Shared-variable synchronization approaches for dynamic dataflow programs

Apostolos Modas[1], Simone Casale-Brunet[2], Robert Stewart[3], Endri Bezati[2], Junaid Ahmad[4*], Marco Mattavelli[1]
[1]EPFL SCI STI MM, École Polytechnique Fédérale de Lausanne, Switzerland
[2]SIB Swiss Institute of Bioinformatics, Lausanne, Switzerland
[3]Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, United Kingdom
[4]J Nomics, Manchester, United Kingdom

***Abstract*—This paper presents shared-variable synchronization approaches for dataflow programming. The mechanisms do not require any substantial model of computation (MoC) modification, and is portable across both for hardware (HW) and software (SW) low-level code synthesis. With the shared-variable formalization, the benefits of the dataflow MoC are maintained, however the space and energy efficiency of an application can be significantly improved. The approach targets Dynamic Process Network (DPN) dataflow applications, thus making them also suitable for less expressive models *e.g.* synchronous and cyclo-static dataflow that DPN subsumes. The approach is validated through the analysis and optimization of a High-Efficiency Video Coding (HEVC) decoder implemented in the RVC-CAL dataflow language targeting a multi-core platform. Experimental results show how, starting from an initial design that does not use the shared-variable formalism, frames per second throughput performance is increased by a factor of 21.**

I. Introduction

In recent years, there has been a renewed interest in the field of dataflow programming. This has been driven by the limitation of the frequency increases of deep sub-micron CMOS silicon technology, which has shifted the evolution of processing platforms to systems comprising heterogeneous arrays of parallel processors. The emergence of these manycore architectures poses new problems and challenges for compiling applications efficiently to them. A major challenge is the portability of applications across heterogeneous architectures, in particular, the portability of the *parallelism* present in each specific application [1]. Dataflow programming, in all its different model of computations (MoCs), is well placed to overcome the challenges of exploiting efficient heterogeneous architectures. Dataflow MoCs are widely used for the specification of data-driven algorithms in many application areas, e.g. video and audio processing, bioinformatics, financial trading, packet switching applications. In these domains, scalability and composability of systems are increasingly important requirements, and dataflow MoCs are an efficient way of implementing algorithms in standardized high-level languages [2], [3], [4].

Dataflow MoCs are architecture agnostic, making them highly valuable for the specification of performance portable applications that can be deployed on a wide variety of computing platforms. Dataflow programs are mapped to specific architectural components with low-level, target specific code synthesis.

By relying on isolation and non-sharing, an actor can access its own state without fear of data-races. The approach can introduce inefficiencies in generated code, *e.g.* allocations causing high memory use, and high levels of actor idle time whilst data is explicitly copied. As an example, in the context of video compression, the output data generally depends on intermediate data structures that different actors are obliged to replicate since they cannot be shared. This can severely impact both the time and space performance and the energy efficiency of an application.

In this paper, we present a set of shared-variable synchronization approaches that do not require any substantial MoC modification and that are portable both for HW and SW low-level code synthesis. The main advantage of this formalization is the fact that the benefits of the dataflow MoC are maintained, however, the space and time performance and energy efficiency of an application can be significantly improved. The method targets Dynamic Process Network (DPN) dataflow applications, thus making them also suitable for less expressive models *e.g.* synchronous and cyclo-static dataflow that DPN subsumes.

The paper is structured as follows: Section II provides an overview of current dataflow MoCs and shared-variable approaches. Section III presents a novel and generic shared variable paradigm that can be introduced in a dataflow MoC without fear of data-races. Section IV describes how this methodology has been implemented in the standardized RVC-CAL dataflow language. Experimental results are provided in Section V, where an HEVC video decoder implemented using the RVC-CAL dataflow language has been optimized with the use of shared variables. Finally, Section VI concludes the paper and discusses future work directions.

