Comparing the StreamIt and ΣC Languages for Manycore Processors

Xuan Khanh Do, Stephane Louise
CEA, LIST
91191 Gif-sur-Yvette Cedex, France
Email: xuankhanh.do@cea.fr, stephane.louise@cea.fr

Albert Cohen
Inria, Parkas Team
24 rue d'Ulm, 75005 Paris, France
Email: albert.cohen@inria.fr

***Abstract*—Embedded many-core systems offering thousands of cores should be available in the near future. Stream programming is a particular instance of data-flow programming where computations are expressed as the data-driven execution of repetitive "filters" on data streams. Stream programming fits these many-core systems' requirements in terms of parallelism, functional determinism, and local data reuse. Statically or semi-dynamically scheduled stream languages like *e.g.* StreamIt and ΣC can generate very efficient parallel code, but have strict limitations with respect to the expression of dynamic computational tasks, context-dependent modes of operation, and dynamic memory management. This paper compares two state-of-the-art stream languages, StreamIt and ΣC, with the aim of better understanding their strengths and weaknesses, and finding a way to improve them. We also propose an automatic conversion method and tool to transform between these two languages. This tool allows to port and evaluate the classical StreamIt benchmarks on Kalray's MPPA, a real-world many-core processor representative of tomorrow's embedded many-core chips. We conclude with propositions for the evolution of stream-programming models.**

I. Introduction

The emergence of chip multiprocessor architectures integrating hundreds or thousands of processing units in a single chip pushes for profound changes in the expression of scalable parallelism up to millions of concurrent tasks and its automatic exploitation on complex, often distributed memory hierarchies. It is generally accepted that such many-core processors cannot be efficiently exploited without a careful selection of scalable algorithms and the introduction of new programming models. But programmers currently lack a clear path to navigate the landscape of many-core programming models and tools, and programming tools are unfortunately lagging behind the fast paced architectural evolutions.

Indeed, today's most mature parallel programming models, such as OpenMP, do not really fit the architectural constraints of many-core systems, and embedded many-core architectures in particular [12]. Data-flow programming address some of these issues, meeting the requirements in terms of scalable task parallelism, functional determinism, and temporal and spatial data reuse in these systems. Among the class of data-flow languages, stream programming languages provide a natural representations of streams, offering a portable and productive way to deploy complex task- and data-parallel pipelines [16]. Among these, statically or semi-dynamically scheduled streaming languages offer additional guarantees that can be leveraged at compilation time to map a parallel application onto a complex architecture [11]. These streaming languages are particularly well suited for the emerging class of grid-based architectures that are well-suited for stream processing [20]. Perhaps the primary appeal of C is that it provides a "common low-level yet portable abstraction" for von-Neumann architectures. This no longer holds for many-core, grid-based architectures, partly due to their non-globally addressable memories. Stream programming concepts are serious contenders to play the front roles in the quest for a more suitable portable abstraction, as witnessed by the maturation of research languages like StreamIt [20] into more production-ready offerings like ΣC [8].

Unfortunately, these streaming languages are often too static to meet the needs of emerging embedded applications, such as context- and data-dependent dynamic adaptation. In this paper, we are interested in four major classes of embedded applications: *Compression/Depression* of video, *Event Monitoring* such as automated surveillance and data aggregation, *Advanced Networking* including software-defined radio and data-dependent routing, *Parsing/Extraction* involving tokenization and context sensitive data analysis [17]. These applications expose a fundamentally dynamic behavior, although only a part of the data-flow is dynamic. For example, a network monitor has to identify different security violations by using only a static rate pattern matcher. An MPEG decoder routes I-frames and P-frames along different paths, but the operators on these paths all have static rates. Based on this observation, we re-evaluate the Model of Computation (MoC) of the stream programming languages using static graphs and its adequation with the requirements of modern embedded applications. In particular, we conduct a systematic comparison between StreamIt and its more recent, industrial counterpart ΣC. This comparison is based on our experience of developing tools to program the Kalray MPPA many-core platform. The weaknesses and strengths of these streaming languages are analyzed to motivate concrete propositions for language evolution. In summary, this paper makes the following contributions:

- A survey and comparison of the programming model of StreamIt and ΣC, in light of the requirements of modern embedded applications running on many-core systems.
- A method and tool to transform StreamIt it programs to ΣC.
- A discussion and concrete propositions for the evolution of streaming languages using static graphs.

The rest of this paper is organized as follows. Section II provides an overview of the related work. In Section III and Section IV, we describe the model of computation of StreamIt and ΣC, to support the comparison of the two languages in Section V. The translation between these languages is presented in Section VI, before a discussion about language improvements in Section VII and conclusions in Section VIII.

