Interactive Debugging of Dynamic Dataflow Embedded Applications

Kevin Pouget*†
Miguel Santana*

*STMicroelectronics
Crolles, France
first.lastname@st.com

Patricia López Cueva*†
Jean-François Méhaut†

†University of Grenoble, LIG, CEA
Grenoble, France
first.lastname@imag.fr

***Abstract*—Debugging parallel and concurrent applications is well-recognized as a time-consuming task, which often requires a significant part of the application development process. In the context of embedded systems, Multi-Processor-System-on-Chip (MPSoC) architectures feature numerous multicore processors which may be coupled with heterogeneous processors like Digital Signal Processors (DSPs) and/or application-specific accelerators. In this situation, it is important that developers are provided with high-level programming environments able to efficiently exploit these architectures, as well as suitable debugging tools.**

Dataflow programming models were explicitly designed to program parallel architectures and they have the ability to abstract away heterogeneous computing complexity. In addition, the stream-processing aspect of multimedia algorithms naturally exhibits data-dependency graphs, which simplifies application design and implementation.

In this paper, we propose a new approach for interactive debugging of dataflow applications. Going beyond the long-established ability of interactive debuggers to support sequential programming languages, we describe the functionalities they should be able to provide to debug embedded and parallel dataflow applications. Then we demonstrate our solution to this problem with a proof-of-concept debugger targeting the dataflow framework used on an industrial MPSoC platform. We also explain the development challenges we faced during the implementation of this GDB-based debugger and illustrate its efficiency through a case study of a video decoder debugging session.

***Index Terms*—dataflow; debug; embedded; parallel; mpsoc;**

I. Introduction

In the context of embedded computing, Multi-Processor-System-on-Chip (MPSoC) architectures have been introduced during the last decade to answer the ever increasing demand for computational power. MPSoCs typically offer heterogeneous parallelism, with numerous general-purpose multicore processors, but also Digital Signal Processors (DSPs) and application-specific accelerators. Today's multimedia embedded applications need to support high-quality image, video or voice processing. So, they have to be fully adapted to their underlying platform if they want to meet performance expectations. However, traditional models for parallel programming like multithreading are not able to efficiently exploit such a heterogeneous parallelism.

Multimedia applications are good candidates for a dataflow based implementation. Indeed, stream-processing algorithms naturally exhibit data-dependency graphs, which simplifies application design and implementation. Recent publications like [1], [2] show successful demonstrations of the efficiency of dataflow languages to implement video decoders. Also, in 2008, the Moving Picture Experts Group (MPEG) used a dataflow language (CAL, [3]) to describe their new video standard, MPEG-RVC [4].

Moreover, the heterogeneous parallelism of MPSoCs can be exploited by dataflow programming models. Indeed, these models hide the underlying processor specificities and discharge the mapping, scheduling and tuning to the runtime system [5], [6]. Hence, dataflow programming on MPSoC systems appears as an interesting trade-off.

The dataflow programming models have been developed since the 1970s/1980s as an alternative to the conventional paradigms based on "von Neumann" processors [7]. In these models, the focus is shifted from the stream of instructions being executed (i.e. moving the program counter) towards the dependencies between the data. Put briefly, this means that an instruction, or a block of instructions, is not executed when the program counter reaches it, as in imperative programming, but rather when its operands are ready. The models were explicitly designed to exploit parallel architectures and try to avoid the main bottlenecks of von Neumann hardware: the global program counter and the global updatable memory.

As dataflow models put an important focus on data dependencies, applications designed with such models can be represented as directed graphs. The nodes of the graph correspond to the different data transformations of the application. The inbound arcs represent their input data and the outbound arcs represent the data they produce. Hence, the arcs materialize the data dependencies.

In this paper, we primarily focus on *dynamic* dataflow models [8], which provide a high expressiveness and limit the constraints imposed to developers. Indeed, *decidable* models like synchronous dataflow allow correctness analysis with formal methods, static and deadlock-free actor scheduling, etc., but that comes at the price of a reduced expressiveness

and ease of modeling. Thus, these models are not always suitable for all the requirements, especially in the case of applications processing dynamic streams. So the dynamic dataflow models targeted by our approach would usually not support model analysis and verification, but rather emphasize programmability and ease of modeling.

The absence of outstanding dataflow languages is slowing down the adoption for industrial development. Indeed, although some languages have been designed, they remain at the state of research prototypes. It is important for the industry to rely on standard technologies to ensure the durability of their products. Thus, the only viable alternatives appear to be the use of dataflow frameworks based on standard languages, which are *de facto* imperative or object-oriented languages. See for example PEDF in Section IV, CILK [9], or the initial version of STREAMIT [10].

Executing a dataflow application under the control of an interactive debugger would allow developers to clearly understand the subtleties of the execution flow. Well-crafted breakpoints and step-by-step execution can highlight problems in complex situations, which would have been impossible to foresee with manual code analysis or trace tools. Developers can also use debuggers to monitor and profile applications. This allows them to have real-time feedback about the actual application execution, which may depend of the underlying platform or the local environment. Furthermore, the deterministic nature of dataflow communications fades away the intrusiveness brought by debugger breakpoints and user interactions. Indeed, the execution semantic is not altered by the slowdown they introduce.

However, it appears that current interactive debuggers only provide programming-language level features, and actually only for imperative languages. Apart from threads and processes, none of the high-level concepts used by developers to build dataflow applications are materialized in current tools.

We believe that the next generation of debuggers should take into account the programming model as well as the runtime environment of applications. Indeed, they have a key role in the overall execution and they reflect the programming guidelines followed during development.