II. Background work

A. Dataflow model of computations

Dataflow programming models have a long and rich history dating back to the early 1970s [5], [6]. As depicted in Fig. 1a, a *dataflow program* is defined as a (hierarchical) directed graph in which nodes (called *actors*) represent the computational kernels and directed edges (called *buffers*) represent the lossless, order preserving, and point-to-point communication channels

*This work has been done while author was with EPFL SCI STI MM

between actors. Buffers are used to communicate sequences of atomic data packets (called *tokens*). In literature, several variants of dataflow *Models of Computation* (MoC) have been introduced [6], [7], [8]. One of their common properties is that individual actors encapsulate their own state which is not shared among other actors of the same program. Instead, actors communicate with each other exclusively by sending and receiving tokens by means of buffers connecting them. The absence of race conditions makes the behavior of dataflow programs more robust to different execution policies, whether those be truly parallel or some interleaving of the individual actors.

(a) An example of a dataflow network with three actors (i.e. *Producer*, *Filter* and *Consumer*) and two buffers (i.e. b_1 and b_2).

```
actor Producer() ==> int Y :
 int cnt := 0;
 produce: action ==> Y:[cnt]
 guard cnt < 3
 do
  cnt := cnt + 1;
 end
end
```

(b) Producer.cal: at each firing it produces a token on output port Y.

```
actor Filter() int X ==> int Y :
 copy:   action X:[a] ==> Y:[ a] end
 invert: action X:[a] ==> Y:[-a] end

 schedule fsm state1 :
   state1(copy)   --> state2;
   state2(invert) --> state1;
 end
end
```

(c) Filter.cal: at each firing it consumes a token from input port X and it produces a token on output port Y.

```
actor Consumer() int X ==> :
 consume: action X:[a] ==> end
end
```

(d) Consumer.cal: at each firing it consumes a token from input port X.

Fig. 1. RVC-CAL program example: dataflow network configuration and actors source code.

DPN is an expressive MoC in comparison with other dataflow models, *e.g.* by supporting dynamic branching with actor firings predicated on token *values*. This makes the DPN model sufficient for expressing dynamic, complex algorithms, but comes at the cost of analysis and optimization opportunities. To execute, DPN actors perform a sequence of discrete computational *steps* (or *firings*). During each step, an actor can consume a finite number of input tokens, produce a finite number of output tokens, and modify its own internal state variables if it has any. The behavior of a DPN actor is specified as a set of *firing rules* and *firing functions*. Formally an actor is defined as a pair of (f, R) such that $f : \mathbb{S}^m \mapsto \mathbb{S}^n$ where f is a firing function that consumes a token sequence on m input ports and produces a token sequence on n output ports, $\mathbb{S}$ is the set of all possible sequences and $R = [R_1, ..., R_n]$ is the set of firing rules. These rules determine when the actor may fire or not by describing the input sequences and actor states that need to be present for the actor to execute a step (i.e., for it to be *enabled*). For a given input sequence, the firing functions determine a sequence/state combination for which the actor is enabled according to the firing rule, the output tokens produced in such step, and, if applicable, the next actor state. It must be observed that at each step only one action can be selected and fired. In general, DPN actors may be non-deterministic, which means that the firing function may yield more than one combination of outputs and next states. Furthermore, the execution can be totally dynamic, meaning that the number of consumed/produced tokens may vary according to the input sequence, which severely limits any compile time analysis of DPN actors.

B. Dataflow programming languages

In the last decades, a plethora of different programming languages has been used to model and implement dataflow programs [9]. Imperative languages (e.g. C/C++, Java, Python) have been extended with parallel constructs, or pure dataflow languages (e.g. Ptolemy, Esterel) have been formalized. The RVC-CAL language [10] is the sole standardized dataflow programming language which fully captures the behavioral features of DPN. As an example, the RVC-CAL program reported in Fig. 1a is composed of three actors (i.e., `Producer`, `Filter` and `Consumer`) and two buffers (i.e., b_1 and b_2). The `Producer` actor (see Fig. 1b) has an output port `Y` connected to b_1, the actor internal variable `cnt` and the action (i.e., firing function) labeled as `produce`. During each firing of the action, the value of `cnt` is modified and a token is produced on b_1. Furthermore, the execution of the action is guarded by the guard condition on `cnt`. The `Filter` actor (see Fig. 1c) has an input port `X` connected to b_1, an output port `Y` connected to b_2, the two actions `copy` and `invert`, and an actor state machine (FSM) which drives the action selection. The selected action can be fired only if a token is available on b_1 and there is at least one token place on b_2. During each firing, a token is consumed from b_1 and a token is produced on b_2. The `Consumer` actor (see Fig. 1d) is composed by the input port `X` connected to b_2 and the action `consume`. The action can be fired only if a token is available on b_2. During each firing, a token is consumed from b_2.

