually or globally. It also implements migration mechanisms. To migrate actors between cores, function pointers are moved between threads. To migrate FIFOs, we rely on mechanisms of the underlying operating system providing a way to migrate memory pages between memory banks.

IV. Runtime monitoring

The runtime monitor is responsible for 1) determining if throughput constraints are satisfied or not; 2) identifying which actors in the graph are the bottlenecks and 3) distinguishing between CPU and memory contentions. This section describes conditions under which the runtime layer starts monitoring actors' activity and how it identifies bottleneck actors and impacted resources. Figure 7 gives an overview of the monitoring process for one dataflow application.

This monitoring process is carried out by a specific thread running on a dedicated core as shown on Figure 6. At application startup time, monitoring stage 1 is automatically activated. It consists of periodically checking the value of $\tau^G_{obs}(app)$ as described in Section IV-A. When the throughput reaches the local monitoring threshold $\tau^G_{exp}(app) + \delta$ the monitoring thread enters in stage 2 described in Section IV-B by activating local monitoring for a given duration called the local monitoring period. During this period, each dataflow threads records information about its actors execution times. At the end of the local monitoring period, depending on the existence of bottleneck actors or not, the monitoring thread either enters stage 3 where it performs memory monitoring as described in Section IV-C, or stage 4 where it performs CPU load monitoring as described in Section IV-D. Both stages have a monitoring period, and at the end of this period the monitoring thread invokes adaptation mechanisms described in Section IV-E.

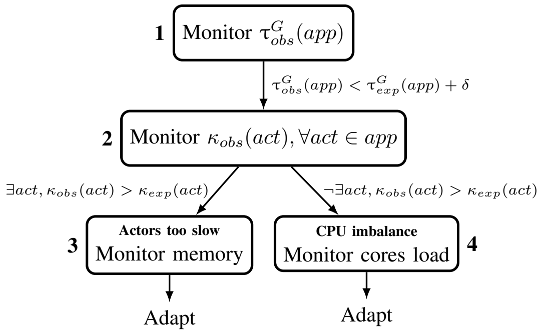

1
Monitor $\tau^G_{obs}(app)$
$\tau^G_{obs}(app) < \tau^G_{exp}(app) + \delta$
2
Monitor $\kappa_{obs}(act), \forall act \in app$
$\exists act, \kappa_{obs}(act) > \kappa_{exp}(act)$
$\neg\exists act, \kappa_{obs}(act) > \kappa_{exp}(act)$
3
Actors too slow
Monitor memory
4
CPU imbalance
Monitor cores load
Adapt
Adapt

Fig. 7: Monitoring stages overview

A. Monitoring the global throughput

In order to check at runtime if the throughput expected for an application is obtained, monitoring code is generated in the sink actor. This code consists of a mere counter increment.

By checking periodically the value of this counter, the runtime layer computes the *global observed throughput* of the application noted $\tau^G_{obs}(app)$. If it is too low (this criterion is detailed later), some more precise monitoring is performed.

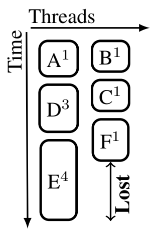
Overhead from this monitoring system comes from two places. Incrementing a counter each time data is written on the channel where τ^G_{obs} is expressed is negligible. The overhead of the simple arithmetic used to compute τ^G_{obs} on regular interval using the counter is also negligible since the monitoring thread is executing on a separate core.

B. Monitoring local execution times

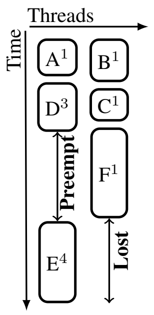
The compiler generates some code that computes the *observed execution time* noted $\kappa_{obs}(act)$ for all the actors of the graph. This time represents the execution time of actors only. It excludes preemption time. We rely on timing system calls provided by the underlying operating system for these measurements. Timing measurements are performed before and after the loop executing each component. To compute the average $\kappa_{obs}(act)$, we divide the measured time by the number of times the actor act has been fired in the component. Measuring outside the loop allows to reduce the number of system calls and thus the overhead introduced by timing measurements. Section V evaluates the overhead of these system calls on our test platform.

