Composite Data Types in Dynamic Dataflow Languages as Copyless Memory Sharing Mechanism

Aurelien Bloch[1(✉)], Endri Bezati[2], and Marco Mattavelli[1]

[1] EPFL SCI STI MM, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
{aurelien.bloch,marco.mattavelli}@epfl.ch
[2] EPFL VLSC, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
endri.bezati@epfl.ch

Abstract. This paper presents new optimization approaches aiming at reducing the impact of memory accesses on the performance of dataflow programs. The approach is based on introducing a high level management of composite data types in dynamic dataflow programming language for the memory processing of data tokens. It does not require essential changes to the model of computation (MOC) or to the dataflow program itself. The objective of the approach is to remove the unnecessary constraints of memory isolations without introducing limitations to the scalability and composability properties of the dataflow paradigm. Thus the identified optimizations allow to keep the same design and programming philosophy of dataflow, whereas aiming at improving the performance of the specific configuration implementation. The different optimizations can be integrated into the current RVC-CAL design flows and synthesis tools and can be applied to different sub-networks partitions of the dataflow program. The paper introduces the context, the definition of the optimization problem and describes how it can be applied to dataflow designs. Some examples of the optimizations are provided.

Keywords: Dynamic dataflow programs · RVC-CAL · Shared memory · Composite data types

1 Introduction

In recent years the difficulties of CMOS technologies to scale-up by increasing the processors frequency, led the processor research and industry to investigate the scale-out by increasing the number of processing units using multi-core, many-core architecture combined with different memory architectures and possibly programmable HW elements building heterogeneous platform. However, these new platforms require software developments to be adapted to the specific platform architecture to take full advantage of the potential hardware parallelism. Such constrains introduce new challenges to software design such as the

© Springer Nature Switzerland AG 2019
J. M. F. Rodrigues et al. (Eds.): ICCS 2019, LNCS 11540, pp. 717–724, 2019.
https://doi.org/10.1007/978-3-030-22750-0_70

portability of applications across platforms or the ability for the programmer to properly abstract and correctly design algorithms using imperative programming languages that take advantage of the processing power available in terms of massive parallelism.

High-level dataflow programming languages are well recognized to be able to overcome those issues [7]. They are used in several fields for modeling data-driven algorithms and in many application areas such as video and audio processing, bioinformatics, financial trading and packet switching. Their essential feature is to be able to abstract parallelism regardless of the targeted hardware platform [4].

The nice properties of dataflow MoC are valid for any type of memory architectures, ranging from the most restricted architectures, for which each memory component is only accessible by a single computational unit, to full shared memory architectures, for which memory is freely accessible by any processing element. However, the performance of dataflow programs on less constrained platforms using the same implementation assumptions of more constrained platforms may result sub-optimal.

Indeed to guarantee the absence of data races in highly parallel platforms, dataflow programs relies on the concept of a full memory isolation for each computational kernel called *actors*. This assumption which provides the guarantees of correctness of the executions for any mapping of the network of actors on any platform, may lead to memory inefficiencies when some actor partitions (i.e. dataflow network partitions) share some or all memory elements.

The paper presents a new approach for data sharing between actors of a dataflow network that reduces the amount of data transfers without changing the model of computation or the semantic of a given application, but only changing the data transfers implementation. The cases for which the communication buffers can be implemented more efficiently, by removing memory isolation constraints for specific partitioning of the dataflow network, are first identified and then three different optimized implementation solutions are defined.

The paper is structured as follows: Sects. 2 and 3 present the context of the dataflow model of computation and compiler. Section 4 provide an overview of the related work. Section 5 presents the design proposition of this new approach, discusses application cases and present different implementations. Finally, Sect. 6 concludes the paper and outlines other directions for further investigations and more effective optimizations.

