Co-optimizing Dataflow Graphs and Actors with MLIR

Pedro Ciambra, Mickaël Dardaillon, Maxime Pelcat, Hervé Yviquel

Co-optimizing Dataflow Graphs and Actors with MLIR

Pedro Ciambra[1]*, Mickaël Dardaillon†, Maxime Pelcat† and Hervé Yviquel*
*Institute of Computing, University of Campinas (UNICAMP), Campinas, Brazil
Email: p137268@dac.unicamp.br, hyviquel@unicamp.br
†INSA Rennes, IETR, UMR 6164, Rennes, France
Email: firstname.lastname@insa-rennes.fr

***Abstract*—Dataflow programming is considered a good solution for the implementation of parallel signal processing applications. However, the strict separation between kernel and coordination codes limits the variety of possible optimizations and the compatibility with state-of-the-art compiler frameworks. We present a prototype static dataflow compiler, built with the LLVM MLIR framework, that overcomes these limitations and enables a previously impossible combination of optimization strategies that leverages information from the dataflow topology. Initial results show 30% wall time improvement and 53% memory usage improvement on a video processing workload.**

***Index Terms*—compiler, dataflow, MLIR, optimization, dead code elimination, IaRa, dialect, co-optimization**

I. Introduction

Dataflow Models of Computation (MoCs) have demonstrated their performances in a large number of signal processing systems studies [1]. Dataflow networks have however not been widely adopted by the compilation community. While a number of tools have been developed to aid the creation and optimization of dataflow algorithms, these projects have limited interoperability, amongst each other and with the outside compiler ecosystem. Meanwhile, the popularization of domain specific languages such as neural network representation formats and other highly parallel programming frameworks has brought many advances in compiler research [2], yet to be leveraged by the dataflow community.

The go-to strategy for existing high-performance dataflow projects has been source-to-source compilation. Instead of directly generating machine code, these projects rely on an existing framework for low-level code generation, usually a C compiler such as the GNU Compiler Collection (GCC) or Clang/LLVM. This strategy presents a couple of issues. One disadvantage is having to commit to the level of abstraction of C, which incurs some loss of information that could otherwise be useful in optimization strategies; another problem is the difficulty to perform transformations in the code once it has been generated, as few C compilers expose their intermediate representations in an accessible way.

In this paper, a dataflow compilation solution is proposed, aiming at co-optimizing kernel and coordination code by using MLIR [3]. In the semantics of MoCs, a coordination language glues together self-contained kernels ("actors" in the dataflow terminology) to which portions of computation are delegated [4]. MLIR is a recent effort by the LLVM project that focuses on modular, intercompatible Intermediate Representations (IRs) to provide compatibility and code reuse between projects. By developing a new IR for dataflow networks that is compatible with the state-of-the-art of compiler infrastructures, we aim to demonstrate the potential for new strategies and foster dataflow methods in the compiling community.

For compile time optimization, Synchronous Dataflow (SDF) and other static dataflow MoCs are of particular interest, since their properties allow for many optimization strategies and expose different forms of parallelism; therefore, they stand to gain the most from traditional compilation techniques. Non-static dataflow MoCs such as PiSDF [5] and Dynamic Dataflow [6] may also benefit from a dataflow compilation framework, as they have conceptual similarities with static models. By making use of the MLIR project to simultaneously access and modify different layers of abstraction, one can develop optimization strategies based on the notion of compilation passes.

The main contribution of this paper is the design of a co-optimization pass that allows for the combination of two previously incompatible strategies, showing experimental gains in both time and memory consumption. To achieve this, we present *IaRa*, a full compilation pipeline within the MLIR framework that includes a Dataflow Interchange Format (DIF) parser [7], a custom IR for dataflow networks, and an integration with an MLIR C frontend called Polygeist [8] for transformation of kernel implementations provided in the C language.

To the extent of our knowledge, this work is the first to concentrate on the co-optimization of dataflow graph structure and kernel code.

In Section II, the semantics of SDF are discussed, as well as the advantages of using MLIR for compiler research. In Section III, we present the proposed compilation pipeline, including a novel IR for dataflow networks, and an approach for co-optimized Dead Code Elimination (DCE). The contribution is compared to related work in Section IV. Section V discusses the experimental improvements in performance and memory usage achieved by co-optimizing an example application. Finally, in Section VI we conclude the paper by discussing further potential applications of MLIR and related projects.