As shown on Figure 7, the runtime activates the local monitoring only when $\tau^G_{obs}(app)$ is getting close to $\tau^G_{exp}(app)$ to identify eventual bottleneck actors. Deciding when the local monitoring should be triggered (*i.e.* the value of δ in Figure 7) is discussed in Section V.

At the end of the local monitoring period, the $\kappa_{obs}(act)$ for each actor is compared to its $\kappa_{exp}(act)$ to tag the bottleneck actors. When all actors respect their expected execution time, that means the dataflow threads are badly balanced and the *Monitor* must identify why. This is discussed in Section IV-D. Otherwise (there is at least one actor with $\kappa_{obs}(act)$ greater than $\kappa_{exp}(act)$), the actors being too slow are known, but



(a) Result with measured real execution times (F is faster than estimated).



(b) Result with preemption.

Fig. 8: Limitations of static scheduling on the SDF graph of Figure 5 executed with steady-state schedule $s1$ on a dual core.

the *Monitor* also needs to identify why. See Section IV-C for details on the latter.

C. Bottleneck actors

In the case where we have identified the bottleneck actors (bottom left branch of Figure 7), we must identify why these actors are too slow. On the homogeneous CPU architectures we target, we may only be able to speed up actors by reducing memory access times. As stated in Section II-C, communication is implemented over shared memory. As a consequence, many of the memory accesses performed by actors correspond to inter actors communications. Memory requests also include accesses to actors internal variables required during actor firing.

To identify potential memory bottlenecks, our runtime provides low-level memory monitoring based on hardware performance monitoring unit (PMU). This hardware, present on Intel, AMD and ARM processors, provides low-level information about the usage of hardware resources with a very low overhead. Using the PMU in *counting mode*, we identify if one or more memory controllers and one or more memory links are overloaded. If we find that one of the memory controllers used by a bottleneck actor is overloaded, we know that communication and local memory accesses time could be reduced by alleviating the load of the controller. Using the performance monitoring hardware in sampling mode along with the knowledge of the dataflow graph, the runtime is able to establish a memory profile for each actor of the graph. This memory profile can be used by a memory adaptation heuristic to alleviate memory controllers and memory links overload.

D. Dataflow threads imbalance

When the global throughput is too low and all actors respect their execution time (bottom right branch of Figure 7), we know that we are facing at best a core imbalance, at worst a globally overloaded system. Two different things may cause an imbalance. First, a wrong estimation of actors execution time used during compilation may lead to imbalance between cores, resulting in cores more loaded than others. The under-loaded cores will thus loose time in the steady-state barrier as depicted on Figure 8a.

The second kind of situations leading to cores imbalance are scenarii where the dataflow applications share core(s) with other applications taking resources that the compiler could not have been aware of. This situation is depicted on Figure 8b. On the first core, dataflow actors are preempted resulting in a longer steady state. As consequence the two dataflow threads are imbalanced and available processor time is lost on the second core where the second dataflow thread is pinned.

In both wrong estimation and preemption cases we need to identify the cores which are overloaded and the ones which are under-loaded. For this purpose we again rely on the underlying operating system. On Linux, we read the $/proc/stat$ virtual file to get this information at the start and at the end of the monitoring period.

E. Towards Runtime adaptation heuristics

Using runtime mechanisms allowing to migrate actors between threads pinned to cores, and to migrate FIFOs between memory nodes we are able to implement adaptations heuristics. For now, we only implemented a simple heuristic consisting of migrating actors from the most overloaded core to the one with the lowest load in the context of a single dataflow application running along with non-dataflow applications.

Algorithm 1 CPU load balancing

```
1: ovld, ovldCore, unld, unldCore = getLoad()
2: toMove = (ovld - unld) ÷ 2, moved = 0
3: act = nextHeaviestActor(ovldCore)
4: while moved < toMove and act != null do
5:    if moved + $\kappa_{obs}(act)$ ≤ toMove then
6:       move(act, ovldCore, unldCore)
7:       move += $\kappa_{obs}(act)$
8:    end if
9:    act = nextHeaviestActor(ovldCore)
10: end while
```

Algorithm 1 shows this runtime heuristic. It moves actors between the core with the highest CPU load to the one with the lowest CPU load using actors measured execution times to minimize the difference of work amount between theses two cores. The `getLoad()` function is the result of cpu load monitoring stage described in Section IV-D and the `nextHeaviestActor(int core)` function returns actors in decreasing execution time order (i.e. the actor with the largest κ_{obs} value is returned first).