C. Shared-variable approaches

Conventional dataflow encapsulates isolated state inside actors. Their only means of communication is by sending and receiving data through dataflow edges. For example, the only way for the `Filter` and `Consumer` actors to know the value of the `cnt` variable in `Producer` is via explicit token passing. Sharing large data structures across multiple actors with token passing can be inefficient for memory use since the shared-nothing dataflow model enforces copying. Combining the dataflow model with transactional memory approaches [11], [12] is a potential solution to this problem, because there is only one copy of the large data structure, with read/modify synchronization controlled by transaction memory protocols.

a) Combining actors and STM: Actor-based programming and Software Transactional Memory (STM) has been recently been combined in [13], although their focus is concurrent agent-oriented cooperative actors in multi-threaded environments, in contrast to our treatment of actors as functional nodes in dataflow graphs. STM and dataflow actor programming is also combined in [14]. Their approach attaches two synchronization points to actors. Actors block until:

1) data at inputs become available, which is the conventional dataflow synchronization mechanism.
2) access to the transactional variables inside its atomic code block are released by another actor.

This contrasts with our approach, which synchronizes access to shared variables using the conventional dynamic dataflow synchronization mechanism of input data availability (the first of those blocking mechanisms), pattern matching on the explicit shared memory protocol signals for actor firing.

b) Efficient Dataflow with Sharing: As the number of parallel tasks increases on parallel hardware, access contention and latency of dataflow communication also increases. At High Performance Computing scale, existing approaches include cache coherent memory subsystems to reduce copying costs of MPI collective operations for data movement between nodes [15]. At embedded FPGA scale, copying quickly becomes prohibitive due to lack of memory, so reusing memories for multiple dataflow buffers is desirable. The approach in [16] statically analyses dataflow programs and computes the minimum buffer requirements for edges between actors. Rather than allocating memory for every edge buffer, the compiler instead reuses memory for multiple edge buffers. The approach is limited to SDF actors, whereas our approach supports dynamic dataflow. Moreover, our approach focuses on shared variables for internal actor data, rather than external shared buffers for communication.

c) Read/Write locks: Readers-writer locks provides synchronization primitives for concurrent access to variables shared between threads. There are many implementations, including `pthread_rwlock_t` in the POSIX standard, and in languages inlcuding C#, Go and Rust. As with our approach they support access policies that prioritise reads or writes (Section III-C). Our actor prioritisation policy also offers a *functionality specific* priority, whereby communication of high priority computation is encapsulated into prioritised actors (Section III-B).

III. Design Architectures

Our shared-variables synchronization architecture is based on the use of a supervisor actor, called the Shared-Memory Controller (SMC), which controls the access right of a single shared (global) variable of a dataflow program network. Fig. 2 depicts the case of a dataflow program composed by a single shared variable, M writers (i.e., actors that have write-only privilege on that variable) and N readers (i.e., actors that have write- read-only on that variable). In general, for each shared variable available on the network, an SMC actor is used to control a mutual exclusion access on the variable according to the design specific requirements. The functionality of the SMC

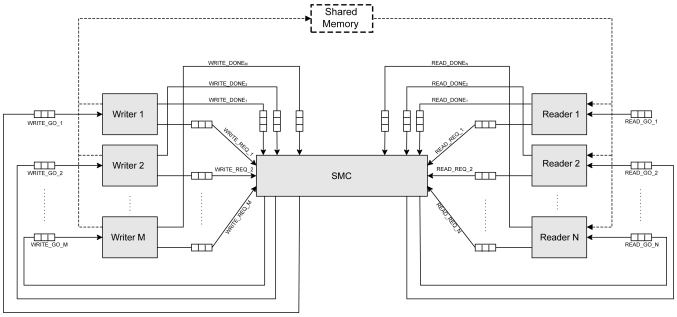


Fig. 2. Dataflow network with a single Shared-Memory Controller (SMC).

is like the one exposed by a monitor: it does not synchronize all actors at a specific point before allowing them to continue (i.e., like barrier-like synchronization method) but allows them to have both mutual exclusion and the ability to wait (lock) for a certain condition to be true.

