Graphical Program Transformations for Embedded Systems

Robert Stewart
Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, UK
R.Stewart@hw.ac.uk

Bernard Berthomieu
LAAS-CNRS
Université de Toulouse
Toulouse, France

Paulo Garcia
Systems and Computer Engineering
Faculty of Engineering and Design
Carleton University
Ottawa, Canada

Idris Ibrahim
Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, UK

Greg Michaelson
Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, UK

Andrew Wallace
Engineering and Physical Sciences
Heriot-Watt University
Edinburgh, UK

ABSTRACT

Dataflow languages are widely used for real-time embedded systems. This paper presents a dataflow program transformation tool for Orcc, a high level embedded systems programming environment. The aim of the tool is to increase program throughput performance of FPGAs and other embedded systems.

ACM Reference Format:
Robert Stewart, Bernard Berthomieu, Paulo Garcia, Idris Ibrahim, Greg Michaelson, and Andrew Wallace. 2019. Graphical Program Transformations for Embedded Systems. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3297280.3297555

1 INTRODUCTION

Dataflow Programming. Dataflow languages and visual programming environments are widely used for designing real-time embedded systems. The dataflow programming model is especially suitable for FPGAs, as mapping and routing distributes dataflow pipelines across the programmable fabric. In the dataflow model, the execution of actors depends only on data availability, allowing each actor in a program to execute asynchronously without a global control flow sequentialising their execution. Large data structures in actors such as intermediate arrays may use on-chip block RAM (BRAMs), which are distributed across FPGA fabric, meaning there is no memory contention when multiple actors update their kernel variables. The number of actors in a program derives the number of independent processing elements, subject to hardware resource availability, *i.e.* both memory access and computation is inherently parallel.

Parallel Refactoring. Parallel refactoring tools aim to speed up code execution by parallelising sequential code across multiple processing elements *e.g.* the cores of a CPU. Some tools exploit language properties to parallelise code, *e.g.* the referential transparency of pure code and equational laws to rewrite a *map* as a parallel *parMap*. Software languages for which parallel transformation tools exist include Haskell [8], Erlang [6] and C++ [7].

For customisable hardware like FPGAs, dataflow programming environments represent a high level abstraction above Hardware Description Languages (HDLs) *i.e.* Verilog or VHDL. This abstraction enables program analysis *e.g.* optimal static scheduling [10] and program transformation (this paper). Some dataflow programming environments support static dataflow models only, *e.g.* direct feedthrough function blocks in Simulink [12]. The open source Orcc development environment [16] supports CAL, a dataflow language for programming real-time embedded systems, which was developed as part of the Ptolemy II project [15]. Programmers can implement CAL actors that have dynamic data rates and internal state.

Parallel code execution can improve performance. However, parallel transformations must preserve a program's functional semantics, *i.e.* a transformed program must have identical functional behaviour to the original program. Related work on verified program transformation includes verification of functional code refactoring using proofs assistants [14]. Value-dependent code scheduling and dynamic data rates complicates auto-parallelisation of dynamic Dataflow Process Network (DPN) [11] actors. Our approach (Fig. 1) overcomes this limitation by using model checking to identify parallelisable multi-rate static (MRDF) [9] and cyclo-static (CSDF) [5] actors in DPN programs. The approach is general to any dynamic dataflow language, this paper demonstrates the approach with CAL in the Orcc development environment.

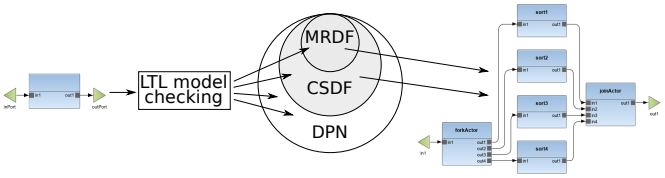


Figure 1: Verifying and Transforming Cyclo-Static Dataflow

2 DATAFLOW

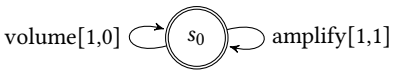
Dataflow programs are directed graphs of connected actors. Each actor encapsulates an algorithmic kernel. Lossless, order-preserving FIFOs through which tokens flow connect actors. Multiple actors