Designing more complex heuristics benefiting from our system-level monitoring is left to future work.

V. Experimental results

We now present the results of our dataflow framework under different scenarii illustrating the different kinds of bottleneck dataflow applications may face. We use StreamIt applications from [30]. The experiments were conducted on a workstation running Linux 3.11 on top of two Intel Westmere-EP processors. Details are shown on table I.

Hyperthreading is disabled in all the benchmarks resulting in a total of 12 cores visible by the software. StreamIt applications used in the experiment are compiled to 10 threads. We keep core 0 available for running all the system's processes

Processors	Westmere-EP: 2x Intel Xeon5650 (hexa core)
Core frequency	2.66 GHz
L1d/L1i caches size	32 KiB / 32 KiB
L2/L3 caches size	256 KiB / 12 MiB
Memory	6 x 8 GiB DDR3-1333

TABLE I: System configuration

and the benchmark scripts in order to minimize the operating system impact on the experimental results. As mentioned in Section IV, the monitoring thread is running on a dedicated core (core 1).

A. Scenario 1: Reaction to preemption by other applications

In this scenario, we run one StreamIt application with an attached throughput constraint along another non-dataflow application introduced in the system. For the applications used in this scenario, $\kappa_{obs}(act)$ values are smaller than $\kappa_{exp}(act)$ values for each actor, i.e. we are not facing bottleneck actors. In this scenario, we first run the application alone on the system and save the value of $\tau^G_{obs}(app)$ at an arbitrary frequency of 20 Hz. Because the monitoring thread is running on a dedicated core, this frequency does not impact streaming applications performances but only changes the time before the runtime may react. Then, we run a non-dataflow application, in this case a compute intensive C benchmark, on core 5 for 10 seconds along with the dataflow application and see how our runtime reacts. The value of the δ parameter introduced in Section IV-B is set to 0. The study of how to compute this value is left to future work. We experimentally evaluated the value of the load balancing monitoring periods described in Section IV to 10 microseconds: with this value we are able to correctly identify the overloaded core.

Figures 9 and 10 show the evolution of $\tau^G_{obs}(app)$ in tokens per microsecond for two applications (fft and $filterbank$) over time along with the $\tau^G_{exp}(app)$ values and the perturbation introduction. fft and $filterbank$ being micro benchmarks, it's difficult to define realistic $\tau^G_{exp}(app)$ values. As a consequence these values have been choosen arbitrarily. Repeating the scenario several times leads to very small throughput variations but clearly shows the same phases. On both plots, we clearly identify three different phases. First, the application is launched without any perturbation showing a stable state. Even if we isolated system processes on core 0, some work is still done by the kernel on other cores resulting in minor variations on the throughput. Then for we clearly see a throughput drop when perturbation is introduced. Finally, after balancing we see that the throughput is reaching back an acceptable value. For both applications, because the perturbation introduced is CPU intensive, all the actors of the graph are moved away from the perturbed core. For $filterbank$, it results in a new throughput slightly lower than the initial throughput obtained without any perturbation. This is explained by the fact that this application is compute intensive with few communication between actors and scales well with the number of core. After rebalancing, it executes on 9 cores whereas it was previously executed on 10 cores, which makes a difference in terms of computations for this application. For fft, the new throughput is higher than the original one. This is also explained by the scaling property of the application. fft steady-state time is small and this application has a lot of communication. The application is thus more efficient when mapped to 9 cores than to 10 cores.