There are three variants of the SMC controller:

1) A generic request-go-done transaction protocol. When simultaneous requests occur, the choice of which actor to grant shared memory access to is non-deterministic (Section III-A).
2) An extension of the generic protocol, prioritising shared memory access requests given a predefined actor priority sequence (Section III-B).
3) Another extension of the generic protocol, this time prioritising read requests before write requests or vice versa (Section III-C).

A. Generic Shared Memory Controller Protocol

Each reader/writer actor sends an access request using a token through the respective request buffer to the SMC (i.e., `READ_REQ_n` for a reader and `WRITE_REQ_m` for a writer). In other words, the actor fires a *lock* action transitioning from FSM state $s0$ to $s1$, and successively switch to a waiting state till an unlock token is received from the SMC through the unlock buffer (i.e., `READ_GO_n` for a reader and `WRITE_GO_m` for a writer). The actor fires the memory operation (either read or write) by firing the action, transitioning from FSM states $s1$ to $s2$. The memory operation happens during the execution of this action. The $s1 \rightarrow s2$ transition consumes the `GO` token from the SMC prior to the memory IO and produces a `DONE` token (either `READ_DONE_n` or `WRITE_DONE_n`) when exiting the action. An example is shown in Fig. 3, showing the message sequence between an SMC, two readers R_1 and R_2 and a writer W_1. It shows that multiple readers may read the shared variable, whilst memory write operations locks the memory location to perform it. On a successful memory IO transation, the FSM in readers and writers are reset from $s2$ to $s0$ for the next shared memory transaction.

B. SMC with Actor Priority

The first pioritiy-aware SMC controller prioritises requests given a predefined priority order of actors. When handling

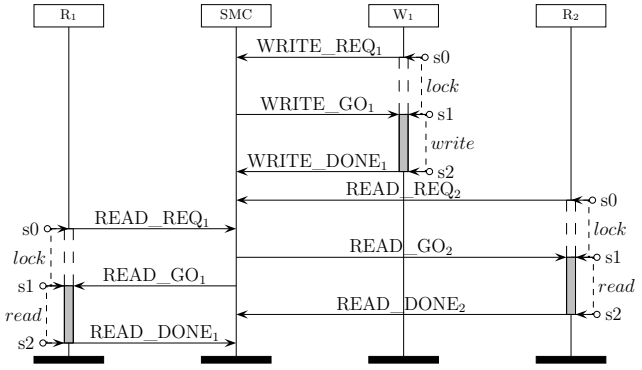


Fig. 3. Example message sequence with readers R_1 and R_2 and writer W_1 using the generic SMC (Section III-A).

new transaction requests, the SMC's scheduler traverses this priority sequence, serving the transaction request it finds by sending an unblock message. When the SMC is notified that an actor has completed the atomic operation, it schedules the next one (if any) or waits for new requests at the input ports. The state machine of the SMC controller for prioritising actor requests is shown in Fig. 4, which has three states and ten transitions between them.

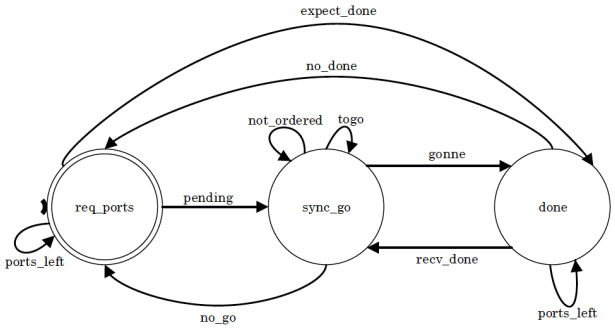


Fig. 4. Actor priority.

C. SMC with Read-Write or Write-Read priority

As an alternative to port (*i.e.* actor) priorities, the other priority-aware SMC design sequences requests according to whether they are read or write requests. That is, the SMC prioritises read requests ahead of any write requests, or vice versa. The state machine of the SMC controller for prioritising actor requests is shown in Fig. 5, which is larger than the FSM for actor priority (Fig. 4), with five states and fifteen transitions between them.