Therefore, the main contribution of this article is a novel approach for the design of interactive debuggers, which describes the key functionalities that should be provided to allow an efficient debugging of dataflow applications. We also detail how this approach has been implemented with an industrial dataflow framework for an MPSOC system, on top of GDB, a popular free debugger.

This article is structured as follows: in Section II we introduce the problems encountered during dataflow application interactive debugging. Then, we detail in Section III our proposal for an efficient dataflow-aware debugger. In Section IV we describe how we applied the approach in the context of an embedded dataflow framework developed by STMICROELECTRONICS. We present how our proof-of-concept debugger was implemented as a GDB extension in Section V and we illustrate its usage in Section VI with the debugging of a video decoding application case study. In Section VII, we discuss the existing work related to debugging applications running on dataflow machines as well as runtime and model awareness in the context of component-based software engineering and task-based parallel programming. Finally, Section VIII summarizes the contributions of this paper and details the future work.

II. DATAFLOW DEBUGGING CHALLENGES

Interactive debugging is a complex activity where developers try to figure out where the actual code behaviour diverges from expectations. To do that, they must have a total and precise control over the application execution.

While debugging dataflow applications, developers face a significant number of challenges which do not exist in traditional von Neumann code. In order to improve the efficiency of the bug tracking activity, the debugger has to accommodate with such difficulties:

Tools Too Low-Level As current dataflow applications are built on top of non-dataflow languages, debuggers are not, natively, able to provide any information specific to the dataflow nature of the application.

Token-Based Execution Firing The execution of a dataflow actor can only start when the required input tokens have been generated. This concept does not exist in the von Neumann model, where the execution of a statement is only conditioned by the path of the program counter.

Non-Linear Execution When a dataflow assignment instruction is executed, the actor(s) connected to this data become(s) executable. Semantically, the execution flow is forked and follows not only its normal stream but also the different outgoing arcs of the node.

Graph-Based Application Architecture The architecture of the application is structured by its data-dependency graph. However, current debuggers only offer a sequential – or multi-sequential if the runtime environment is parallel – view of the source code instruction stream. All the arcs of the graph are unavailable to developers.

Token-Based Application State The validity of a dataflow algorithm depends on the correct dispatching of data tokens. As in the theoretical models, nodes are supposed to be stateless, the set of tokens present in the application holds the entire execution state. Thus, it is important for developers to have the ability to query the debugger about these tokens. Current tools are not aware of this concept and thus are unable to provide such information.

Information Flow Dataflow applications can be composed of a significant number of actors, which successively apply a specific transformation to their incoming data. In order to understand the current value of a token, developers have to figure out the exact sequence of transformations undergone by this token. More concretely, this implies that the debugger should record the different token values over their processing steps.

If we think about the execution of the dataflow application as a state diagram, we can understand that some of the

transition events have a great importance in the debugging process, as they lead the application into an invalid state. So, in order to simplify the debugging activity, an interactive debugger should be able to catch and work with the key events occurring in the dataflow execution. It should then provide developers with an high-level representation of the execution steps, based on the very entities used to develop the application.

In the next section, we present the debugging approach we designed to help developers to locate more easily the problems of dataflow applications. For each of the main steps of the debugging process, we highlight the features which can be used to overcome the issues presented in this section.

III. Dynamic Dataflow Debugging Approach Proposal

Our contribution consists of a set of functionalities that a debugger should implement in order to shift the main focus towards the data-controlled style of execution of the dataflow model. This way, we expect to provide developers with more efficient tools to debug their dataflow applications. Instead of working with threads, processes and function calls, they will interact with dataflow actors and inbound and outbound connections, tight together into a graph of data dependencies.

Our approach relies on the detection and handling of the key events of the dataflow execution. These events should be interpreted to follow the execution of the dataflow-related aspects of the application, allowing the debugger to provide accurate information and control mechanisms to debug dataflow applications.

Stopping the Application Execution Debuggers should leverage the additional events introduced by the dataflow model to provide more adapted flow control mechanisms. Namely, developers should be able to stop the execution when a specific actor gets fired or upon the generation/reception of a specific set of tokens. Conditional breakpoints should also be available, for instance based on the number of tokens transmitted, their source/destination or content.

Executing the Application Step-by-Step The exchanges and interactions between actors should be exploited to improve step-by-step execution. It should be possible to navigate in the application architecture based on the graph of data dependencies. This means that developers should be able to follow an inter-actor data transfer the same way they step into a classic function call (it may involve several framework-dependent function calls and/or another execution context). Furthermore, the concurrent aspect of these instructions should be taken into account. The debugger should allow developers to decide which path to follow (i.e. local or remote actors) or let them block the other execution paths until a latter investigation.

Inspecting the Application State The token distribution is an important aspect of the application state. So, developers should be provided with an overview of the tokens currently available in the data links. Additionally, information about the tokens received (resp. sent) on the actor inbound (resp. outbound) connections would facilitate the detection of unexpected interactions. Details about the state of each actor should also be available, including the source-code line currently executed, and whether or not it is currently blocked, i.e. waiting for more data.

Altering the Normal Execution Developers should be able to tweak the application in order to test or verify debugging hypothesis. With respect to the dataflow programming model, this means inserting, modifying or deleting tokens transmitted over data links. For instance, this capability would allow developers to untie a deadlock situation and analyze the further execution. They could also insert corner-case tokens to follow and ensure their correct processing.