II. Related Work

The principal motivation for research into stream programming or data-flow models [11] comes from the limitations of the von-Neumann architecture to exploit massive amounts of fine grained parallelism [16]. Early data-flow architectures fight the von-Neumann bottlenecks by replacing the program counter with a data-driven execution model (execution of instructions as soon as there is enough data) [21]. More recent data-flow architectures rely on the same principles but execute sequences of instructions, or data-flow threads in parallel, instead of single instructions. While many data-flow programming languages have been proposed to match the needs of such architectures [9], no one provides enough information for a compiler to restructure the data-flow program and adapt it to the exact number of processors, local memories, and communication resources of a target architecture. This makes classical data-flow languages unsuitable for embedded applications, and in particular for real-time ones. Another limitation of of most data-flow languages is the lack of explicit support for stream computing. Haskell and its parallel instances may natively operate on unbounded streams, allowing for lazy and concurrent definition, sampling, and operations on streams, but Haskell suffers from the same lack of statically exploitable information for embedded applications, and it also notoriously suffers from performance issues. One important exception is SISAL [13], where streams have been introduced together with single-assignment arrays. Optimization on arrays and streams such as chaining or fusion have been proposed, as well as copy elimination and in-place operations. But SISAL lacks support for unbounded streams and, again, the level of statically exploitable information needed to adapt a streaming graph to a specific parallel architecture.

StreamIt [20] provides an advanced compiler for streaming applications by performing stream-specific optimization, matching the performance achievable by an expert. The version 2.1 of this language supports some forms of *dynamic rate*. It relaxes the constraint that the exact number of inputs and outputs be known at compile time. However, higher expressiveness comes with performance overheads, impacting the compiler's ability to effectively optimize and map the application, and increasing the complexity of the run-time execution environment. In this case, the StreamIt compiler cannot create a fully static schedule for the streaming application to minimize data copies, memory allocation, and scheduling overhead [17]. ΣC is a programming model and language which ensures programmability and efficiency on many-core processors [8]. It is designed as an extension to C and has the advantage to provide familiarity to embedded developers and the use of underlying C compilation toolchain. Nevertheless, one of the disadvantage of this language is the difficulty to reconfigure the static data-flow graph. The static nature of the data-flow graph is essential to the four main passes and algorithms of the compilation flow. Such limitations have been analyzed in detail on a wireless network application where dynamic agents need to be inserted to tolerate high noise on an input channel, cleaning up the signal [12].

These restrictions led to the quest for a compromise between the language expressive power and the compiler's mapping capabilities. Louise [12] studied the future challenges of embedded many-core processors and made some propositions for the evolution of streaming languages using static graphs. At the other end of the spectrum, focusing on expressive power, Pop and Cohen [16] present a data-flow extension of OpenMP to express highly dynamic control and data flow over nested, dependent tasks. Our work complements these approaches, building on a systematic study of StreamIt and ΣC and on the latter's tool flow targetting a real-world many-core processor (Kalray MPPA) to offer additional insights into the design of expressive streaming languages that remain suitable for compilation on a many-core processor. We also discuss on embedded system requirements regarding the expression of non-functional properties in these languages.

III. StreamIt Background

In this section, we provide a very brief overview of the StreamIt language. See [2] for a more detailed description. Listing 1 shows a snippet of StreamIt code used for transforming a 2D signal from the frequency domain to the signal domain using an inverse Discrete Cosine Transform (DCT). The Figure 1 shows a graphical representation of the same code.

Listing 1: A snippet of StreamIt code demonstrating the construction of the DCT's stream graph shown in Figure 1. Specific keywords of the StreamIt language are underlined.

```
/* Discrete Cosine Transform */
int->int pipeline iDCT4x4_2D_fast_fine() {
   add Source(4);
   add iDCT4x4_1D_X_fast_fine();
   add iDCT4x4_1D_Y_fast_fine();
   add Printer(4);
}
/* Splitjoin filter */
int->int splitjoin iDCT4x4_1D_X_fast_fine() {
   split roundrobin(4);
   for (int i = 0; i < 4; i++) {
      add iDCT4x4_1D_row_fast();
   }
   join roundrobin(4);
}
int->int splitjoin iDCT4x4_1D_Y_fast_fine() {
   /* details elided */
}
/* Operator */
int->int filter iDCT4x4_1D_row_fast() {
   work pop 4 push 4 {
   for (int i = 0; i < 4; i++) pop();
   /* details elided */
   }
}
int->int filter iDCT4x4_1D_col_fast_fine() {
   /* details elided */
}
```

In fact, the StreamIt model of computation is synchronous data-flow (SDF) which is a special case of data-flow model [11]. Data-flow programs are directed graphs where each node (or process) represents a computation (or function) and each

edge represents a data path. The data-flow principle is that any node can perform its function whenever input data are available on its incoming edge. A node with no input edges may be performed at any time. Because the program execution is controlled by the availability of data, data-flow are often said to be data-driven. The central abstraction provided by StreamIt is an operator (called a *filter* in the StreamIt literature). Programmers can combine operators into fixed topologies, as shown in Figure 1, using three composite operators: *pipeline*, *split-join* and *feedback-loop*. Composite operators can be nested in other composites. The example code in Listing 1 shows four principal filters composed in a pipeline. In these filters, there are two split-join filters that connect atomic operators iDCT4x4_1D_row_fast and iDCT4x4_1D_col_fast_fine. Each operator has a *work* function that processes streaming data. To simplify the code presentation, we have elided the bodies of the work functions. When writing a work function, a programmer must specify the *pop* and *push* rates for that function. The pop rate declares how many data items from the input stream are consumed each time an operator executes. The push rate declares how many data items are produced. When all *pop* and *push* rates are known at compilation time, a StreamIt program can be statically scheduled.