IV. RVC-CAL LANGUAGE EXTENSION

A. Shared Dataflow Variables

In the following section we illustrate how the standardized RVC-CAL language has been extended to support the shared variable functionality. The proposed shared variables solution

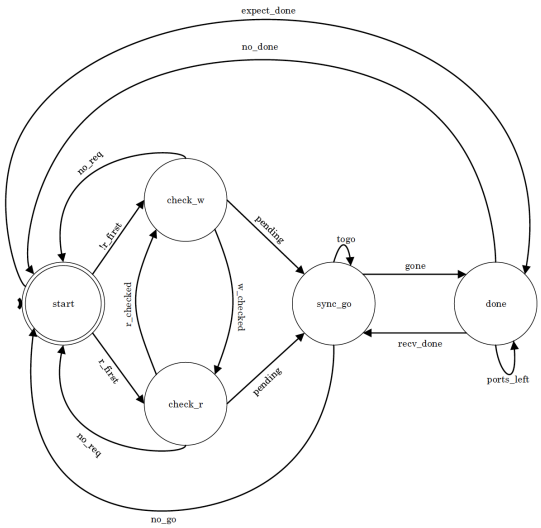


Fig. 5. Read/Write priority.

is supported by the use of a *@shared* tag as an extension to the RVC-CAL language. An example of its use is shown in Fig. 6. Variables to be implemented as shared variables among multiple actors are tagged with the same ID in each actor, *e.g.* the *sv1* ID in Figs 6a and 6b.

```
actor SvWriter() int I ==> int Y :

 @shared(id="sv1")
 int cnt := 0;

 produce: action I:[i] ==> Y:[cnt]
 guard cnt < 3
 do
  cnt := cnt + 1;
 end
end
```

(a) SvWriter.cal

```
actor SvReader() int I ==> int Y :

@shared(id="sv1")
int cnt := 0;

produce: action I:[x] ==> Y:[-cnt] end
```

(b) SvReader.cal

Fig. 6. RVC-CAL actors sharing a single shared variable. variable

B. Using the SMC Protocols

To interact with shared variables, an actor must:

1) Have an `@shared` tag attached to the internal variable.
2) Be connected to the SMC actor responsible for protecting transactions on memory for the shared variable.
3) Honour the SMC transaction protocol (Section III) when communicating with the SMC corresponding to that shared variable.

C. Software Implementation

The support for `@shared` tags is part of the XRONOS [17] CAL to C++ low-level code generation backend implemented by the authors. This is a code generation backend of the widely used Open RVC-CAL Compiler (Orcc) [18]. Through

XRONOS, each actor is transformed into a C++ class that contains the functions corresponding to action computation and its internal actor FSM scheduling. The dataflow graph is transformed into a C++ executable file with a *main* function that instantiates actors and creates FIFOs, implemented as lock-free circular buffers, and executes actor schedulers until the program terminates. All shared variables are declared in this file and are defined as *extern* static C++ variables in actor header files, thus all actors point to the same memory region when reading/writing to a shared RVC-CAL variable.

V. Experimental Results

For the scope of this work, an High-Efficiency Video Coding (HEVC) decoder, also known as MPEG-H Part 2, implemented with the RVC-CAL dataflow language has been used to analyze the acceleration that can be obtained by a dataflow program when the shared variable formalism illustrated in this work is used. The top-level network of the basic MPEG-HEVC decoder [19] is shown in Fig. 7. This design, used in the following as base line comparison for the shared variable implementation, is composed of 32 actors, 112 buffers, and 745 actor internal variables. The main functional components are a bit-stream parser, motion prediction, intra-prediction, inter-prediction, IDCT, reconstruct coding unit, select coding unit, deblocking filter, sample adaptive offset filter, and decoding picture buffer block. The input to the decoder is a compressed 4:2:0 bit-stream and output is the decoded video sequence. A collection of different HEVC anchor bitstreams [20], with different resolution and quantization parameter (QP) values, has been used as input sequences in order to compare the performances for different design versions. Experiments have been performed on an Intel i7-5960X CPU equipped with 32GB of memory.