Two-Level Debugging Finally, as the instructions of a dataflow application are eventually written in a standard programming language and executed by the processor like traditional code, language-based and low-level debugging commands should still be available. Indeed, although some bugs may lay in the programming-model related aspects of the application, there is a chance that the problems are hidden deep down in the language instructions. So, memory and processor inspection, breakpoints and watchpoints (maybe actor-specific) and other step-by-step execution control primitives should be directly available.

The following sections present a proof-of-concept debugger implementing this approach.

IV. Debugging Features for an Embedded Dataflow Framework

In this section, we describe the embedded system *Platform 2012* (P2012) [11] and its dataflow framework *Predicated Execution DataFlow* (PEDF). In this context, we outline the debugging features we designed, based on the approach described in the previous section. The implementation of the debugger is presented in the next section, and Section VI details its usage with a video decoder debugging session.

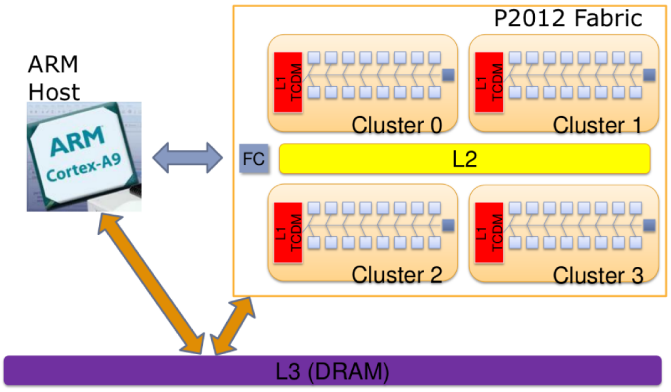

Fig. 1: P2012 architecture

P2012 is a low power, parallel and embedded platform research project developed by STMICROELECTRONICS and CEA. It targets high-definition audio and video processing. The platform is shaped as an accelerator architecture, using a multicore general-purpose ARM processor on the host[1] side and clusters of STXP70 configurable processors on the fabric side. As depicted in Figure 1, the processing elements of a cluster share the L1 memory. Inter-cluster communication is done through the L2 memory, whereas host-fabric exchanges are performed by DMA[2] controllers with the L3 memory.

P2012 also supports heterogeneous computing: hardware accelerators can be wired into the fabric and controlled by the associated processing element. One of the key objectives of such a platform is to support an efficient H.265/HEVC video decoder [12].

In this work, we focused the development of our prototype on the functional simulator of P2012 (at time of writing, there is no real platform available). This simulator runs on standard development workstations and uses SYSTEMC[3] threads to simulate the different processors.

PEDF is a framework for dynamic hybrid dataflow programming, designed to exploit P2012 heterogeneous architecture. It provides a structure dataflow model, similar to what is presented in [7]. PEDF also originates from dynamic dataflow modeling, so it does not enforce any constraint in actor's sending and receiving rates. Besides, it offers advanced scheduling capabilities, allowing the modification of the dataflow graph behavior during its execution (based on a set of predicates) or run some parts of the graph at different rates. It is based on the C++ language to benefit from the existing tool-chain (the compilation suite, but also the platform simulators). PEDF defines three classes of entities:

Filter It is a computing entity, corresponding directly to the *actors* of the dataflow model. Filters have inbound and outbound data links. The code of a filter is written in a subset of the C language which will be eventually synthesized into a hardware accelerator.

Controller There is one controller per module, which is responsible for the scheduling of the relevant filters (i.e. registering filters for execution to the runtime system), according to the application algorithm. A controller runs on a processing element of the fabric.

Module It corresponds to a sub-graph of filters plus a controller. Like filters, modules have inbound and outbound data links, corresponding to the unconnected arcs of the inner graph. Thus, modules can be hierarchically interconnected.

Figure 2 presents the visual representation of a simple module, composed of two filters and a controller.

In the remaining of this section, we describe how we can build and implement this module with PEDF, as well as the debugging features associated with each of the steps.

[1]We use the term *host* to refer to the general-purpose processor of the platform, which is different from its remote debugging/development meaning.

[2]Direct Memory Access

[3]http://www.systemc.org

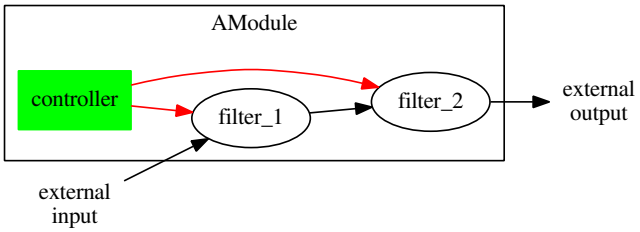

Fig. 2: PEDF Dataflow Graph Visual Representation

A. Designing the Data-Dependency Graph

The PEDF dataflow graph is built with the MIND[4] architecture compilation tool-chain, augmented with PEDF annotations. MIND provides a description language to specify filter's architecture and interfaces. Its compiler generates a C++ version of the architecture, based on PEDF and platform-specific templates. Module entities are in charge of defining inter-filter bindings.