[1]CNPq Scholarship - 133933/2020-2

II. Background

A. Dataflow MoCs

Dataflow is a family of Models of Computation [9] that relies on describing algorithms as directed graphs, where vertices, referred to as "actors", represent side-effect-free units of computation that consume and produce data units ("tokens"), and edges represent the connections between actors in the form of first-in-first-out (FIFO) data structures. An actor "fires", consuming an amount of tokens from its input edges, applying some transformation on them, and inserting the results into the outputs. The source code for this computation is referred to as "kernel", and is traditionally provided in the C language.

A dataflow framework, thus, has the task of determining when to fire each actor (scheduling), and where to allocate the memory for the processed tokens (bufferization), using only information from the graph topology. The code generated by this process is referred to as "coordination" code. This separation of kernel and coordination code allows for a flexible design process while maintaining high performance and parallelism.

Static Dataflow [9], [10] is a subset of dataflow MoCs with the property of fixed production and consumption rates. This allows a compiler to, ahead of execution, find an order of firings that is periodic and data-independent, and also determine size and location in memory for every produced token in a period. This removes the need for a runtime and allows for optimizations that are otherwise impossible, making it a popular choice in the field of Digital Signal Processing (DSP), where performance, energy efficiency, parallelism and memory footprint are often very relevant considerations.

In the Related Works section, we go over a number of existing solutions in the dataflow space.

B. MLIR

MLIR [3] is a recent initiative from the LLVM team that establishes a framework for the development of modular intermediate representations. By allowing different concepts and levels of abstraction to be separated into different *dialects* while allowing them to coexist with each other as the same data structure, the project fosters reuse of code transformations between different compiler projects while preserving safety, performance and separation of concerns. MLIR has quickly achieved relevance in the compiler community, having been adopted by various projects such as TensorFlow [11], which focuses on machine learning; CIRCT [12], which provides hardware design tools; and Microsoft's Project Verona [13], which is a research compiler for concurrent ownership. MLIR is also used by traditional general-purpose compiler projects such as Polygeist [8], which converts C source code into MLIR and provides a number of polyhedral optimizations. One thing that makes MLIR very attractive as a compatibility medium is the fact that its built-in dialects are easily convertible into LLVM IR, a low-level IR that has already been adopted by many languages and that can be compiled into highly optimized code for many target architectures.

In analogy to the assembly-language-inspired *instructions* of LLVM IR, MLIR code consists of *operations*. However, differently from instructions, operations are not restricted to single physical actions in a processor; they can represent any kind of abstract information, and can contain other operations. This allows operations to represent complex concepts such as modules, functions, and structured control flow. Related operations are grouped into *dialects*, and developers are free to pick and choose which dialects are relevant to their project, and to define and share their own dialects and operations. MLIR currently provides built-in dialects for control flow, memory management, arithmetic and tensor operations, function and scope management, parallelization, and other concepts that are useful for general-purpose compilers. One of these dialects maps directly into LLVM IR, and is commonly used as a target by projects that wish to leverage the existing framework around it.

In Listing 2, an example of MLIR can be seen. Notice that every operation includes the name of its dialect; operations starting with `llvm` are part of the LLVM dialect, but others like `arith` and `math` are higher-level and can be used in compilers for different architectures.

These dialects are converted between each other by "passes", which consist of formalized transformation procedures that preserve certain guarantees about the state of the IR and allow for automatic parallelization of the compilation pipeline. The fact that these passes work on any level of abstraction is useful in the context of new compiler research: before this technology was introduced, the options for the development of a new compilation technique were limited to either the creation of a source-to-source compiler and commitment to the user-facing level of abstraction of an existing target language such as C, or to delve into the code of an existing general-purpose compiler, whose complexity may hinder the development and adoption of the contribution. MLIR introduces the possibility of focusing only on the appropriate levels of abstraction and relying on existing passes to deal with everything else, and the fact that all dialects expose a standardized interface makes it easy to ensure that the contribution can be easily adopted and expanded upon by others.