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SAC '19, April 8–12, 2019, Limassol, Cyprus
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5933-7/19/04.
https://doi.org/10.1145/3297280.3297555

can execute in parallel if the underlying hardware architecture has multiple processing elements, each assigned at least one actor.

```
❶ actor Amplifier () In, Volume ==> Out :
❷    vol := 1;
❸    volume: action Volume:[newV] ==>
❹      do vol := newV; end
❺    amplify: action In:[d] ==> Out:[d*vol] end
❻    schedule fsm s0 :
         s0 (volume)  --> s0;
         s0 (amplify) --> s0;
     end
❼    priority volume > amplify; end
end
```

(a) Actor Implementation



(b) Actor FSM Execution Model

Figure 2: An Amplifier Dataflow Actor

Actor Programming. Fig. 2a shows a volume amplifier example. ❶ labels actor ports. The actor may have a store of private variables ❷. Actors contain procedures, called actions. Actions match on input patterns to consume tokens and output patterns produce tokens (❸ and ❺). Actions may also update store variables ❹. A Finite State Machine (FSM) determines enabled actions from each FSM state ❻. When multiple actions are fireable from a given state, a priority block can disambiguate multiple enabled actions ❼.

Programmed actors are then instantiated in Orcc's graphical development environment to construct dataflow graphs. The parallel transformation in Section 3 treats instantiated actors as the unit of computation to be parallelised.

Actor Execution Model. An FSM transition system drives actor execution, *e.g.* the FSM in Fig. 2b. Actions have data rates $[C/P]$, meaning they consume C tokens and produce P tokens when fired. The transition sequence in dynamic DPN actors may depend on the *value* of these tokens. Satisfying both of the following conditions enables an action:

(1) its value-dependent guard evaluates to true (if a guard exists),
(2) there is enough tokens to match its input patterns (if an input pattern exists).

3 DATAFLOW TRANSFORMATIONS

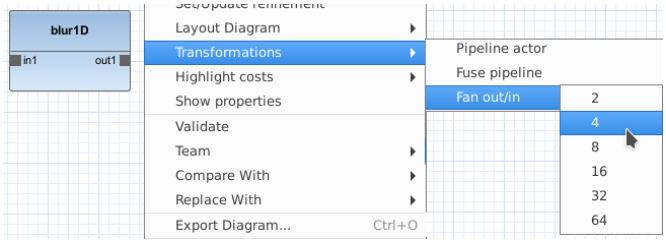
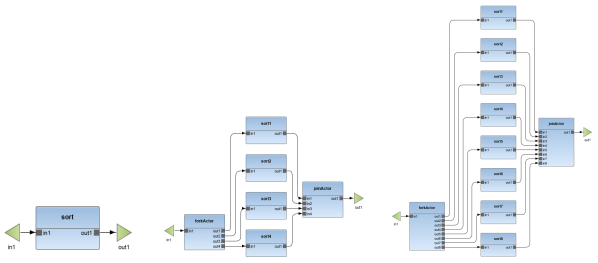


Figure 3: Interactive Dataflow Transformation

Graphical Transformations Environment. We have extended Orcc with a refactoring tool for parallelising static and cyclo-static actors



(a) original **(b) 4 actors** **(c) 8 actors**

Figure 4: Parallel Program after Transformation

which introduces *fork* and *join* actors to scatter/gather data to/from parallel replicas of a single (sequential) actor (Fig. 3). The user chooses how much parallelism they want (Fig. 4).

Model Checking Dataflow Actors. Our verification approach is:

(1) Translate an actor to a Fiacre specification [1]. Fiacre is a formal intermediate model to represent both the behavioural and timing aspects of embedded and distributed systems for formal verification.
(2) Fiacre specifications are checked with the TINA model checker [2] against Linear Temporal Logic (LTL) properties that we have formulated to capture cyclo-static actor scheduling.
(3) Our graphical transformations tool parallelises cyclo-static actors where the model checker is unable to find counterexamples to the LTL properties.

Parallelisation Algorithm. The transformation algorithm of an actor A in the transformation tool is:

(1) Extract the number of firings required for an FSM cycle, and the consumption and production data rates in this sequence.
(2) Create N parallel instances A_1 to A_N.
(3) Create a *fork* actor that distributes data across $A_1, .., A_N$.
(4) Create a *join* actor to consume chunks from actors $A_1, .., A_N$.
(5) Output the joined chunks as a sequential stream.