The following code excerpt presents the definition of Module `AModule`, which contains a controller and two filters. It has two external connections. The filter definition is presented afterwards. In the last lines of the module definition, we can see how the different connections are bound together. The graph in Figure 2 was generated from this description.

```
@Module
composite AModule {
  contains as controller {

    output U32 as cmd_out_1;
    output U32 as cmd_out_2;

    source ctrl_source.c;
  }

  // External connections
  input  U32 as module_in;
  output U32 as module_out;

  // Sub-components
  contains AFilter as filter_1;
  contains AFilter as filter_2;

  // Connections
  binds controller.cmd_out_1
     to filter_1.cmd_in;
  binds controller.cmd_out_2
     to filter_2.cmd_in;

  binds this.module_in
     to filter_1.an_input;
  binds filter_1.an_output
```

[4]http://mind.ow2.org/

```
    to filter_2.an_input;
  binds filter_2.an_output
    to this.module_out;
}
```

Both filters are defined by the primitive type `AFilter` presented below. They have private and attribute data, as well as an input and an output data dependency. They also have a control input dependency.

```
@Filter
primitive AFilter {
  data      stddefs.h:U32 a_private_data;
  attribute stddefs.h:U32 an_attribute;
  source    the_source.c;

  input  stddefs.h:U32 as an_input;
  input  stddefs.h:U8  as cmd_in;
  output stddefs.h:U32 as an_output;
}
```

Contribution #1: Graph Reconstruction
On the debugger side, this graph structure will be dynamically reconstructed during the initialization phase of the framework. As it is a key abstraction for the application architecture, it is made directly available to developers through all of the dataflow-related functionalities of the debugger. Auto-completion capabilities make it straightforward for developers to use filter and connection names while they are typing their commands.

The graph is shown upon user request and can either be updated in real time or only when the execution is stopped. (The former case may introduce an additional delay, due to the graph generation time.)

B. Module Controller Execution

Module controllers are responsible for triggering the execution of the filters of their module. Filter execution model is based on *step*s. For each *step*:

1) The controller decides which filter must be executed: `ACTOR_START(name)`.
2) The `WORK` method of filters scheduled for execution is started.
3) The controller can wait for the actual beginning of the execution: `WAIT_FOR_ACTOR_INIT()`.
4) The controller can request filters to stop their execution at the end of this *step*: `ACTOR_SYNC(name)`.
5) The controller can wait for the actual end of the *step*: `WAIT_FOR_ACTOR_SYNC()`.

(NB: `START` and `SYNC` commands can be merged into a single `ACTOR_FIRE` command).

This scheduling capability is not part of the common dataflow models, although it shares some similarities with the control tokens, with the exception that the deterministic property is lost.

Contribution #2: Scheduling Monitoring
Our dataflow debugger captures this information, so that developers can quickly review which filters are ready to be executed, not scheduled, or have already finished the *step*. Developers can also request an execution stop at the beginning or end of a *step*, or when a controller schedules a filter for execution.

C. Filters Data Processing

PEDF filters implement the core data processing tasks of the application. They are intended to be synthesized into hardware accelerators, and for this reason, strong constraints have been defined for their implementation. In particular, the use of a restricted subset of the C language, which permits a direct transformation to RTL (Register Transfer Level) circuits. Filters must define a `WORK` method, implementing one *step* of the processing. They can access their private data, attributes and connections with the name specified in the architecture definition, prefixed by `pedf.data.`, `pedf.attribute.` and `pedf.io.`, respectively.

With respect to the dataflow part, the data exchanges are transparent to the developers. In the previous example, a filter can read (resp. write) its data with `pedf.io.an_input[n] = d;` where `n` is the highest unread (unwritten) index. (This array notation corresponds to the *structure* model of dataflow mentioned earlier.)

Contribution #3: Execution Flow Monitoring
On the debugger side, we focused on the flow-of-token aspect, which is key to the dataflow model. Namely, we enabled the possibility of following a token through a dependency by intercepting the indexes of the token pushed in and out of the link. As the model and the implementation ensure that the data order is preserved, we can stop the execution at the right location in a deterministic way.

In this section, we have detailed our contribution to improve interactive debugging of PEDF applications. This consists in adapting the debugger features to the dataflow programming model: we introduced debugging abstractions closely matching PEDF dataflow actors (filters, modules and controllers). We also detailed how to refined the debugger's commands to better take into account the graph structure of the application. We finally described new functionalities to exploit the scheduling capabilities of PEDF controllers.

In the following section, we introduce the inner implementation specificities of this debugger and how we extended GDB to build it.

V. Extending GDB to Support Dataflow Debugging

GDB is the debugger of the GNU project [13]. It has a very large user community in both general and embedded computing. We decided to base our work on this tool because of its advanced process inspection and control capabilities and to simplify user and products handovers. Moreover, recent versions of GDB export PYTHON bindings, which allow an easy and efficient development of extensions. Thus, all the classic functionalities of GDB are available in our debugger. Figure 3 presents a schematic representation of a traditional debugger like GDB connected to an MPSoC platform, as well as our dataflow-awareness extension.

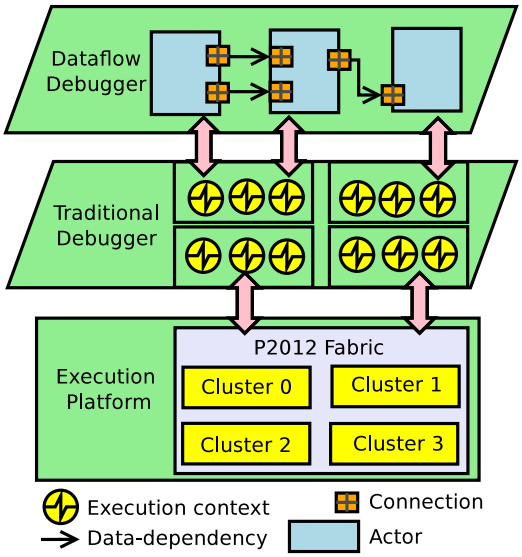

Fig. 3: Two-level debugging of an MPSoC platform.