2 Dataflow Model of Computations

A dataflow program is composed of a (hierarchical) directed graph called, *network*, where each node is an actor and each directed edge is a lossless and order preserving communication channel called *buffer*. These buffers are used to asynchronously transmit atomic data packets called *tokens* between actors. Different dataflow MoC have been defined. A common characteristic is the fact that actors do not have access to a shared memory allowing parallel executions without data race.

Dynamic Process Network (DPN) is one of the most expressive MoC where the actors consumption and production of tokens can vary according to the nature of the available inputs and their internal states [8]. This flexibility is well suited for designing, real-world and complex algorithms at the cost of facing more challenging analysis and optimization problems.

In this work RVC-CAL, a dataflow programming language standardized by the MPEG committee, which fully captures the behavioral features of the DPN model of computation [6] is used. In RVC-CAL, each actor can contain a set of atomic firing functions, called *actions* and internal state variables. When an actor is executed, only a single action can fire at a time. The firing of an action depends on the input availability and values of its tokens, the output available spaces, and the internal state of the actor.

3 Dataflow Compiler

The Open RVC-CAL Compiler (Orcc) is an open-source Integrated Development Environment (IDE) based on Eclipse and dedicated to dataflow programming [1]. It is the compiler used in this work and is mainly a source-to-source compiler that translates the RVC-CAL application into another programming language depending on the backend selected during compilation.

In this work, the Xronos backend [3] is used. It generates from an RVC-CAL description, a C++ implementation with all the necessary library dependencies. The objective is to improve the quality of the generated code by minimizing the overall amount of memory copies by providing when compatible with the dataflow MoC, specific memory sharing mechanism across actors. The introduction of such optimizations can improve the performance of implementations for specific partitioning and scheduling configurations. It can also provide an extension of the design exploration space and yield new scheduling, partitioning and buffer size design points for the design space exploration framework, TURNUS [2].

4 Related Work

A first approach for solving this problem of memory sharing across actors has been presented in the same design context [9]. it is proposed to have actual shared variables, breaking the encapsulation of actors. To do so, internal variables that are shared among multiple actors are tagged with *@shared*. In addition, a Shared Memory Controller (SMC) along with a specific protocol were designed for access synchronization. The solution has shown the benefice of relevant performance gains due to the instant access to the shared memory once granted access to it and low overhead of the synchronization protocol. The drawback of the solution is that designers have to modify the model of computation and break the principle of memory encapsulation of actors to allow to share their internal states. This means than the compiler cannot guarantee that the generated code is free of data races and that the validity solution has to rely on the designer knowledge. This

brings the solution closer to what it is obtained in more traditional settings in which parallelism is obtained by introducing additional synchronization barriers to general purpose imperative languages.

There also exists in the literature some work done such as [5] but they mostly target Synchronous Dataflow (SDF) MoC.

5 Design Proposition

The proposed solution aim at optimizing the performance of the implementation of the generated code for data communication between actors, when targeting a hardware platform where actors partitions are mapped to processing units having access to a common physical memory. In doing so, the model of computation remains unchanged keeping all the properties guaranteed by construction by applying code synthesis and compilation.

5.1 Composite Data Types

The current implementation of the code generation for the standard version of RVC-CAL, processes inter-actor communications by instantiating buffers using primitive data types. A consequence of this relatively low granularity of data transfer may impact the performance of the application compared to design using other models of computation. Furthermore depending on the application some amounts of data might need to be copied over different buffers through the dataflow network even if not all data is relevant for the processing of an actor internal algorithm. To illustrate this fact an example can be useful. Consider an action that produce five tokens to its output buffer at each firing. Figure 1a represents the content of such a buffer after two firings.

Currently, when synthesizing code for such simple program, the Orcc compiler generates a loop, that copy the tokens from the internal memory of the actor to the memory of the output buffer. This makes the amount of copy to the same physical memory for each firing linearly proportional to the production of tokens.