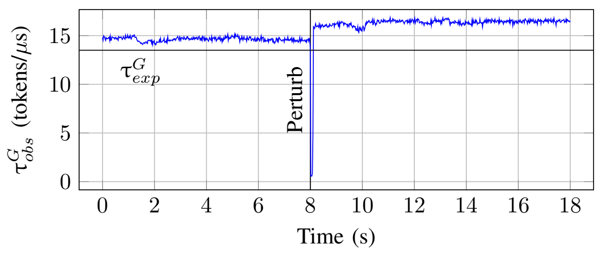


Fig. 9: Scenario 1 - fft

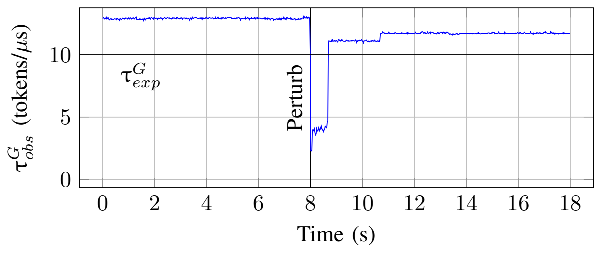


Fig. 10: Scenario 1 - filterbank

B. Scenario 2: Identification of bottleneck actors

In this scenario, we highlight the effectiveness of bottleneck actors detection mechanisms. For this purpose we changed the $filterbank$ application used in scenario 1 in order to increase the execution time for the actor called *combine* with artificial computing code. In this case, as soon as the application is started, τ^G_{obs} is smaller than τ^G_{exp} and the local monitoring is started. Table II shows the average execution times for 3 actors at the end of the local monitoring period. The local monitoring clearly identifies the modified *combine* actor as a bottleneck actor: κ_{obs} is almost five times longer than κ_{exp}.

Actor	$\kappa_{exp}(\mu s)$	$\kappa_{obs}(\mu s)$
fir_filter	780	12.231
combine	780	**3508.678**
down_sample	97.500	16.078

TABLE II: Actors execution times

Once *combine* is identified as a bottleneck actor we start memory monitoring as described in Section IV-C. In this fictive scenario, our memory monitoring subsystem does not detect any memory bottleneck. The monitored memory throughput on the two memory controllers of platform is below 1 gigabyte per second whereas the theoretical maximum throughput is 32 gigabytes per second. The runtime then concludes that *combine* execution time can't be reduced using memory adaptations and is only able to report the bottleneck.

On our benchmarking platform, using combination of Streamit benchmarks we were not able to saturate the memory system. As discussed in the conclusion, we plan to run our memory monitoring system with real world dataflow applications on platform where memory latency vary more and where we are able to overload memory controllers.

C. Runtime monitoring and adaptation overhead

We now evaluate the overhead of the proposed mechanisms.

a) τ^G_{obs}*:* As stated in Section IV-A the overhead induced by the global monitoring system is negligible because we only add a counter increment to the original application and because the monitoring thread is executing on a separate core.

b) κ_{obs}*:* To evaluate the overhead of local execution time monitoring we first evaluated the cost of the system call `clock_gettime(CLOCK_THREAD_CPUTIME_ID)` that we use to monitor actors execution time excluding preemption. On our platform, the system call average execution time is 170 ns. The overhead on our application is thus in theory $2*170$ multiplied by the number of components in the steady state chosen by the compiler (the system call is made at the beginning and at the end of the component's execution). Table III shows the average experimental overhead over 20 runs computed by monitoring τ^G_{obs} with and without local monitoring activated.

App.	τ^G_{obs}(tk/μs)	τ^G_{obs} with local mon(tk/μs)	Slowdown(%)
audiobeam	2.04	2.44	-19.80
fft	14.85	16.24	-9.36
filterbank	12.69	12.48	3.65
fmradio	3.82	3.88	-1.52

TABLE III: Local monitoring overhead

Surprisingly, for 3 of the 4 applications shown on this table, activating local monitoring increases throughput. This is explained by the very short length of the steady-state (the synchronization time is greater than the steady state time for $fmradio$ and $audiobeam$ and equal to the steady state time for fft). These applications spend a lot of time synchronizing on the steady-state barrier. Adding time measurement thus affects the synchronization time positively. For the $filterbank$ application, the steady state is longer and is about 10 times the synchronization time spent on the barrier. In this case, adding time measurement increase steady state time and has a negative impact on the throughput reaching almost 4%.

c) Rebalancing: Because we have to suspend actors execution while doing the adaptation, applications are not able to output any tokens during rebalancing. As a consequence, we have to minimize the time required to provide a new mapping of the application. Our simple first rebalancing algorithm has a complexity of $O(n)$ where n is the number of actors. In our two examples of scenario 1, the average rebalacing times over 20 runs are 489 and 474 microseconds for fft and $filterbank$ respectively. We consider this time acceptable if the adaptation heuristic is not invoked too often.

VI. Related Works

There is substantial work in the literature relating to the efficient execution of SDF programs. A large portion of it concentrates on static scheduling and mapping of dataflow programs on multi-core architectures, especially for the StreamIt language [14], [13]. Mostly, these works make the assumption that the application considered will run standalone on the target platform. They also use user-provided execution times for balancing actors. In this way they are dependent on the precision of the information provided by the programmer and cannot cope with variations in the availability of resources.