IV. ΣC: A PROGRAMMING LANGUAGE FOR EMBEDDED MANYCORES

ΣC is a language designed in order to ensure programmability and efficiency on many cores. This language is built as an extension of the C language, with buildings suitable for expression of parallelism by using stream programming concepts. Close and familiar with C, it minimizes the syntactic burden of learning a new language, while making explicit the construction of parallelism [8]. In fact, ΣC defines a superset of Cyclo-Static Data-Flow (CSDF [4]), a generalization of SDF, which remains decidable though allowing data dependent control to a certain extent. Along with that, ΣC is sufficient to express complex multimedia implementations such as H.264/MPEG-4 Part 10 or AVC (Advanced Video Coding) [8].

A. The ΣC Programming Model

The ΣC programming model builds networks of connected agents. An *agent* is an autonomous entity, with its own address space and thread of control. It has an interface describing a set of ports, their direction and the type of data accepted; and a behavior specification describing the behavior of the agent as a cyclic sequence of transitions with consumption and production of specified amounts of data on the ports listed in the transition.

A *subgraph* is a composition of interconnected agents and it has also an interface and a behavior specification. The contents of the subgraph are entirely hidden and all connections and communications are done with its interface. Recursive composition is possible and encouraged; an application is in fact a single subgraph named root. The directional connection of two ports creates a communication link, through which data is exchanged in a FIFO order with non-blocking write and blocking read operations (the link buffer is considered large enough). An application is a static data-flow graph, which means there is no agent creation or destruction, and no change in the topology during the execution of the application. Entity instantiation, initialization and topology building are performed offline during the compilation process. System agents ensure distribution of data and control, as well as interactions with external devices. Data distribution agents are *Split, Join* (distribute or merge data in round robin fashion over respectively their output ports / their input ports), *Dup* (duplicate input data over all output ports) and *Sink* (consume all data).

B. The ΣC Programming Language

The ΣC programming language is designed as an extension to C. It adds to C the ability to express and instantiate agents, links, behavior specifications, communication specifications and an API for topology building by using some new keywords like *agent, subgraph, init, map, interface,...* but does not add communication primitives. The communication ports description and the behavior specification are expressed in the *interface* section. Port declaration includes orientation and type information, and may be assigned a default value (if oriented for production) or a sliding window (if oriented for intake).

The construction of the data-flow graph is expressed in the *map* section using extended C syntax, with the possibility to use loops and conditional structures. This construction relies on instantiation of ΣC agents and subgraphs, possibly specialized by parameters passed to an instantiation operator, and on the oriented connection of their communication ports (as in Figure 2). All assignments to an agent state in its *map* section during the construction of the application is preserved and integrated in the final executable. Listing 2 and Figure 2 shows an example of building a new data-flow graph in ΣC.

Listing 2: Topology building code of the ΣC subgraph shown in Fig. 2. Specific keywords of the ΣC language are underlined.

```
subgraph iDCT4x4_1D_X_fast_fine() {
  interface {
    in<int> input;
    out<int> output;
    spec{ {input[4] ; output[4] } };
  }
  map {
    int i;
    agent mSplit= new Split<int>(4,2);
    agent mJoin= new Join<int>(4,2);
    for (i = 0; i < 4; i++) {
      agent iDCT= new iDCT4x4_1D_row_fast();
      connect(mySplit.output[i], iDCT.input);
      connect(iDCT.output, myJoin.input[i]);
    }
    connect(input, mySplit.input);
    connect(myJoin.output, output);
  }
}
```

Exchange functions implement the communicating behavior of the agent. An *exchange* function is a C function with an additional exchange keyword, followed by a list of parameter declarations enclosed by parenthesis. Each parameter declaration creates an exchange variable mapped to a communication port, usable exactly in the same way as any other function parameter. A call to an exchange function is exactly like a standard C function call, the exchange parameters being hidden to the caller. An agent behavior is implemented as in C, as an entry function named *start()*, which is able to call other

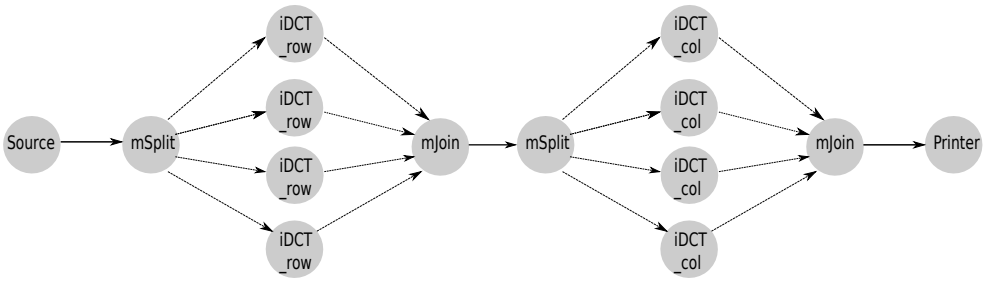

Fig. 1: Stream graph for the DCT application

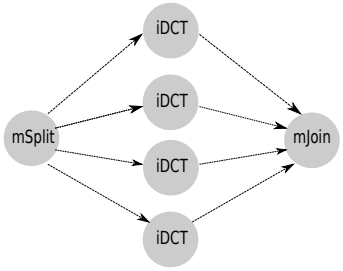

Fig. 2: Process Network topology of a ΣC subgraph, as shown in Listing 2. In this topology, multiple iDCT agents (iDCT) connected to one split (mSplit) and one join (mJoin)

functions as it sees fit, functions which may be exchange functions or not. Figure 3 and Listing 3 show an example of an agent declaration in ΣC. The main difference could be seen here is that ΣC could create CSDF graph with cyclically changing firing rules.