As MLIR operations may have an arbitrary number of inputs and outputs, they are ideal for representing directed graph structures; they also allow for arbitrarily deep nested or hierarchical composition. These characteristics make them especially well-suited for dataflow networks, as we can then use the built-in data structure algorithms.

III. Contributions

This section covers the structure of the proposed compiler, its purpose-built dataflow graph IR, and a co-optimization strategy that combines memory pooling and dead code elimination to achieve gains in performance and memory usage.

A. Multi-Source Compilation Flow

To explore the potential of co-optimization in dataflow compilation, both kernel code and dataflow graph data must be manipulated at the same time. By converting them into MLIR,

it becomes possible to use the built-in graph transformation tools to implement a SDF flattening, a DCE and memory-optimizing transformations. The built-in MLIR conversion passes are leveraged to generate LLVM IR that can be compiled into an executable with existing tools. The proposed MLIR dialect is capable of representing all of the abstractions necessary for an SDF graph (graphs and subgraphs, actortypes, nodes, edges, ports, parameters etc.).

Figure 1 illustrates the proposed compilation flow. Kernel code is transformed into high-level MLIR using the C front-end of the Polygeist project [8]. A custom parser for the DIF language [7] extracts dataflow graph information and transforms it into the proposed dataflow graph dialect. The structured topology is analysed, at which point performance-related features (unused ports, statically-known parameters, etc.) can be extracted. During co-optimization, this information is used to perform transformations in the kernel code and graph data. After co-optimization finishes, the graph is flattened, scheduled and bufferized, and the built-in lowering passes are used to transform the resulting code into LLVM IR which is then compiled into an executable using LLVM.

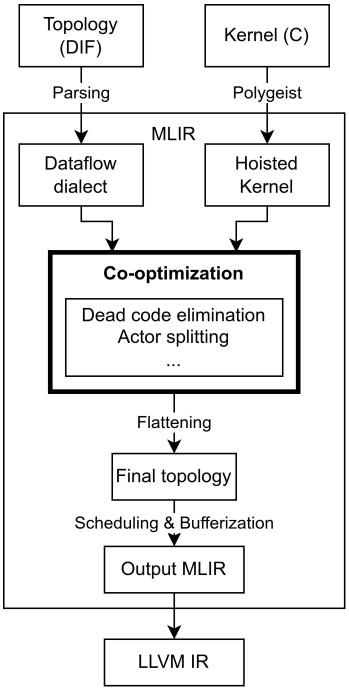

Fig. 1. Proposed compilation pipeline, with co-optimizations

B. Dataflow IR

The proposed dataflow IR consists of operations for four main abstractions: *Graph*, *Kernel*, *Actor*, and *Edge*.

- *GraphOp* represents a single hierarchical level of a graph. GraphOp encodes the name, MoC, graph-level parameters and, in the case of a sub-graph, its outside interface input and output ports. It contains KernelOps, NodeOps and EdgeOps within its single block, and may coexist with other GraphOps in the same MLIR module.
- *KernelOp* represents an external function that constitutes the implementation for one or more nodes. It encodes the function name of the implementation and its interface (parameters and input/output ports). It is also possible to define default values for the parameters.
- *NodeOp* encodes a single actor in the graph topology. It provides a name for the node and values for the actor parameters. It also contains a reference to an implementation, which can either be an KernelOp that represents a concrete function or a GraphOp that represents an interface to a sub-graph.
- *EdgeOp* encodes the edges of the graph. It consists simply of two node-port pairs that represent the input and output ports within the graph. Optionally, one of the node names may be replaced by the containing graph name, which represents an interface port for hierarchical graphs.

While MLIR includes support for representing arbitrary directed graph structures by using the built-in Value system, these structures rely on the ordering of incoming and outgoing edges and do not allow for keyword-based assignment, which harms the human legibility of the IR. We have instead opted to use explicit operations for edges, which allows for the nodes and ports to be referred to by name.

C. Dead code elimination by Co-optimization

Using Polygeist to convert ("raise" or "hoist") the C implementation of actors into higher-level MLIR enables access and modification to their internal operations through MLIR's robust API. This enables the development of fine-grained optimization strategies that make use of both coordination and kernel information that have not been explored in source-to-source dataflow projects, due to previously described challenges. In this work, we focus on Dead Code Elimination, but there are other possible optimizations, such as actor splitting. This consists of breaking an actor into components, exposing hidden parallelism (not supported by our current single-core implementation).