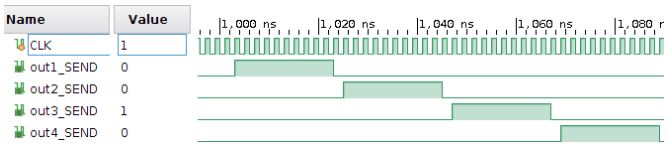
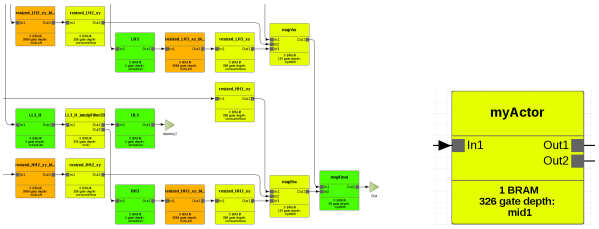


Figure 5: FPGA Waveforms of *fork* Actor Data Scattering

The transformed graph remains partially sequential due to the nature of the linear stream that the *fork* actor distributes. Fig. 5 shows the sequential nature of stream propagation as it broadcasts tokens to each parallel instance in sequence, in this case transmitting 10 tokens, 1 token per cycle, to each identical actor. Stream gathering by the *join* actor is sequentially also. These sequential phases limits the achievable speedup using the transformation tool.

Hardware Profiling. The Xronos [3] Orcc backend generates Verilog for FPGAs. It uses Xilinx's open source OpenForge [4] compiler to generate hardware cost reports for each actor at compile time. These are: 1) the number of BRAM blocks, and 2) the datapath

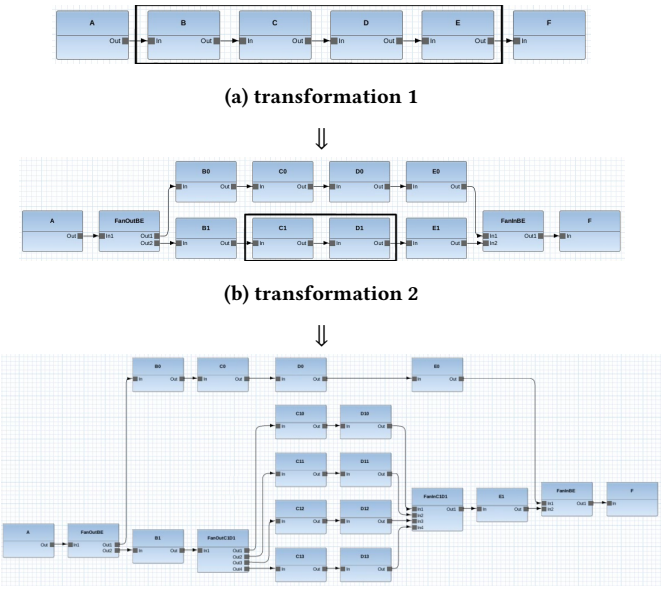
depth for each action in an actor. The datapath depth determines the minimum latency (microseconds) between each clock cycle. The action with the longest datapath determines the clock frequency of the generated FPGA design. To identify hardware performance bottlenecks, our tool lifts these hardware costs into the visual dataflow editor (Fig. 6). Green actors have small datapath depths relative to the orange critical actor(s). BRAM counts for implementing actor internal variables is also shown to the programmer.



(a) Graph hardware critical path (b) one actor

Figure 6: Visualising FPGA Resource Costs

4 CONCLUSION



(a) transformation 1

(b) transformation 2

(c) final version

Figure 7: Successive Transformations

This paper presents a program transformation tool that verifies and parallelises a single cyclo-static actor into multiple parallel actors. LTL formulas capture cyclo-static actor data rates. The workflow abstracts dataflow actors to Fiacre specifications, and the TINA model checker searches for counterexamples of the LTL formulas in the Fiacre abstraction to verify the correctness of parallel transformation.

Our approach provides a step towards auto-parallelisation of dynamic dataflow programs with a data-parallel transformation. It also serves as an optimisation layer for high level FPGA languages that compile to dataflow graph intermediate representations [13]. Verifying other parallel transformations, *e.g.* task and pipelined parallelism, and successive transformations of subgraphs of actors, would extend our approach. We have prototyped the latter in Orcc (Fig. 7). The broader aim of this work is to integrate automated formal verification into everyday embedded systems development.