Listing 3: The iDCT agent's ΣC source code. Specific keywords of the ΣC language are underlined.

```
agent iDCT4x4_1D_row_fast() {
  interface {
    in<int> input;
    out<int> output;
    spec{input[2]; input[4] ; output[2] ; output[4]
        };
  }
  void step(int qin, int qout) exchange (input
      myIn[qin], output myOut[qout]) {
     /* details elided */
  }
  void start() {
     step(2, 2);
     step(4, 4);
  }
}
```

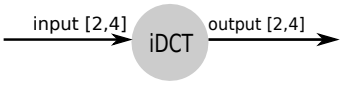

Fig. 3: The iDCT agent used in Figure 2 with one input and one output, and its cyclically changing firing rules

C. ΣC toolchain

The tool chain of ΣC includes four principal stages. The first stage of compilation named Frontend which performs syntactic and semantic analysis of the program and generates per compilation unit a C source file with separate declarations. The second compilation step of the tool chain aims at building the data-flow graph. Once built, further analyses are applied to check that the graph is well-formed and that the resulting application fits to the targeted host. The third stage makes the place and route (assign set of agents to a given cluster), and parallelism adaptation to the target. The last stage generates a relocatable object file with all the user code, user data and run-time data, then the final binary with the execution support. The diagram of this toolchain can be seen in the Figure 4.

D. ΣC applications

Some representative applications have been developed in ΣC in laboratory. We present results about the stability of the execution time of each agent measured on the *MPPA - 256* [5] clustered architecture. Figure 1 and 6 show two graphs of the DCT application (the graph created by ΣC is the same as the one generated by StreamIt because the source code is automatically translated as presented in the Section VI) and the Motion Detection application (a process of detecting a change in position of an object relative to its surroundings or the change in the surroundings relative to an object). The execution time of each agent in these streaming programs is described in Figure 5 and 7 as a function of the number of cores. These values are stable, in accordance with the assumptions of compilation heuristics for CSDF graphs: the execution time of each agent does not depend on the number of cores.

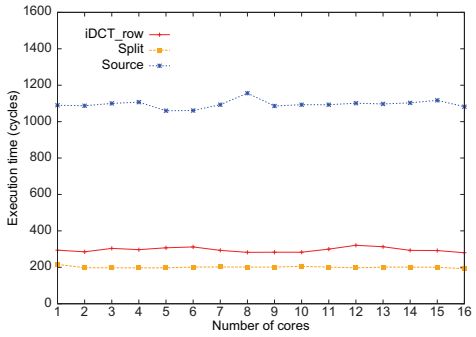

Fig. 5: The DCT execution time of each agent by number of cores

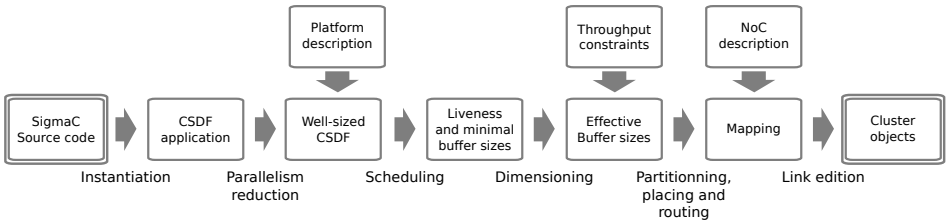

Fig. 4: The different stages of the toolchain. Starting with an application written in ΣC, we obtain an executable for the MPPA architecture.

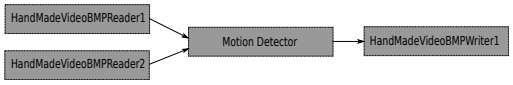

Fig. 6: The Motion Detector Graph

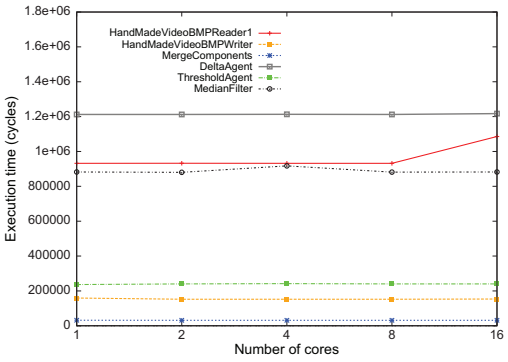

Fig. 7: The Motion Detection execution time of each agent by number of cores, *MergeComponents, Delta, Threshold* and *MedianFilter* located in the *MotionDetector* subgraph

V. Comparison between StreamIt and ΣC

This section discusses similarities and also differences between ΣC and StreamIt, with the aim of identifying limitations in meeting the growing requirements of modern embedded applications.