In an SDF graph, it is possible that some of the output ports of an actor are not connected to any edges, meaning that a significant portion of its operations may be generating unused data and is eligible to removal.

The removal of unused ports and operations brings improvements in both memory and performance. By removing ports, we remove the necessity to allocate "scratch" space in which to write the output, decreasing the total memory requirements and improving cache effectiveness. Additionally, any removed operation will no longer generate processor instructions, reducing the number of cycles required for the execution of that actor. This optimization is a kind of code specialization, potentially trading off additional program size for improved program performance. In this work we choose to perform systematic specialization, other trade offs could be explored in the future.

Most traditional compilers implement some form of DCE, but they are usually restricted to a single compilation unit, and only to cases where there is no memory aliasing. This cannot

be applied when using common SDF memory optimization strategies such as shared memory pools [14], which rely on memory recycling and thus create strong aliasing before DCE can be applied by the compiler.

However, with access to topology information, we can determine ahead of time if an output port won't be used, and MLIR allows us to operate directly on the Single Static Assignment (SSA) values of the kernel code.

```
void cartesian_to_polar(
    float *x, float *y,     // input ports
    float *r, float *theta) // output ports
{
  *r = sqrt(*x * *x + *y * *y);
  *theta = atan2f(*y, *x);
}
```

Listing 1. Example kernel implementation in C.

```
func @cartesian_to_polar(
  %arg0: llvm.ptr<f32>, %arg1: llvm.ptr<f32>,
  %arg2: llvm.ptr<f32>, %arg3: llvm.ptr<f32>)
{
  %1 = llvm.load %arg0 : !llvm.ptr<f32>
  %2 = arith.mulf %1, %1 : f32
  %3 = llvm.load %arg1 : !llvm.ptr<f32>
  %4 = arith.mulf %3, %3 : f32
  %5 = arith.addf %2, %4 : f32
  %6 = math.sqrt %5 : f32
  llvm.store %6, %arg2 : !llvm.ptr<f32>
  %9 = call @atan2(%3, %1)
    : (f32, f32) -> f32
  llvm.store %9, %arg3 : !llvm.ptr<f32>
  return
}
```

Listing 2. MLIR Polygeist output from source code in Listing 1. Operations that flow into specific output ports are highlighted.

Consider the C function in Listing 1 and its MLIR representation (Listing 2), where the operations that flow exclusively into **r** and theta are highlighted in **bold** and underlined, respectively. These represent the sets of operations that can be removed if the corresponding output port is unused. Finding this separation consists of following the value Directed Acyclic Graph (DAG) backwards from the port-writing `llvm.store` operations, and keeping a table of which output values depend on each operation.

As MLIR automatically keeps a graph data structure that tracks the references and definitions of all intermediary values, this can be implemented with a simple recursive algorithm that walks this graph. Unlike LLVM IR, MLIR provides a structured control flow dialect that simplifies the tracking of values entering and leaving control flow regions; reasoning about branches and jumps is not necessary. However, since we're relying on LLVM to generate optimized machine code, we can skip this recursive step; it is enough to just delete all operations that directly access the memory of the output we want to remove. We can leverage the built-in function-scope dead code elimination implemented in LLVM to guarantee that all unused intermediary values and their associated operations will be removed in the final binary.

There are a couple of caveats. This method does not apply for operations that access global variables or call functions with side effects, as they may be difficult to trace between different sections of the function. It also assumes that there is no aliasing between the pointers of each port, i.e., the memory assigned to one port won't be accessed by offsetting the pointer of another port. In the case of SDF, it is assumed that a correct kernel implementation conforms to this.

The compiler then creates a copy of the kernel, removes the relevant operations, and updates the C interface to exclude the removed output parameters. All nodes that reference this kernel with this specific subset of unused ports have their implementation fields updated. If there are no remaining nodes that reference the original function, it is deleted.