In order to limit the intrusiveness of the debugging in the execution, we decided not to alter the dataflow framework. Therefore, the debugger needs to update its internal representations whenever a dataflow event occurs.

Our runtime-information capture mechanism relies on internal function breakpoints set at the entry and exit points of the programming-model related functions exported by the dataflow framework. Based on the API definition, calling conventions and debug information, we parse the relevant function arguments to extract the information required to update the debugger internal state.

Function breakpoints extend the traditional breakpoint mechanism by attaching to the breakpoint the semantic definition of the operation it monitors. Each time the breakpoint is triggered, a specific action is executed to update the internal representations. Based on PEDF API and source code, we elected the locations responsible for key dataflow operations and implemented the corresponding representation modification back to the debugger's function breakpoints. The concept of *finish breakpoint* was also introduced in GDB Python API to programmatically catch the return point of a function. This allows the interception of output and updated parameters.

The top part of Figure 3 presents the internal representation used to describe a dataflow application:

Actor objects represent the filters, controllers and modules of the PEDF application. We keep a reference to the execution context to which they are bound and their list of inbound and outbound connections.

Token objects are created and transmitted over the different debugger dataflow entities. They are not associated with any framework object, their state only correspond to the logical implications of runtime events. They can trigger an execution stop when they reach their target actor, or according to the content of their payload.

Connection objects correspond to a data dependency of an Actor object. They are associated with the simulator entity responsible for the data transfer. These objects produce or consume Token objects when the corresponding event is produced by the simulator. They are also responsible for stopping the execution if the relevant conditions are met.

Link objects bound together an incoming and an outgoing connection. They receive, hold and transmit Token objects required by Connection objects.

In the current PEDF implementation, the framework API and implementation are rich enough to capture all the information required to implement our support. The only static information we rely on is provided through the standard Dwarf [14] debug structures.

However, in other dataflow frameworks, further compile-time information might be required, such as details about the implementation of the actors or the graph of data dependencies.

Using only standard debug information allows developers to connect more easily our debugger to their application, as it does not require any specific recompilation. On the other side, we could exploit compiler-generated information to enrich our model and improve the overall performance.

Indeed, our frequent use of breakpoints introduces a slowdown in the application. This is mainly due to the breakpoints related to data exchanges, which can be triggered very frequently in a fine-grain dataflow application. In order to cope with this problem, two options have been considered:

- Disabling the data exchange breakpoints until the "critical part" of the execution is reached. Control tokens do not rely on the same breakpoints, so they can still be used. Source code actor-specific breakpoints and watchpoints can also help reaching the suspicious area more rapidly.
- Framework cooperation If the framework could provide actor-specific location for data exchange breakpoints, then the set of enabled breakpoints could be reduced to the actors of interest. This would significantly improve performance during the non-interactive parts of the execution.

For the sake of simplicity, only the first option has been implemented so far.

In the following section, we detail a case study experimentation illustrating how we used this debugger to control the execution of a video decoder based on PEDF and running on P2012 MPSoC.

VI. Case Study: Debugging a H.264 Video Decoder

This section presents a case study which illustrates our approach in the context of a real-world application: a H.264 video decoder [15] designed with PEDF to exploit Platform 2012 heterogeneous computing fabric. In the following, we come back on the challenges highlighted in Section II and explain how each point is addressed by our debugger.[5]

[5]We assume that the reader is familiar with GDB command-line syntax. Otherwise, please refer to [13].

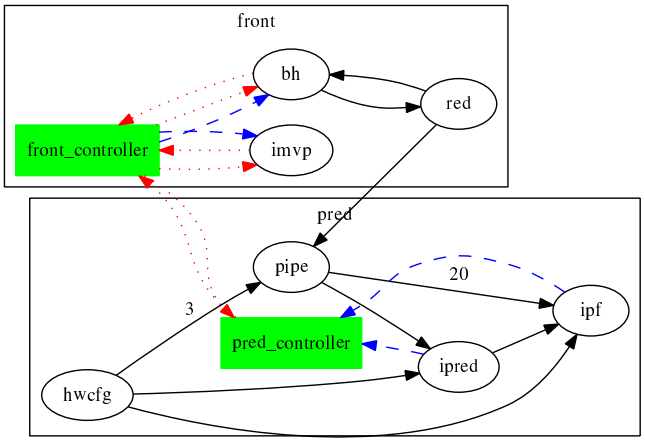

Fig. 4: Graph of Dataflow Actors and Data Dependency of a H.264 Video Decoder

A. Graph-Based Application Architecture

During the design of the P2012 H.264 decoder application, a special focus was put on the module/filter decomposition. Indeed, as filters are intended to be synthesized into hardware accelerators, it was important to optimize their architecture and interactions. The graph presented in Figure 4 is a simplified version of the application architecture, based on its dataflow dependencies. It is composed of two modules, `front` and `pred`. Each module contains a controller (the green rectangular boxes) and a set of filters (the round boxes). The arrows connecting the different entities materialize data-dependencies. We can distinguish three different types: plain-line arrows are pure data links between hardware filters, whereas dotted and dashed arrows correspond to control links, which may be assisted by DMA controllers (the dashed lines).

As we mentioned earlier, this graph is a key element for the application architecture. Thus, it is available through most of the dataflow-related commands. See for instance the command ❶ in the next subsection, where filter and interface names where suggested by the auto-completion mechanism.

In the current implementation, the graph is plotted with Graphviz[6] DOT format and displayed with the system's default image visualizer. However, a debugger with a graphical interface could provide a more interactive view where the graph elements can be directly used to interact with the debugger.