First, we can say that StreamIt and ΣC have a lot of similarities. The core of these languages are *agent* and *filter*, both are autonomous entities with their own address spaces unshared with other application tasks. In addition, StreamIt and ΣC programs are interconnected graphs (for ΣC, it is a composition of subgraph and interconnected agents, for StreamIt it is a set of filters connected to each other). Another similarity which is also a weakness, is that both languages support only static software architecture (links between tasks, data exchange amounts are determined at compile time) while new embedded many-core applications require dynamic expressivity (for ΣC) and context-dependent adaptation (the ability to change of the synchronous data-flow graph depending on the context). For example, the Motion Detector program, as seen in Fig. 6, in some cases need to add some agents before the HandMadeVideoBMPReader to clean up noises if the quality of the video is too low. Accommodating this requirement requires further research.

Another similarity of the two languages are real-time requirements that are often not well supported, although easily achieved. The main reason is that the model of computation of these languages is *data-driven* (actors are fired as soon as there are enough tokens in all of their input edges) and not *time-driven*. A solution could be found in [12] and will be discussed further in Section VII-B.

Besides these similarities, we can recognize a lot of differences between these two languages. While StreamIt tries to create a SDF graph of connected filters, the model of computation of ΣC is CSDF, which is also a special case of data-flow process networks. In SDF, actors have static firing rules: they consume and produce a fixed number of data tokens in each firing. This model is well suited to multirate signal processing applications and lends itself to efficient, static scheduling, avoiding the run-time scheduling overhead incurred by general implementations of process networks. In CSDF, which is a generalization of SDF, actors have cyclically changing firing rules. In some situations, the added generality of CSDF can unnecessarily complicate scheduling. Some higher-order functions can be used to transform a CSDF graph into a SDF graph, simplifying the scheduling problem [15]. To resolve this issue, a new scheduling policy noted Self-Timed Periodic (STP) Schedule, which is a hybrid execution model based on mixing Self-Timed schedule and periodic schedule while considering variable Inter-process Communication (IPC) times, could be implemented in ΣC [6]. In other situations, CSDF has a genuine advantage over SDF: simpler precedence constraints. This makes it possible to eliminate unnecessary computations and expose additional parallelism.

Another difference is that networks of processes in StreamIt are defined directly through a dedicated coordination language, distinct from the Java or C implementation of StreamIt filters. This limits the topology of the associated Network (StreamIt topology is hierarchical, and is mostly limited to series-parallel graphs with the important addition of feed-back loops. Special features like teleport-messaging are required to overcome this limitation, see [20]). While in ΣC, the networks of processes are built through compilation of a single language: the first step of the compilation symbolically executes the code constructing the network of so-called “agents” (individual tasks in the stream model), and the associated communication interconnect called “subgraph”. This approach has the advantage of expressing more general topologies, because it proceeds to an off-line execution of the first-stage compilation to build the process network [12]. Therefore, it is not necessary to describe the concept of *feedbackloop* in ΣC because the task graph model is more flexible than the series-parallel model of StreamIt.

In other words, feedback loops are used in StreamIt only to alleviate limitations of the programming model.

As a compiler, ΣC on MPPA can be compared to the StreamIt/RAW compiler, that is the compilation of a high level, streaming oriented, source code with explicit parallelism on a manycore with limited support for high-level operating system abstractions. However, the execution model supported by the target is different: dynamic tasks scheduling is allowed on MPPA; the communication topology is arbitrary and uses both a Networks on Chip (NoC) and shared memory. Moreover, the average task granularity in ΣC is far larger than the typical StreamIt filter (supposed to be more than 1 μS) because the current implementation does not provide task aggregation like in StreamIt. So a task switch always pay a tribute to the execution support (see [7]). In addition, the current ΣC toolchain does not support paging when the cluster memory size is insufficient. Furthermore there is no way to make the distinction between several states within an application (init, nominal, ...). Lastly, the toolchain does not take into account some other aspects like power consumption, fault management and safety.

VI. Transformation between ΣC and StreamIt

After studying StreamIt and ΣC, a method and tool were developed to convert StreamIt benchmarks in ΣC. This work aims to better understand these two languages and create a library of benchmarks for ΣC. This library allows us to use the many existing StreamIt examples as the number of ΣC applications is still insufficient and requires more programs to test the ability of language. But we have seen that there are some situations where it is undesirable to perform this transformation, as we shall see in the following example.

A. Rules for transforming

As shown in Section V, there are a lot of similarities between StreamIt and ΣC. One of them is the similarity between two models of computation SDF and CSDF. Transforming between these languages is also transformation from the SDF graph of StreamIt to the CSDF graph of ΣC. This leads to a problem that is the loss of the dynamism of the CSDF model because in a cycle, a CSDF agent can have many different firing rules. In this paper, we restrict our discussion to a language transformation from a SDF model of an StreamIt application to the same ΣC model.