D. Dataflow Scheduling and Bufferization

After kernel optimizations are applied and and the hierarchical network is flattened into a single level, scheduling and bufferization take place. Scheduling consists of determining a valid order in which to fire each actor, and bufferization consists of assigning a location in memory to each token, which will be passed as input and output arguments to the kernel function of each actor. Many strategies require both processes to be done at the same time.

The allocation of memory in dataflow compilation is a well-researched problem. When minimizing memory usage, an useful abstraction is the Memory Exclusion Graph (MEG), which consists of Memory Objects that represent each token in an SDF period [14]. Tokens that are involved in the same actor activation (and which, therefore, must not share the same space in memory) are connected together in the MEG.

There are several strategies to bufferize a MEG, particularly when it comes to parallel schedules. It can be proven that the minimum memory footprint for an SDF network is equivalent to the MEG's Maximum Weight Clique (its largest fully connected sub-graph), which can be computed by exact algorithms or approximated with an heuristic. However, devising a schedule for an optimal MEG is a computationally complex problem. It has been experimentally determined [14], however, that there are simple heuristics that achieve similar results with little compromise in memory usage and performance.

In the current single-core approach of IaRa, we have selected a post-scheduling, first-fit allocation strategy and a self-timed greedy scheduling strategy, for their simplicity and high memory efficiency. Through a process called symbolic execution [15], the size and location of tokens in a shared memory pool is determined and the MEG and the schedule are simultaneously constructed.

This consists of following the scheduling strategy as if real data were being processed, but without executing the kernels. One by one, nodes are selected to be symbolically executed and a size and location for the memory of their ports is allocated in the memory pool, which grows as necessary if empty space of suitable size is not found. This ensures that any patch of memory is only allocated while its contents are

live, and that it can be freed and reused after the contents are consumed by an actor.

Before executing this algorithm, the total number of executions of each node per SDF iteration must be determined. This is achieved by computing the liveness and boundedness of the SDF network [15]. This number can then be used to determine a self-timed schedule (that is, a schedule where each node executes as soon as its inputs become available).

Once the algorithm finishes, the schedule contains an ordered list of firings and the memory offsets for its function call arguments, as well as the total size of the shared memory pool, to be statically allocated.

There is some flexibility when choosing heuristics for node selection and Memory Object allocation, which can affect the performance of the schedule and the final pool size. A schedule that favors firing the same node sequentially as many times as possible has better cache characteristics than code that fires available nodes in a round-robin fashion; likewise, it is preferable to fire a node whose input tokens have been recently produced, as it is likely that they will still be cached. When optimizing for memory, the compiler can instead prioritize nodes that will cause the smallest increase in the pool size.

Given the possibility space when choosing these strategies, we opted to implement a baseline strategy that does not differentiate between candidate nodes; advanced strategies are kept for a future work. Our pool memory allocation scheme uses a first-fit strategy that has been experimentally determined [14] to be close in effectiveness to the theoretical optimum.

IV. Related works

Since the beginning of dataflow programming in the late eighties [9], multiple development frameworks [16]–[22] have been proposed by both academia and industry to ease the development of signal processing applications. Most of them allow developers to compile their dataflow programs into binaries so they can be later executed. In all these frameworks, while the dataflow graphs are usually described using dataflow languages (graphical or not), the code of the actor can be either described with a traditional procedural language (e.g. C/C++) or a dataflow-specific language (e.g. CAL [23]).

Frameworks that use these dataflow languages [20], [21] are usually able to co-optimize kernel and coordination code using a custom intermediate representation. Unfortunately, most dataflow-specific languages did not reach adoption by the compilation community so many projects [17], [19], [22] rely on describing the actor code with classical procedural languages like C/C++. As such, those projects use traditional C/C++ compilers (GCC or Clang) to compile their code, which prevents them from performing optimizations on actors that require knowledge of the dataflow graph.

V. Experimental Results

The experiments aim to demonstrate that utilizing dataflow graph data to assist kernel code compilation can provide a clear performance advantage. For that purpose, an example algorithm is developed that is representative of some real-life DSP use-cases where some code, while being present in a function, is not necessary for the algorithm that uses the function. Our hypothesis is that a co-optimized DCE pass will reduce memory requirements and increase performance of the executable.