B. Token-Based Execution Firing

As we described in Section IV-B, PEDF dataflow model differs from the traditional models, in the sense that the execution of an actor is first of all conditioned by the triggering of a `fire` event by its module controller. This event can be interpreted as a special *control* token in the dataflow semantic, or as an asynchronous function call in the imperative semantic.

Developers can handle it with the following *catchpoint* command. In this case, it stops the execution when the `WORK` method of Filter `pipe` is triggered:

[6]http://www.graphviz.org/

```
(gdb) filter pipe catch work
```

The execution can also be stopped when a filter has received a given amount of tokens:

```
(gdb) filter ipred catch Pipe_in=1, \
                         Hwcfg_in=1      ❶
(gdb) filter ipred catch *in=1           ❷
```

These two commands stop the execution as soon as Filter `ipred` has received a token in both of its two inbound data links, `Pipe_in` and `Hwcfg_in`. The first command ❶ shows the explicit way, interface by interface, whereas the second one ❷ applies the condition to all the inbound interfaces. The syntax can be applied to any interfaces of a given filter.

C. Non-Linear Execution

During step-by-step execution of filter code, a special attention must be put on dataflow assignment. Indeed, such instructions may enable and trigger the execution of the filter dependent on this data. To accommodate with this eventuality, our debugger offers the `step_both` command, which inserts a "double" breakpoint, at both ends of the link:

```
(gdb) list
220 // push add2dBlock to ipf
221 pedf.io.Add2Dblock_ipf_out[...] = ...;
(gdb) step_both
[Temporary breakpoint inserted after input
    interface `ipf::Add2Dblock_ipred_in']
[Temporary breakpoint inserted after outpt
    interface `ipred::Add2Dblock_ipf_out`]
...
[Stopped after receiving token
    from `ipf::Add2Dblock_ipred_in']
(gdb) continue
...
[Stopped after sending token
    on `ipred::Add2Dblock_MB_out`]
```

In this excerpt, the execution was stopped right before the execution of a dataflow assignment (line 221), where a piece of data is sent though Filter `ipred`'s `Add2Dblock_ipf_out` interface. The command `step_both` instructs the debugger to stop both ends of the execution. The execution flow first reaches the `ipf` filter, on the other side of the data dependency. Then the developer orders to continue the execution and finally the second stop occurs, right after the assignment. The order of these two stops is implementation and architecture dependent.

D. Token-Based Application State and Information Flow

The fluency of the token flow in the overall application is an important concern for correctness and performance. If two filters connected by a data-dependency do not produce and consume tokens at the same rate, the application may stall because of link over/underflow. It can also lead to erratic results if the synchronization of multiple interfaces is not respected.

As an example, the graph presented in Figure 4 shows that the link `pipe` → `ipf` currently holds 20 tokens, which

may indicate a problem in the sending or receiving rate. Link `hwcfg → pipe` contains three tokens, and all the other links are empty. Our debugger can also record and display the *content* of the tokens. This feature may require a significant quantity of memory, thus it has to be explicitly enabled:

```
(gdb) iface hwcfg::pipe_MbType_out record
...
(gdb) iface hwcfg::pipe_MbType_out print
 #1 (U16) 5
 #2 (U16) 10
 #3 (U16) 15
```

In this example, only three messages were recorded, but a communication-intensive filter may quickly generate a large number of tokens, impossible to record efficiently and useless for developers.

Filters can also exhibit clear patterns in their communication behavior. This characteristic can be exploited by our debugger to improve the details about the information flow. Indeed, this allows following a token over several components. However, as this behavior depends of the filter implementation, the debugger cannot automatically figure it out. The developer has to provide it manually to the tool.

For instance, Filter `red` acts as a *splitter*: it receives data from Filter `bh`, processes it and sends the data it generated to all of its outbound interfaces. This can be provided our debugger with the following command:

```
(gdb) filter red configure splitter
```

To better explain the purpose of this feature, we need to place ourselves in a more concrete situation. Let us consider that there is an *observable* error at some point of the execution. With the help of the mechanisms previously described, the developer stops the execution as close as possible to the error trigger. For instance in Filter `pipe`, after receiving a token from Interface `Red2PipeCbMB_in`:

```
(gdb) filter pipe catch Red2PipeCbMB_in
...
[Stopped after receiving token
    from 'pipe::Red2PipeCbMB_in']
```

At this point, the developer ensures that the situation is actually erroneous and tries to understand where the fault came from. Token path becomes useful:

```
(gdb) filter pipe info last_token
#1 red -> pipe (CbCrMB_t){Add=0x145D,...}❶
#2 bh  -> red  (U32)      127             ❷
```

We can see that the last token was received from Filter `red` (step ❶), with a given value. If this value is incorrect, it means that the error arrived from Filter `red`. Step ❷ helps understanding the conditions in which this token was produced: after receiving an integer token (127) from Filter `bh`. To complete this information, further details about the filter state can be recorded, such as attribute values.

Recording token contents may appear excessive when querying the content of a single link (it could be directly read from the framework memory), however it becomes mandatory when we want to follow its path over multiple actors.