The work of [29] proposes a runtime for StreamIt programs able to balance actors on the available cores in order to maximize throughput. To do so, it measures actors execution time to adapt the static placement by taking runtime conditions in considerations. Our proposal extends [29] to identify individual bottleneck actors and also to establish whether the slowdown comes from CPU or memory contentions which will eventually permit more sophisticated adaptation heuristics.

A technique to identify bottleneck actors is proposed in [7]. However its adaptation possibilities are defined by particular patterns specified statically. [6] extends this approach by allowing for dynamic splitting of actors. The main limitation of both approaches lie in the fact that programs need to be acyclic. This is rather unrealistic since streaming applications often have feedback loops. Our proposition is valid on any SDF program. Moreover, both papers limit their bottleneck identification to execution times of actors.

Other works [3], [2] propose a runtime for dataflow programs that is able to adapt applications in order to maintain pre-defined quality-of-service levels, on a platform presenting varying execution conditions. Adaptation schemes are however application-specific, i.e. the application designer needs to specify which actor can be split and which can be skipped.

Compilation strategies taking minimum throughput requirements into account have also been proposed. Various propositions can be found in [4], [5], [8], [12], [24], [27] or [28]. These and all static optimization decisions found in the literature are complementary to our runtime proposals.

To our knowledge, our proposal is the first to complement application-level monitoring facilities (throughput in terms of tokens per time unit) with system-level (especially memory-oriented) monitoring to distinguish between CPU and memory contentions for bottleneck detection of dataflow programs.

VII. Conclusion

We have investigated how the information contained in Synchronous Dataflow graphs can be used to maintain, at runtime, some quality-of-service requirements. We have proposed a two level runtime monitoring approach which is able to 1) identify individual actors responsible for slowing down a dataflow application; and 2) pinpoint reasons of the observed loss in performances, be they due to processor or memory congestion.

The approach has been integrated to the StreamIt compiler and runtime. It has been experimented on a 12 cores platform, using several of the StreamIt benchmark programs. For all, it has shown its capabilities to identify the impact of system-wide perturbations on the managed application and the actual contentions at the system level, with a very low overhead.

This lays the path for several important future works. First, we plan to experiment our monitoring approach over platforms for which memory latency vary more and for which the placement decisions will have a greater impact. This will be a strong complement to existing compilation strategies that already take the underlying memory hierarchy into account, eg [26], [23], [11], [10], [1].

As shown in Section IV, our monitoring mechanisms have an impact on the performance of applications. One way to reduce it would be to integrate these mechanisms into the operating system's kernel in order to remove system calls overheads. It will also allow future adaptation heuristics to be implemented with information about non dataflow applications running concurently to our dataflow applications.

Finally, and more importantly, we plan to apply our approach to streaming languages that implement more general dataflow models. Indeed, SDF can not be used to model many real-life applications for which input and output rates of actors vary dynamically, as a function of input values or actors' internal state. In the near future, we will therefor investigate the inclusion of our monitoring in the runtime of the RVC-Cal language, benefiting from both the maturity of the associated tools and the existence of realistic case studies. In this case, static information can not be used and the runtime needs to learn expected throughput, e.g. during system warm-up. We will also confront our approach to various runtimes that exist for dynamic dataflow [19], [20], [25], [21].

References

[1] K. Agrawal, J. T. Fineman, J. Krage, C. E. Leiserson, and S. Toledo. Cache-conscious scheduling of streaming applications. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 236–245. ACM, 2012.

[2] A. Albers and P. de With. Task complexity analysis and qos management for mapping dynamic video-processing tasks on a multi-core platform. *Journal of Real-Time Image Processing*, 7(3):185–202, 2012.

[3] R. Albers, E. Suijs, and P. de With. Qos management of dynamic video tasks by task splitting and skipping. In *IEEE/ACM/IFIP 7th Workshop on Embedded Systems for Real-Time Multimedia, ESTIMedia.*, pages 64–69. IEEE, 2009.

[4] A. Bonfietti, L. Benini, M. Lombardi, and M. Milano. An efficient and complete approach for throughput-maximal sdf allocation and scheduling on multi-core platforms. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition, DATE*, pages 897–902. IEEE, 2010.