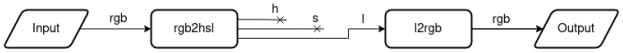

Fig. 2. SDF dataflow network for the video processing algorithm used in the experiment. One data token firing corresponds to an entire frame of uncompressed 480p video.

This algorithm takes as input a raw rgb24-encoded color video file and sets its hue and saturation levels to zero (transforming it into a grayscale video), before outputting it in the same rgb24 format, resulting in a grayscale version. Here, we use a general-purpose RGB to HSL actor, and discard the H and S outputs; the L to RGB actor reverses the conversion, setting the H and S channels to 0. Both actors operate on an entire 640x480 video frame at a time, as opposed to single pixel values.

To measure the impact of the DCE pass, we generate two versions of the executable. Both versions are compiled by raising the C implementation of the actors into MLIR, using the C front-end of Polygeist. Using this method, all of LLVM's built-in optimizations will be applied (such as automatic inlining and constant propagation). The optimized version differs from the baseline version only on the enabling of a DCE pass based on coordination code.

Experiments were conducted on a desktop computer running Manjaro Linux 21 with an Intel i5-4460 3.20GHz CPU, 16GB of RAM, and SSD storage. The project is built against development versions of LLVM (commit `89525cbf`) and Polygeist (commit `745d6841`). Timing information was obtained by using the `time` command on a Linux machine. The measurements were done 50 times for each version, and the statistical bootstrap method was used to compute the average and standard error.

TABLE I
Improvements of execution time and memory consumption

	Wall time [s]	**Pool size** [kB]
Baseline	5.29 ± 0.34	4608
DCE	4.07 ± 0.28	2150.4
Improvement	23% reduction	53% reduction

The DCE pass removes kernel operations related to the computation of the H and S values. This can be observed as a wall time reduction of 23%. There is also a 53% reduction in the amount of allocated static memory for the shared memory pool; this corresponds to the amount of memory required to receive the tokens of the H and S unused outputs.

The application in question was chosen to highlight the potential of this optimization strategy. The hue and saturation outputs of the RGB to HSL algorithm are relatively expensive to compute. This sort of readily-available function is common

in real-life applications, and we are confident that the chosen example faithfully represents a real use-case.

MLIR made it possible to implement IaRa within 1 person-year without any previous experience of LLVM/MLIR. A similar optimization would be feasible using a custom IR, but this would also require the reimplementation of a whole C compiler, which is far from trivial. Not relying on LLVM/MLIR infrastructure and existing dialects would require the designer to build a novel IR target, to reimplement all of the fine grain optimizations available natively in LLVM, such as DCE, and to emit correct machine code. Meanwhile, modifying an existing source to source compiler like Preesm [24] to allow for this functionality would require extensive modification of the internal graph representation and code generation, as it is not prepared to modify the internals of a C function in a flexible way.

VI. Conclusion

Dataflow models of computation have been originally developed to assist in the development of portable, high performance applications, and they have proven useful for signal processing systems. When it comes to the applicability of dataflow MoCs advances in compiler research, the strict separation that Dataflow enforces between coordination and kernel code has been problematic.

With our experiments, we have demonstrated that MLIR makes it possible to bridge this gap. By retrieving information from the graph topology description, we have combined two optimization strategies that would be incompatible otherwise: alias-creating memory pools and dead code elimination. To achieve this, we have used existing tools such as DIF and Polygeist, and implemented traditional dataflow algorithms in a cutting edge compilation framework in the form of the IaRa dialect, showing that there is potential for further integration between projects. With this implementation, we have achieved a 30% wall time improvement and a 53% memory usage improvement in a video processing application.

There is a wide set of future directions to explore from here, many of them enabled by the large variety of projects that are already found within the MLIR ecosystem. For instance, efforts can be made in integrating Polygeist polyhedral optimization passes as well as MLIR's built-in dialects for parallelism, such as OpenMP. Current machine learning dialects such as ONNX [25] and TensorFlow contain highly-optimized implementations of tensor operations, which could be integrated as actor kernels in the proposed framework.

References

[1] S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, *Handbook of Signal Processing*, 3rd ed. Springer, 2019.