E. Two-level Debugging

We are aware that a *dataflow* debugger may not be enough to locate and understand all the possible problems which can occur during the execution of a dataflow application. For this reason, a traditional, full-flavored GDB is always available during the debugging session. Our debugger extension only handles the dataflow-specific commands, and the underlying GDB manages the rest of the debugging environment. This means that when the execution is stopped, like in a previous example, the developer can ask our extension to display the last token received and then use GDB to analyze its C structure and inner content:

```
...
[Stopped after receiving token
    from 'pipe::Red2PipeCbMB_in']
(gdb) filter print last_token
$1 = (CbCrMB_t){Addr=0x145D, ...}
(gdb) print $1
$2 = { Addr = 0x145D,
       InterNotIntra = 1,
       Izz = 168460492, ... }
```

F. Qualitative Analysis

Debugging such an application with current debuggers, like a standard GDB, would be quite challenging. The current P2012 simulator relies on user-level threading for the implementation of the different processors, which are usually not correctly handled (only a few kernel-level threads are detected). But even with user-level thread support, the tools would not be able to distinguish the framework (or the simulator) code from the application itself. Hence, developers would have to look for the relevant information in the middle of all the runtime libraries. These pieces of information are frequently mangled: for instance, filter `Ipf WORK` method correspond to the symbol `IpfFilter_work_function` whereas controller `pred_controller WORK` method is `__component_PredModule_anon_0_work`. Only information about the filter *currently* running (if any) could be retrieved at a given time.

Some other information is just unavailable, like the architecture graph. Hence, developers would have to refer to architecture description files and apply at their best this information to the current runtime context. Another example is the number of tokens accumulated in a given data link. This figure can be manually computed, with breakpoints set at both ends of the link and a pen and paper count, but this is tedious and error-prone.

In order to further validate the efficiency of our approach, it would be interesting to measure the time required to locate different kinds of bugs, for instance related to the dataflow architecture, the token passing or the application algorithm itself. These results could be compared against more common methods like source-level debuggers or execution traces analysis. It would also be interesting to study how our tool can help during the early development of complex applications, when the entire architecture is being assembled.

In the following section, we discuss the related work focusing on the debugging of complex applications and programming models.

VII. Related Work

Our approach can be related to different research areas. In the first part of this section, we introduce a study focusing on task-based parallel application debugging. After that, we present how a similar debugging approach has been applied to component-oriented applications. Then, we describe the programming environment of the StreamIt language, which features a dataflow debugger. Finally, we introduce a study from 1988 proposing a methodology for debugging dataflow machines. To our knowledge, this is the only research study available related to dataflow debugging.

A. Task-Based Parallel Application Debugging

Temanejo [16] is a debugger for the task-based StarSs programming models. These models, similarly to dataflow programming, put an important focus on the data dependencies of the different task which form the application. The runtime framework of such applications is in charge of executing the different tasks in parallel while preserving the correct ordering. The debugger is able to follow and reconstruct the graph of task executions. This graph is useful to developers as its structure is not known at compile time. Additionally, the debugger also allows developers to control task executions. It can change their priority, block them or measure their duration. Temanejo can also launch a debugger (i.e. GDB) upon specific events, like the beginning of a task execution.

Although the tool offers interesting representations, it currently lacks advanced interactive debugging commands. Indeed, Temanejo is totally decoupled from GDB, hence neither of the tools can benefit from the other. GDB remains at source level, and Temanejo cannot provide any language-related information.

B. Component-Based Application Debugging

Our previous article [17] is closely related to this work. It describes how the idea of enhancing interactive debuggers with the help of programming models can be applied to component-based software engineering. In this model, components are standalone computational entities. They are meant to perform a formally specified task, and it should be possible to reuse a component in another application without any modification. Components provide services based on the data supplied on their input interfaces and serve the response on output interfaces. Thus, a component application can be built just with the inter-binding of the relevant components.

The proposed approach explains how debuggers can be tailored to component-based application. This includes the ability to set breakpoints on component execution and interface exchanges, but also on messages being transmitted over different components.

Debugging these applications is challenging because of their complex and dynamic architecture. In opposition with dataflow applications, the interconnection of component interfaces can be changed at runtime, for instance to adapt the application to runtime constraints. Components may also be executed in parallel to improve performance. For similar reasons to those described in Section II, traditional tools are not adapted to debugging component-based applications either. However, the concept of token/message is not as important in component-based software engineering as it is in dataflow models. The information transmitted from one component to another is primarily a service request or response. In dataflow, as we described in Section II and illustrated in Section VI, they hold an important part of the application state. The non-linearity of the execution is also more prominent in dataflow application debugging.

C. StreamIt Debugger

StreamIt is a programming language for high-performance streaming applications [10], which shares properties with the dataflow models and in particular synchronous dataflow. The StreamIt Development Tool [18] provides a graphical environment to assist the coding and debugging of stream applications. The environment provides a graphical debugging tool adapted to StreamIt programming language. Similarly to our approach, their debugger takes into accounts StreamIt specificities and allows developers to interact with a graph representation of the application. They also display information about the communication channels, such as the token they hold or stream statistics.

So their tool appears to solve some of the challenges we describe in Section II, like the graph-based architecture or the token-based execution filtering and application state. However, they did not focus on the problems of dynamic dataflow, as StreamIt is a synchronous dataflow language: all data must be received before execution and the date of actors input and output tokens is defined at compile time. So a substantial part of the interactive debug challenges is avoided. Finally, it appears that they do not address the issues of the non-linear execution or following the information flow.

D. Dataflow Application Debugging

In an article from 1988, Wahl *et al.* described a debugging methodology for distributed dataflow program [19], which shares similarities with our approach. However, at that time, they still had faith in *real* dataflow machines, with non von Neumann architectures. Their methodology points out that they wanted "to allow the user to debug a program in a way that is close to his or her conceptual model of the program." They also mentioned that "at the same time, the user must be supplied with a set of debugging commands that includes those with which he and she is familiar with in the context of uni-processor von Neumann machines." We intimately share these convictions, which drove our work for this paper.