[5] S. Carpov, L. Cudennec, and R. Sirdey. Throughput constrained parallelism reduction in cyclo-static dataflow applications. *Procedia Computer Science*, 18:30–39, 2013.

[6] Y. Choi, C.-H. Li, D. D. Silva, A. Bivens, and E. Schenfeld. Adaptive task duplication using on-line bottleneck detection for streaming applications. In *Proceedings of the 9th conference on Computing Frontiers, CF*, pages 163–172. ACM, 2012.

[7] R. L. Collins and L. P. Carloni. Flexible filters: load balancing through backpressure for stream programs. In *Proceedings of the seventh ACM international conference on Embedded software, EMSOFT*, pages 205–214. ACM, 2009.

[8] M. Damavandpeyma, S. Stuijk, M. Geilen, T. Basten, and H. Corporaal. Parametric throughput analysis of scenario-aware dataflow graphs. In *Proceedings of the IEEE 30th International Conference on Computer Design, ICCD*, pages 219–226, 2012.

[9] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 381–394, New York, NY, USA, 2013. ACM.

[10] S. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. Profile-guided deployment of stream programs on multicores. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems, LCTES*, pages 79–88. ACM, 2012.

[11] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. Orchestration by approximation: mapping stream programs onto multicore architectures. *ACM SIGPLAN Notices*, 46(3):357–368, 2011.

[12] A. Ghamarian, M. Geilen, T. Basten, and S. Stuijk. Parametric throughput analysis of synchronous data flow graphs. In *Proceedings of Design, Automation and Test in Europe, 2008. DATE '08*, pages 116–121, 2008.

[13] M. I. Gordon. *Compiler techniques for scalable performance of stream programs on multicore architectures*. PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, 2010.

[14] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 151–162. ACM, 2006.

[15] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress*. North-Holland, 1974.

[16] R. Lachaize, B. Lepers, and V. Quéma. Memprof: A memory profiler for numa multicore systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 5–5, Berkeley, CA, USA, 2012. USENIX Association.

[17] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[18] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[19] H. Lee, W. Che, and K. Chatha. Dynamic scheduling of stream programs on embedded multi-core processors. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES+ISSS*, pages 93–102. ACM, 2012.

[20] C. Min and Y. I. Eom. Danbi: dynamic scheduling of irregular stream programs for many-core systems. In *Proceedings of the 22nd international conference on Parallel Architectures and Compilation Techniques*, PACT, pages 189–200. IEEE Press, 2013.

[21] C. Miranda, A. Pop, P. Dumont, A. Cohen, and M. Duranton. Erbium: A deterministic, concurrent intermediate representation to map dataflow tasks to scalable, persistent streaming processes. In *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES, pages 11–20, New York, NY, USA, 2010. ACM.

[22] D. Molka, D. Hackenberg, R. Schöne, and M. S. Müller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, PACT*, pages 261–270, 2009.

[23] A. Moonen, M. Bekooij, R. van den Berg, and J. van Meerbergen. Cache aware mapping of streaming applications on a multiprocessor system-on-chip. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 300–305, March 2008.

[24] P. Poplavko, M. Geilen, and T. Basten. Predicting the throughput of multiprocessor applications under dynamic workload. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 282–288. IEEE, 2010.

[25] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis. Dynamic fine-grain scheduling of pipeline parallelism. In *PACT*, pages 22–32, 2011.

[26] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache aware optimization of stream programs. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES)*, volume 40, pages 115–126. ACM, 2005.

[27] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the 44th ACM/IEEE Design Automation Conference*, pages 777–782. IEEE, 2007.

[28] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Proceedings of the International Conference on Embedded Computer Systems, SAMOS*, pages 404–411. IEEE, 2011.

[29] C. Tan. A hybrid static/dynamic approach to scheduling stream programs. Master's thesis, Massachusetts Institute of Technology, 2009.

[30] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 365–376, New York, NY, USA, 2010. ACM.

[31] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction, CC*, Grenoble, France, Apr 2002.

[32] F. Zheng, C. Venkatramani, R. Wagle, and K. Schwan. Cache topology aware mapping of stream processing applications onto cmps. *2013 IEEE 33rd International Conference on Distributed Computing Systems*, 0:52–61, 2013.