The approach described in this paper is to introduce composite data types as objects manipulated with pointers, which would allow fewer data copies. Lists (arrays) are considered here an example. Figure 1b shows how the status of the buffer might look like, when list to represent data in the same example of two firings of five tokens each are used. It can be observed that instead of having a loop copying the ten primitive values, it is only necessary to copy the two pointers to the corresponding memory chunks. This makes the amount of data copy proportional to the number of moved chunks instead of the number of tokens. This approach keeps the same philosophy for avoiding data races like what Orcc is currently implementing. In fact, it does not propose to modify the model by offering shared memories between actors and still can rely on buffers to synchronize communication between them.

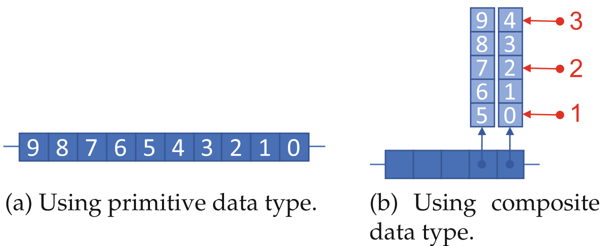


Fig. 1. Buffer filled with two firings of five primitive tokens each.

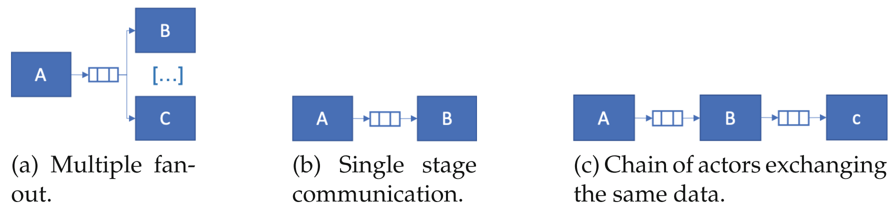
5.2 Buffer Identification

In this section the different cases where this optimization can be applied are identified. First of all, this proposition can only be beneficial for actions that produce multiple tokens in a single firing. This is the case for example, when actions uses a *repeat* expression. In addition to that, three different configuration cases can be identified.

The first one is when a buffer has multiple fan-out as shown in Fig. 2a. In this case, it is necessary to duplicate the composite data to avoid the data race. This is due to the fact that pointers are used, to be copied at the place of the data. If the data is not duplicated, each actor has a reference to the same piece of data, which might yield data races problems. This configuration should only improve performances to a fraction inversely proportional to the fan-out numbers, as only the first actor will have access to the original data whereas the others need a copied version.

The second case is when a list is transmitted only between two actors as shown in Fig. 2b. In this case the proposition will not result in any performance improvement as it is already optimized by the current implementation of the Orcc compiler. Indeed, instead of generating the tokens to a local array and then copying this data to the output buffer memory, the compiler generated code uses a pointer to the buffer memory and store the tokens directly there when they are available. In the same way, the consumer actor (actor B in the schema) use a pointer to the buffer memory to directly process the data read, instead of first copying them locally and then processing them. This optimization prevent the introduction of list to bring performance improvement in this particular case.

The third case is the one where the use of composite data types can provide the higher performance gain. It can be identified when multiple actors process the same composite structure of data. An illustration of this case is depicted in Fig. 2c. The achievable performance improvement is proportional to the size of the chunk of data and to the number of stages the same data is processed by a different actor.



(a) Multiple fan-out.

(b) Single stage communication.

(c) Chain of actors exchanging the same data.

Fig. 2. Buffer identification.

5.3 Implementations Discussion

In this section the implementation challenges that need to be addressed so that the Orcc compiler is able to generate valid C++ code implementing the proposed optimization is discussed.

Fully Dynamic Solution. In this case the consumer can read list of token of any sizes regardless of the size of the emitted list. This solution is the most general. It means that actors can read chunks of data smaller than the actual list and even read chunks that span across two physical data allocations. An illustration of this implementation is depicted in Fig. 1b, where the pointers returned to the reader at each firing are the red arrows. In this example, the read size is 2 and it can be observed that the third read (composing or the numbers 4 and 5) span across the two continuous memory allocation. This fully flexible settings raises two implementation challenges.