In StreamIt, *filter* is the atomic element of programming corresponding to the concept *agent* ΣC. In addition, *Pipeline, SplitJoin*, and *FeedbackLoop*, three constructs for composing filters into a communicating network are able to be replaced by a *subgraph* in ΣC. The first rule to transform between ΣC and StreamIt is to find a way to convert a *filter* of StreamIt in an *agent* in ΣC. To realize this rule, the first thing we have to do is to determine the behavior of the filter which is declared in the *work* part of a filter while this portion of input and output is declared in the *interface* part of an agent. The second rule is to understand how filters of a StreamIt application connect between them. Basically, StreamIt uses *pipeline* as factor to connect between filters. Likewise, ΣC programmers have to create a new instance of agent and *connect* between these instances. Likewise, *splitjoin* filter in StreamIt can be replaced by a subgraph with system agents *Split* and *Join* in ΣC.

B. An automatic method to transform StreamIt programs in ΣC

The method used here for the transformation between these two languages is to build controller loops that detect StreamIt filters and convert them into ΣC agents. The diagram in Figure 8 represents how the method and tool work. 1 and 6 are beginning and ending states, relatively. Firstly, the tool will find in the StreamIt code the main program (state 2), which is a special pipeline filter. After, other elements will be detected and handled by other controllers: normal filter (state 3), Split or Join filter (state 4) and Source or Printer filter (state 5).

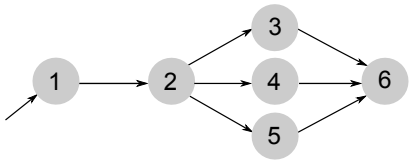

Fig. 8: Diagram of the Python program

The tool will read the code line by line when it encounters the declaration of the main program; it begins to create a main program that starts with ΣC subgraph *root* (the name of the main part in ΣC). Then it use the declarations of the StreamIt code to either prepare to chain agents for **pipeline** declarations, prepare to connect a list of agents for **split** and **dup** declarations, or reconnect the previous gathered list for **join** declarations. The tool is tested with several programs from the StreamIt benchmark suite. The ΣC generated code is compiled and have correct performance, as seen in the Section VI-C. However, as it is an automatic generated code, it sometimes looks like generated code, and is not as readable as hand programming. Nevertheless, the two codes play the same role and perform the same task in the program.

C. Evaluation

We tested the performance of the ΣC applications automatically translated from StreamIt source code by performing an experiment on some applications with several levels of complexity. The streaming applications used in the experiment are chosen among the StreamIt benchmark suite [19]. In total, 12 applications from different domains have been translated as shown in Table I.

TABLE I: Benchmarks used for evaluation

Domain	No.	Application	Source
Audio Processing	1	Audio beam former	[19]
Signal Processing	2	Multi-Channel beamformer	
	3	Discrete cosine Transform (DCT)	
	4	Fast Fourier Transform (FFT) kernel	
	5	Low-pass filter	[1]
	6	Band-pass filter	
	7	FMRadio with equalizer	
Mathematics	8	Minimal program	
	9	Moving Average Filter	
	10	Parallel computing	[17]
	11	Multiply two matrices	[19]
Sorting	12	Bitonic Parallel Sorting	

As can be seen in the Table II, the ΣC code is a little longer than the StreamIt code. This difference between two

languages could be explained by the following facts: the declaration of ΣC's *interface* is more complicated and flexible. Instead of using only one *work* function like in StreamIt, programmers can create CSDF graph while declaring ΣC's *interface*. Another reason is the number of filters of ΣC. StreamIt gives the concept of *anonymous stream*, a special filter unnamed and is used only once in the program. As there is no concept of anonymous agent in ΣC, the tool will automatically create a new agent in the generated code, which could be reused when needed and also explain why the ΣC code is longer than the StreamIt code.

TABLE II: StreamIt vs. ΣC

	StreamIt		ΣC	
Benchmark	**Lines**	**Filters**	**Lines**	**Filters**
Audio beam former	219	5	350	8
Multi-channel beamformer	398	9	380	9
Discrete cosine Transform (DCT)	651	19	995	21
Fast Fourier Transform (FFT) kernel	168	8	251	8
Low-pass filter	43	4	96	4
Band-pass filter	61	7	154	7
FMRadio with equalizer	167	12	335	14
Minimal program	12	3	45	3
Moving Average Filter	64	4	97	4
Multiply two matrices	163	9	290	12
Bitonic Parallel Sorting	260	10	412	10

After translation into ΣC, we ran some of the benchmarks that exhibit different levels of task granularity to understand the impact of extra communication costs on parallel efficiency. We ran the tests on the *MPPA - 256* [5] clustered architecture, from Kalray, comprising 256 user cores (i.e., cores with fully processing power provided to the programmer for computing tasks) organized as 16 (4×4) clusters tied by a Network-on-Chip (NoC) with a torus topology. As can be seen in Figure 9, the throughput normalized (in comparison with the throughput obtained in the case of mono-core) of the programs augmented when the number of cores used increased with a bottleneck after 12 cores and at this point, the throughput has only increased by a factor of 1.3. This result could be explained because of many reasons: there was not sufficient parallelized work to offset the extra communication costs because of the lack of coarse-grained parallelism benchmarks in StreamIt. In addition, the overhead of semi-dynamic scheduling of tasks within a cluster in MPPA is another reason for this result, because it augments as the number of cores to manage augments. A better result could be obtained if we use coarse-grained benchmarks with better coefficient of parallelism to overcome the extra communication costs between cores.