[2] N. Vasilache, O. Zinenko, A. J. C. Bik, M. Ravishankar, T. Raoux, A. Belyaev, M. Springer, T. Gysi, D. Caballero, S. Herhut, S. Laurenzo, and A. Cohen, "Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction," *arXiv preprint arXiv:2202.03293*.

[3] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," *International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14, 2021.

[4] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communications of the ACM*, vol. 35, no. 2, p. 96, 1992.

[5] K. Desnos, M. Pelcat, S. S. Bhattacharyya, and S. Aridhi, "PiMM: Parameterized and Interfaced Dataflow Meta-Model for MPSoCs Runtime Reconfiguration," in *Embedded Computer Systems (SAMOS), 2013 International Conference on*, 2013.

[6] E. A. Lee and T. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[7] C.-J. Hsu, F. Keceli, M.-Y. Ko, S. Shahparnia, and S. S. Bhattacharyya, "Dif: An interchange format for dataflow-based design tools," in *International Workshop on Embedded Computer Systems*. Springer, 2004, pp. 423–432.

[8] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist: Affine c in mlir," 2021.

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

[10] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cvclo-Static Dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397 – 408, 1996.

[11] J. Pienaar, "Mlir in tensorflow ecosystem," 2020, compilers For Machine Learning (C4ML) 2020, San Diego, CA, USA.

[12] S. Eldridge, P. Barua, A. Chapyzhenka, A. Izraelevitz, J. Koenig, C. Lattner, A. Lenharth, G. Leontiev, F. Schuiki, R. Sunder *et al.*, "Mlir as hardware compiler infrastructure," in *Workshop on Open-Source EDA Technology (WOSET)*, 2021.

[13] Microsoft, "Project verona," https://github.com/microsoft/verona, 2019.

[14] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi, "Memory Analysis and Optimized Allocation of Dataflow Applications on Shared-Memory MPSoCs," *Journal of Signal Processing Systems*, vol. 80, no. 1, pp. 19–37, Jul. 2015.

[15] A. Ghamarian, M. Geilen, T. Basten, B. Theelen, M. Mousavi, and S. Stuijk, "Liveness and boundedness of synchronous data flow graphs," in *2006 Formal Methods in Computer Aided Design*, 2006, pp. 68–75.

[16] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming Heterogeneity - The Ptolemy Approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[17] M. Pelcat, J. Piat, M. Wipliez, S. Aridhi, and J.-F. J.-F. Nezan, "An Open Framework for Rapid Prototyping of Signal Processing Applications," *EURASIP Journal on Embedded Systems*, vol. 2009, no. 1, 2009.

[18] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. Von Platen, M. Mattavelli, and M. Raulet, "OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems," *SIGARCH Computer Architure News*, vol. 36, no. 5, pp. 29–35, 2009.

[19] J. Castrillon, R. Leupers, and G. Ascheid, "MAPS: Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs," *IEEE Transactions on Industrial Informatics*, vol. X, no. X, pp. 1–19, 2011.

[20] H. Yviquel, A. Lorence, K. Jerbi, A. Sanchez, G. Cocherel, and M. Raulet, "Orcc: Multimedia development made easy," in *Proceedings of the 21st ACM international conference on Multimedia*, 2013, pp. 863–866.

[21] J. Kodosky, "LabVIEW," in *Proceedings of the ACM on Programming Languages*, vol. 4, no. June. ACM, 2020, pp. 1–54.

[22] J. Boutellier, J. Wu, H. Huttunen, and S. S. Bhattacharyya, "PRUNE: Dynamic and decidable dataflow for signal processing on heterogeneous platforms," *IEEE Transactions on Signal Processing*, vol. 66, no. 3, pp. 654–665, 2018.

[23] J. Eker and J. W. Janneck, "CAL language report: Specification of the CAL actor language," University of California, Berkeley, Berkeley, Tech. Rep., 2003.

[24] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi, "Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming," in *Education and Research Conference (EDERC), European Embedded Design in*, Sept 2014, pp. 36–40.

[25] T. Jin, G.-T. Bercea, T. D. Le, T. Chen, G. Su, H. Imai, Y. Negishi, A. Leu, K. O'Brien, K. Kawachiya *et al.*, "Compiling onnx neural network models using mlir," *arXiv preprint arXiv:2008.08272*, 2020.