The methodology they proposed is close to our contribution, however they only skimmed over the interactive debugging aspect and they did not provide details about its actual usage. Our approach extends and deepens this specific aspect.

Furthermore, they explained that their methodology relies on a dataflow machine simulators, which have to be modified to support debugging. This requirement strongly limits the scope of their work, as their debugging module would have to be implemented at hardware level. Our approach does not face this problem. Indeed, the debugger only interacts with the software dataflow framework.

VIII. Conclusion and Future Work

In this article, we introduced our new approach to debugging dataflow applications interactively. We noticed some flaws in the available tools, which are not adapted to the debugging of current dataflow applications. These applications rely on frameworks written on top of imperative or object-oriented languages, but traditional debuggers only operate at source-language level and do not account for the dataflow abstractions introduced by the programming model.

We presented the main issues faced by developers when they try to debug such applications, which are related to the graph structure of the architecture, the token-based actor firing and application state, the non-linear execution and the complex flow of information over the different actors. We described our contribution to lighten these difficulties, an approach where debuggers are able to provide developers with high-level abstractions related to the dataflow programming model. Based on the detection and interpretation of key events in the execution of the dataflow framework, our approach allows the debugger to take into account the data-dependency graph and dynamically track the tokens flowing into the application. Thus, our approach offers a better overview of the actual state of dataflow applications. It also allows setting breakpoints on specific actors, on their data dependencies or directly on tokens.

Then, we described how we applied this approach to *Predicated Execution Dataflow*, an industrial parallel dataflow framework for the MPSoC system *Platform 2012*, developed by STMicroelectronics and CEA. We presented the different parts of the framework API, along with the associated debug capabilities. We also detailed how we developed a proof-of-concept debugger as an extension of GDB, the free debugger of the GNU project. GDB is frequently used to debug applications running on embedded systems. Based on its Python API, we introduced the concept of *function breakpoints* as an extension of the traditional breakpoints. These breakpoints are tailored to the framework function they monitor and update the debugger internal representations each time their function is executed.

Finally, we presented a case study that illustrates the debugging of a H.264 video decoding application. We confronted our tool with the challenges presented earlier and we highlighted how our approach can facilitate developers debugging task, in comparison with the available tools.

Interactive debugging of complex applications based on their programming model appears to be a promising direction to lighten the bug tracking hassle. In future work, we will investigate how the idea of leveraging the programming model to improve the debugging experience can be applied to different models, and how visualization can help developers to better understand the details of the execution. We expect our debugger to be able to easily encompass new models, thanks to a generic code base. Using GDB as the low-end debugger should also help other framework developers to adapt our tool to their environment.

References

[1] E. Bezati, M. Mattavelli, and M. Raulet, "RVC-CAL dataflow implementations of MPEG AVC/H.264 CABAC decoding," in *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, United Kingdom, 2010.

[2] H. Yviquel, E. Casseau, M. Wipliez, and M. Raulet, "Efficient multicore scheduling of dataflow process networks," in *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, Lebanon, Dec. 2011.

[3] J. Eker and J. W. Janneck, "Cal language report specification of the cal actor language," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M03/48, 2003. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2003/4186.html

[4] E. S. Jang, J. Ohm, and M. Mattavelli, "Whitepaper on reconfigurable video coding (RVC)," http://www.chiariglione.org/mpeg/technologies/mpbrvc/index.htm, Antalya, Turkey, 2008.

[5] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar, "Mapping a data-flow programming model onto heterogeneous platforms," in *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, ser. LCTES '12. ACM, 2012.

[6] T. D. R. Hartley, E. Saule, and U. V. Çatalyürek, "Automatic dataflow application tuning for heterogeneous systems," in *HiPC'10*, 2010.

[7] W. M. Johnston, J. R. P. Hanna, Richard, and J. Millar, "Advances in dataflow programming languages," *ACM Comput. Surv*, vol. 36, 2004.

[8] S. Bhattacharyya, E. Deprettere, and T. B.D., "Dynamic dataflow graphs," in *Handbook of Signal Processing Systems*. Springer, 2012.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '95. New York, NY, USA: ACM, 1995.

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

[11] STMicroelectronics and CEA, "Platform 2012: A Many-core Programmable Accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology," http://www.cmc.ca/WhatWeOffer/ /media/WhatWeOffer/TechPub/20101105_Whitepaper_Final.pdf, 2010.

[12] T. Wiegand, W.-J. Han, B. Bross, J.-R. Ohm, and G. J. Sullivan, "WD2:Working Draft 2 of High-Efficiency Video Coding," *JCTVC-A200*, Apr. 2011.

[13] The Gnu Project, "GDB," http://www.gnu.org/software/gdb/.

[14] Free Standards Group, "The DWARF debugging standard," http://dwarf.freestandards.org/, Jun. 2010.

[15] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, Jul. 2003.

[16] S. Brinkmann, J. Gracia, C. Niethammer, and R. Keller, "TEMANEJO - a debugger for task based parallel programming models," in *International Conference on Parallel Computing*, 2011.

[17] K. Pouget, V. Marangozova-Martin, M. Santana, and J.-F. Mehaut, "Debugging component-based embedded applications," in *Proceedings of the 15th International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '12. ACM, 2012.

[18] K. Kuo, "The streamit development tool: A programming environment for streamit," M.Eng. Thesis, Massachusetts Institute of Technology, Cambridge, MA, Jun 2004.

[19] N. Wahl and S. R. Schach, "A methodology and distributed tool for debugging dataflow programs," in *Software Testing, Verification, and Analysis, Proceedings of the Second Workshop on*, jul 1988.