D. Limitations of the translation tool

A problem we encountered when translating from StreamIt to ΣC is the conversion of anonymous filters. The new agent created automatically results in difficulties when connecting between ports of agents. For example, new agents created in the *Multiply two matrices* application increased significantly the compilation time (127s in comparison with 5s when revising the code by hand). In addition, there is some similar concepts in StreamIt and ΣC, such as *Identity* filter. However, if this concept is used automatically in ΣC source code, the application's graph becomes more complicated, resulting in a decrease in performance. This problem could be resolved by making some changes in the source code by hand. For

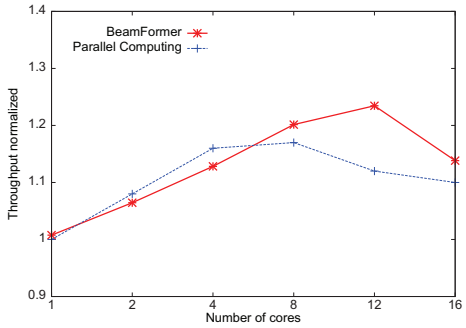

Fig. 9: Throughput normalized for the *BeamFormer* and *Parallel Computation* application

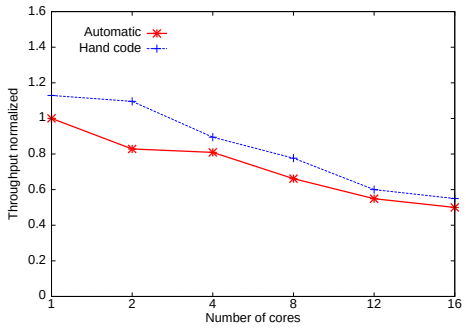

Fig. 10: Throughput normalized for the DCT application implemented by hand and by the Python program

the *Multiply two matrices* application, the number of agents declared could be reduced to 9 by removing the Identity filter used in the StreamIt source code along with improvements in throughput (with an average of 55%).

Another problem can be seen in Fig. 10, the throughput decreases when the number of cores increases. To understand the impact of the results, we use the concept of *synchronization time*. Synchronization is a special form of communication, for which data are control information. Its role is to enforce the correct sequencing of actors firing, and to ensure the mutually exclusive access to certain shared data. In the case of this example, the execution time of each agent is relatively small when compared to the synchronization time, so the efficiency that the parallelism gives can not offset the extra synchronization costs, along with the switching time between cores. A few changes in the automatically translated source code could deliver an average improvement of 15% in throughput as can be seen in Fig. 10. In addition, the problems of synchronization time will be discussed in Section VII-B. As mentioned earlier, a slight difficulty is that there is no concept of *feedback loop* in ΣC because the task graph model is more flexible. Therefore, a small number of StreamIt applications using this concept have to be implemented by a *for* loop in ΣC. This translation can also be automated without loss of expressiveness.

VII. Discussion

As we saw in Section V, StreamIt and ΣC, despite its differences, have a lot of common limitations to meet the demands of emerging embedded applications. In this context, this

section will introduce some ideas to improve these languages in the future.

A. Dynamic expressivity and reconfiguration

One of the main limitation of the compilation toolchain of StreamIt and ΣC is that it mostly meant for static instantiation (the set of tasks and the program's graph is in most cases fixed at compilation-time). A more versatile toolchain is required for future applications. For example, several configurations must be determined depending on the possible dynamic instantiation of subgraphs [12]. For that purpose we can think of a short-term approach beased on the compilation of several static configurations, and a more ambitious path relying on runtime configurations and an online (just-in-time) graph compiler. Supporting several static graphs seems rather immediate: enumerate all possible dynamically instantiated graphs, and generate a statically configured set of tasks based on the worst-case instantiation. For further improvements, it would be possible to identify several modes and special point where to switch from one mode to the other, and changing the static allocation of on-chip loaded tasks and their configuration, accordingly. Nonetheless, the case of a fully dynamic execution would require more research (it would be alike an on-chip stream programming graph compilation and optimization framework). Another improvement would be the support of dynamic *data transfer rates*. In the real world, many applications require this capacity (see Section I). StreamIt also supports dynamic input and output rates, at the expense of compilation-time optimizations and performances. In this context, a design of a hybrid static-dynamic scheduler can be developed for streaming languages as the extension for StreamIt presented in [17].

B. Real-time interface and implementation of scheduling theories

One aspect where stream languages such as StreamIt and ΣC must improve is the expression of non-functional requirements. In particular, embedded applications need to capture guarantees on worst case latency or delay, and to associate these with tasks and filters. Such guarantees may apply to a subset of the stream graph entries, and a subset of the outputs, or a specific map between the two. An extension called StreamFlex introduced new keywords to these language, such as *delay, clock, starttime,...* [18]. It is a programming model inspired both by the StreamIt language and, loosely, by the Real-time Specification for Java. The really difficult point is the calculation of the Worst Case Execution Time (WCETs) for each agent of the stream. If it can be computed (this depend on the complexity of the algorithms, on the micro-architecture of the target processor, and on the ability to model interferences between cores), a compiler-only approach should be sufficient. For the general case, online monitoring of the latency must be implemented in the execution support. These questions remain essentially open.

