A Novel Approach for Interactive Debugging of Dynamic Dataflow Embedded Applications

(short version)

Kevin Pouget[*][†]
Miguel Santana[*]
[*]STMicroelectronics
Crolles, France

Patricia Lopez Cueva[*][†]
Jean-François Mehaut[†]
[†]University of Grenoble, CEA, LIG
Grenoble, France

ABSTRACT

In this paper, we propose a new approach for source-level dataflow debuggers. Going beyond their long-established ability to support sequential programming languages, we describe the functionalities a debugger 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.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Code inspections and walk-through*; D.3.2 [**Programming Languages**]: Dataflow languages

Keywords

Dataflow, Debugging, Embedded, Parallel, MPSoC

1. INTRODUCTION

Since the 1970s/1980s, the dataflow programming model has been developed as an alternative to the conventional paradigms based on "von Neumann" processors [4]. This model was explicitly designed to exploit parallel architectures, and tries to avoid the main bottlenecks of von Neumann hardware: the global program counter and the global updatable memory. In this model, the key focus is shifted from the stream of instructions being executed (i.e.: moving the program counter) towards the dependencies between the data. In a word, this means that an instruction is not executed when the program counter reaches it, but rather when its operands are ready.

In this paper, we primarily focus on dynamic dataflow languages [2], which provide a high expressiveness and limit the constraints imposed to the 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 dynimic streams. So the dataflow languages targeted by our approach would usually not support model analysis and verification, but rather emphasized the programability and the ease of modeling.

Multimedia applications are good candidates for a dataflow based implementation. Indeed, stream-processing algorithms naturally exhibit dataflow graphs, which simplifies the application design and task implementation. Recent publications like [1, 7] show successful demonstrations of the efficiency of dataflow languages to implement video decoders.

Executing a dataflow application under the control of an interactive debugger would allow developers to clearly understand the subtleties of the execution flow. Wellcrafted 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. Furthermore, these tools can be used to monitor and profile applications. This allows developers to have realtime feedback about the actual application execution, which may depend of the underlying platform or the local environment.

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 the debuggers.

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

2. MOTIVATIONS

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.

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

Token-Based Execution Firing The execution of a dataflow actor can only start when the required input data-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 actors connected to this port become executable. Semantically, the execution flow is forked and follows not only its normal stream but also the different outgoing arcs of the graph.

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

In the next section, we present the debugging approach we designed to help developers to locate more easily the problems of dataflow applications.

3. HOW TO DEBUG A DATAFLOW APPLICATION?

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 in and outbound connections, tight together into a graph of data dependencies.

Stop the Application Before the Error The debugger should leverage the additional events introduced by the dataflow model to provided 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 data-tokens.

Execute the Application Step-by-Step The exchanges and interactions between the actors should be exploited to improve step-by-step execution. It should be possible to navigate in the application architecture based on the dataflow graph of 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.

Inspect 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.

The next section and Section 5 present a proof-of-concept debugger implementing this approach.

4. DEBUGGING FEATURES FOR AN EMBEDDED DATAFLOW FRAMEWORK

In this section, we describe our proof-of-concept debugger and the features it provides, which implement the approach described in Section 3. We focus on the embedded system *Platform 2012* (P2012) [5] and its dataflow framework *Predicated Execution DataFlow* (PEDF).

P2012 is a low consumption, parallel and embedded platform research project developed by STMicroelectronics and CEA. It targets high-definition audio and video processing. PEDF is an hybrid dataflow framework designed to exploit the heterogeneity of P2012 architecture. It provides a *structure* dataflow model, as presented in [4]. PEDF also offers more advance control features allowing module controllers to modify the behavior of the dataflow graph during its execution (based on a set of predicates) or run some parts of the graph at different rates. PEDF defines three classes of entity:

Filter It is a computing entity, corresponding directly to the *actors* of the dataflow model. Filters have in 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, 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 in and outbound data links, corresponding to the unconnected arcs of the inner graph.

In the remaining of this section, we describe how we can design a dataflow application with PEDF, as well as the debugging features associated with each of the steps.

4.1 Designing the Data-Dependency Graph

PEDF dataflow graph is built with Mind[1] architecture compilation tool-chain, augmented with PEDF-specific annotations. Mind provides a description language to specify filter's architecture and interfaces. Module entities are in charge of defining the inter-filter bindings.

Our contribution: 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.

4.2 Filters Data Processing

PEDF filters implement the core data processing tasks of the application. Filters must define a `WORK` method, implementing one step of the processing.

Our contribution: 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 data-token through a dependency by counting the tokens pushed in and out of the link. As the model and the implementation ensure

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

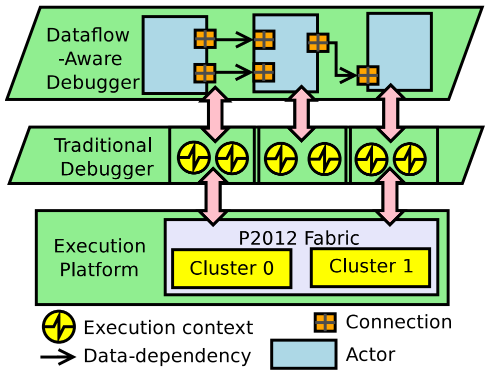

Figure 1: Two-level debugging of an MPSoC platform.

that the data order is preserved, we can stop the execution at the right location in a deterministic way.

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

5. EXTENDING GDB TO SUPPORT DATAFLOW DEBUGGING

GDB is the debugger of the Gnu project [6]. 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 1 presents a schematic representation of a traditional debugger like GDB connected to an MPSoC platform, as well as our dataflow-awareness extension. The top part of the figure schematize the internal representation used to model the dataflow application:

Our runtime-information capture mechanism relies on internal function breakpoints set at the entry and exit 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.

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 in and outbound connections.

Token objects are created and transmitted over the different debugger dataflow entities. 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. They are associated with the simulator entity responsible for the data transfer. These objects produce or consume Token objects when the corresponding events are 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 the Token objects required by the 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 [3] 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 a specific recompilation. On the other side, we could exploit compiler-generated information to enrich our model and improve the overall performances.

6. CONCLUSION AND FUTURE WORK

In this article we introduced our new approach to debugging dataflow applications interactively.

We presented the main issues faced by developers when they try to debug such applications, which are related to the graph-based architecture, the token-based actor firing and the non-linear execution. 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.

Interactive debugging of complex applications based on their programming model appears to be a promising direction to reduce 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.

7. 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*, pages 207 –213, United Kingdom, 2010.
[2] S. Bhattacharyya, E. Deprettere, and T. B.D. Dynamic dataflow graphs. In *Handbook of Signal Processing Systems*. Springer, 2012.
[3] Free Standards Group. The DWARF debugging standard. `http://dwarf.freestandards.org/`, 2010.
[4] W. M. Johnston, J. R. P. Hanna, Richard, and J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv*, 36:1–34, 2004.
[5] STMicroelectronics and CEA. Platform 2012: A Many-core Programmable Accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology, 2010.
[6] The Gnu Project. GDB. `http://www.gnu.org/software/gdb/`.
[7] 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*, pages 198 – 203, Liban, Dec. 2011.