One is that since a single continuous allocation can be linked to from different actors during the runtime of the application, it can be difficult to pinpoint when this memory chunk can be released especially in environment with no native garbage collection like in C++. For that, we used *std::shared_ptr* that offers a kind of autorelease mechanism once a memory chunk is no longer referenced by any actors or buffers.

Another technicality to be solved is the need to transparently handle reads that can reach multiple memory chunks. For this purpose a custom proxy class has been developed that is returned, instead of a direct pointer to a memory block, that act as a middle man and handles reads through an indirection, which can affect performance and prevent processor vectorization. This side-effect might be somewhat mitigated if the proxy is used only in the corner cases where it is necessary and if the memory allocator used is tuned so that most consecutive chunks would be allocated consecutively in memory removing the need also in these cases. A custom allocator that would be used explicitly and provided with network specific information to make the use of consecutive allocation more frequent could also be considered.

Semi Dynamic Solution. This case is a constrained version of the previous more general case, where the consumer can only read at each firing a number

of tokens that is a divider of the size of the produced list. This constraint is equivalent to impose that a read would never reach across two different memory allocation, which removes the need for any proxy or special allocator, while keeping some amount of flexibility. The difficulty here is to be able to guarantee that this property is always satisfied to allow the safe usage of this implementation solution.

Static Solution. In the third case, the reader has always to consume an entire chunk of data seen in this case as an object. This is the most restrictive configuration, but also the simpler to implement. The need to use memory releasing mechanisms can be avoided as only a single actor or buffer can reference an object at any time allowing for an explicit freeing of memory from the actor itself when the chunk is no longer needed.

6 Conclusions

This paper presents a new approach for the synthesis of efficient implementations of data sharing between actors of a dynamic dataflow networks in the context of the RVC-CAL programing language. The approach introduces composite data types as a ways to avoid data copies whenever possible. It shows for which buffer configuration the optimization solutions can be beneficial and specifies the different ways of implementing them in C++ depending on the flexibility, given to the rate at which an actor can consume data.

Future work considers automatizing the selection by the Orcc compiler of the generated solution depending on the configuration (i.e. Buffer configuration, Network partition, targeted platform). Moreover, this would enable more design point to be considered by the TURNUS framework.

References

1. Orcc. http://github.com/orcc/orcc. Accessed Apr 2019
2. Casale-Brunet, S.: Analysis and optimization of dynamic dataflow programs. Technical report EPFL (2015)
3. Casale-Brunet, S., Bezati, E., Mattavelli, M.: Programming models and methods for heterogeneous parallel embedded systems. In: 2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), pp. 289–296. IEEE (2016)
4. Castrillon, J., Leupers, R.: Programming Heterogeneous MPSoCs. Tool Flows to Close the Software Productivity Gap. Springer, Switzerland (2013). https://doi.org/10.1007/978-3-319-00675-8
5. Desnos, K., Pelcat, M., Nezan, J.F., Aridhi, S.: Distributed memory allocation technique for synchronous dataflow graphs. In: 2016 IEEE International Workshop on Signal Processing Systems (SiPS), pp. 45–50. IEEE (2016)
6. Eker, J., Janneck, J.: CAL language report: Specification of the CAL Actor Language. Technical Memo UCB/ERL M03/48, Electronics Research Laboratory, University of California at Berkeley, December 2003

7. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) Information Processing, pp. 471–475. North Holland, Amsterdam, Stockholm, Sweden, August 1974
8. Lee, E., Parks, T.: Dataflow process networks. In: Proceedings of the IEEE, pp. 773–799 (1995)
9. Modas, A., Casale-Brunet, S., Stewart, R., Bezati, E., Ahmad, J., Mattavelli, M.: Shared-variable synchronization approaches for dynamic data flow programs. In: 2018 IEEE International Workshop on Signal Processing Systems (SiPS), pp. 263–268. IEEE (2018)