ACKNOWLEDGMENTS

We acknowledge the support of the Engineering and Physical Research Council grant references EP/K009931/1 (Programmable embedded platforms for remote and compute intensive image processing applications), EP/N014758/1 (The Integration and Interaction of Multiple Mathematical Reasoning Processes) and EP/N028201/1 (Border Patrol: Improving Smart Device Security through Type-Aware Systems Design), and the Scottish Funding Council for a SICSA Postdoctoral and Early Career Researcher Exchanges grant.

REFERENCES

[1] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Frédéric Lang, and François Vernadat. 2008. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *ERTS 2008*. Toulouse, France.

[2] B. Berthomieu, P.O. Ribet, and F. Vernadat. 2004. The tool TINA - Construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research* 42, 14 (July 2004), 2741–2756.

[3] Endri Bezati. 2015. *High-Level Synthesis of Dataflow Programs for Heterogeneous Platforms: Design Flow Tools and Design Space Exploration*. Ph.D. Dissertation. School of Engineering, Ecole Polytechnique Federale de Lausanne, Switzerland.

[4] Endri Bezati, Hervé Yviquel, Mickaël Raulet, and Marco Mattavelli. 2011. A Unified Hardware/Software Co-Synthesis Solution for Signal Processing Systems. In *DASIP 2011, Tampere, Finland, November 2-4, 2011*. IEEE, Tampere, Finland, 186–191.

[5] Greet Bilsen, Marc Engels, Rudy Lauwereins, and J. A. Peperstraete. 1996. Cyclo-static dataflow. *IEEE Trans. Signal Processing* 44, 2 (1996), 397–408.

[6] Christopher Brown, Marco Danelutto, Kevin Hammond, Peter Kilpatrick, and Archibald Elliott. 2014. Cost-Directed Refactoring for Parallel Erlang Programs. *International Journal of Parallel Programming* 42, 4 (2014), 564–582.

[7] Christopher Brown, Kevin Hammond, Marco Danelutto, Peter Kilpatrick, Holger Schöner, and Tino Breddin. 2011. Paraphrasing: Generating Parallel Programs Using Refactoring. In *FMCO 2011, October 3-5, 2011, Revised Selected Papers*. Springer, Turin, Italy, 237–256.

[8] Christopher Brown, Hans-Wolfgang Loidl, and Kevin Hammond. 2011. ParaForming: Forming Parallel Haskell Programs Using Novel Refactoring Techniques. In *TFP 2011, May 16-18, 2011, Revised Selected Papers*. Springer, Madrid, Spain, 82–97.

[9] Rudy Lauwereins, Marc Engels, Marleen Adé, and J. A. Peperstraete. 1995. Grape-II: A System-Level Prototyping Environment for DSP Applications. *IEEE Computer* 28, 2 (1995), 35–43.

[10] Edward A. Lee and David G. Messerschmitt. 1987. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. Computers* 36, 1 (1987), 24–35.

[11] E. A. Lee and T. M. Parks. 1995. Dataflow process networks. *Proc. IEEE* 83, 5 (May 1995), 773–801.

[12] MATLAB. 2018. Simulink. https://uk.mathworks.com/products/simulink.html.

[13] Robert Stewart, Kirsty Duncan, Greg Michaelson, Paulo Garcia, Deepayan Bhowmik, and Andrew Wallace. 2018. RIPL: A Parallel Image Processing Language for FPGAs. *TRETS* 11, 1 (2018), 7:1–7:24.

[14] Nik Sultana and Simon J. Thompson. 2008. Mechanical verification of refactorings. In *Proceedings of the 2008 ACM SIGPLAN, PEPM 2008, January 7-8, 2008*. ACM, San Francisco, California, USA, 51–60.

[15] University of California at Berkeley. 2018. The Ptolemy Project. https://ptolemy.eecs.berkeley.edu/.

[16] Hervé Yviquel, Antoine Lorence, Khaled Jerbi, Gildas Cocherel, Alexandre Sanchez, and Mickaël Raulet. 2013. Orcc: multimedia development made easy. In *ACM Multimedia Conference, MM '13, October 21-25, 2013*. ACM, Barcelona, Spain, 863–866.