In addition, the Self-Timed Scheduling (STS) strategy, also known as as-soon-as-possible, is considered as the most appropriate policy for streaming applications modeled as data-flow graph, because it delivers the maximum achievable throughput and the minimum achievable latency [3]. However, this result assumes that synchronization time is negligible [6]. In practice, each synchronization event can cost up to four accesses to shared memory [14]. Due to the complex and irregular dynamics of self-timed operations, and to model the significant synchronization overhead, many different hypotheses were suggested. These include contention-free communication or considering uniform costs for communication operations [3]. However, neglecting subtle effects of synchronization is not realistic with regards to actual systems and their hard real-time guarantees. In addition, using a predefined schedule of accesses to the shared memory [10] makes run-time less flexible. To cope with this challenge, periodic scheduling is receiving much more attention for streaming applications [3]. These algorithms provide many nice properties such as timing guarantees for each application, which is always very important in the critical embedded systems in avionics, temporal isolation and low complexity of the schedulability test. It was shown that interprocessor communication overhead can be defined as a monotonically increasing function of the number of conflicting memory accesses in a given period of the schedule. For this purpose, new scheduling policy such as Self-Timed Periodic (STP) Schedule [6] (which can achieve 25% to 60% improvement in term of latency compared to the Implicit-Deadline Periodic (IDP) schedule, firstly presented in [3], and yield the maximum achievable throughput obtained under the STS schedule for a large set of graphs) would be implemented in the toolchain of new stream programming languages.

VIII. Conclusions

Stream programming provides a sound basis for parallel programming on many-core architectures. This study compares StreamIt and ΣC, two static stream programming languages used in research and production, respectively. There are a lot of common aspects between these two stream languages. One important difference is that the task graph model of ΣC is more expressive while StreamIt models aspects related to execution time more precisely. StreamIt has been developed over 10 years, and its library of benchmarks is large in comparison to ΣC. We designed a method and tool to transform the library written in StreamIt into a ΣC library. This tool has been evaluated on diverse StreamIt programs and provided very promising results, generated fully automatically, and running on many-core hardware, the Kalray MPPA 256. Based on this study, the paper makes some propositions for future stream programming models. The success in mastering these issues could provide the key to the support of dynamic, data- and context-dependent streaming applications. Better support for real-time and other non-functional constraints also remains a critical objective.

References

[1] Streamit cookbook. Technical report, MIT, September 2006.

[2] Streamit language specification. version 2.1. Technical report, MIT, September 2006.

[3] M. Bamakhrama and T. Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 195–204, New York, NY, USA, 2011.

[4] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3255–3258 vol.5, May 1995.

[5] B. Dupont de Dinechin, P. Guironnet de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel. A distributed run-time environment for the kalray mppa-256 integrated manycore processor. *Procedia Computer Science*, 2013.

[6] A. Dkhil, X. Do, S. Louise, and C. Rochange. Self-timed periodic scheduling for cyclo-static dataflow model. In *ICCS*, volume to be published of *Procedia Computer Science*, 2014.

[7] P. Dubrulle, S. Louise, R. Sirdey, and V. David. A low-overhead dedicated execution support for stream applications on shared-memory CMP. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '12, pages 143–152, New York, USA, 2012. ACM.

[8] T. Goubier, R. Sirdey, S. Louise, and V. David. ΣC: A programming model and language for embedded manycores. In *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP'11, pages 385–394, 2011.

[9] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[10] M. Khandelia, N.K. Bambha, and S.S. Bhattacharyya. Contention-conscious transaction ordering in multiprocessor dsp systems. *IEEE Transactions on Signal Processing*, 2006.

[11] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE, vol. 75, no. 9,*, pages 1235–1245, 1987.

[12] S. Louise. Programmability in the age of the manycore, beyond stream programming. *ACM Trans. Embedd. Comput. Syst.*, to be published.

[13] J. R. McGraw., S. K. Skedzielewski, S. J. Allan, R. R Ollehoeft, J. Glauert, C. Kirkham, T. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language. Technical report, Lawrence Livermore National Laboratory, March 1985.

[14] P. K. Murthy and S. S. Bhattacharyya. Memory management for synthesis of dsp software. 2006.

[15] T.M. Parks, J.L. Pino, and E.A. Lee. A comparison of synchronous and cycle-static dataflow. In *Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on*, volume 1, pages 204–210 vol.1, Oct 1995.

[16] A. Pop and A. Cohen. Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Trans. Archit. Code Optim.*, 9(4):53:1–53:25, January 2013.

[17] R. Soulé, M. I. Gordon, S. Amarasinghe, R. Grimm, and M. Hirzel. Dynamic expressivity with static optimization for streaming languages. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 159–170, New York, NY, USA, 2013. ACM.

[18] J. H. Spring, J. Privat, R.d Guerraoui, and J. Vitek. Streamflex: High-throughput stream programming in java. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 211–228, New York, NY, USA, 2007. ACM.

[19] 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, USA, 2010.

[20] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe. Teleport messaging for distributed stream programs. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 224–235, 2005.

[21] I. Watson and J. Gurd. A practical data flow computer. *Computer*, 15(2):51–57, 1982.