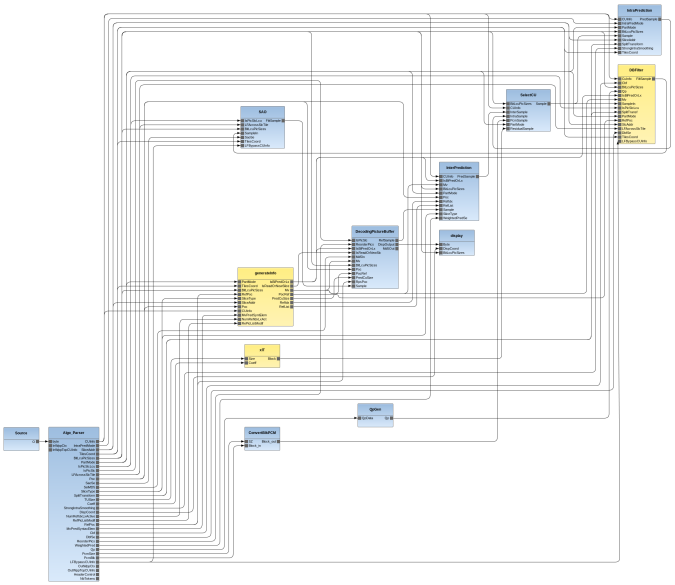


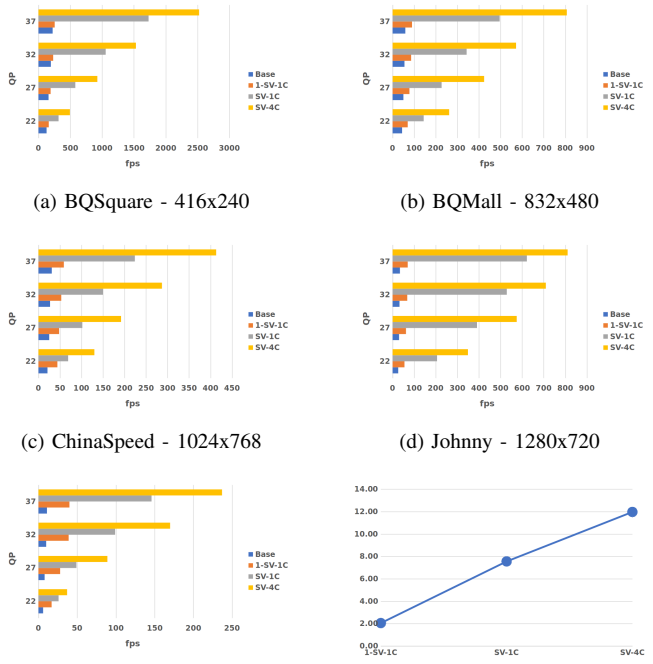
Fig. 7. HEVC RVC-CAL decoder top-level network. Blue boxes are actors, yellow boxes are sub-networks (i.e., composition of actors). The overall design is composed of 32 actors, 112 buffers, and 745 actor internal variables.

The base HEVC design has been initially executed using one core partitioning. Table I summarizes the performance results in terms of frame per second (fps) obtained for bitstreams with a resolution from 416x240 pixels and a variable QP from 22 to 37. Performances vary from 6 fps (i.e., high resolution, small QP) to 223 fps (i.e., small resolution, high QP). The first modification changes the base design by introducing only a single shared variable placed between the deblocking filter and the sample adaptive offset filter since in the original design the entire frame-buffer (i.e., a multidimensional array containing the reconstructed pictures) is transferred from one filter to the other using FIFO queues. As such, increasing the image resolution, the number of transfered and identical data increase accordingly. Performances of this modified design configuration, denoted by 1-SV-1C in Table I, vary from 6 fps (i.e., high resolution, small QP) to 225 fps (i.e., small resolution, high QP). Hence, the second step identifies all variables that could be shared among actors. Buffers that impact on the overall program execution time (and decrease the fps performance as a consequence) were identified using the TURNUS design space exploration framework [21] that the authors have developed, that is based on the analysis of the program post-mortem execution trace graph [22]. This information has been successfully used to highlight parts of the design where (identical) data exchange through FIFO queues can be removed by introducing shared variables among actors. Performances of the new design, denoted by SV-1C in Table I, vary from 29 fps (i.e., high resolution, small QP) to 1030 fps (i.e., small resolution, high QP). The last modification is of the mapping of the SV-1C design to multiple cores in order to verify the effectiveness of this approach also on multi-core configurations. Actors have been partitioned on four different CPU cores using the mapping heuristic algorithms available in the TURNUS frameworks. Performances of this design configuration, denoted by SV-4C in Table I, vary from 37 fps (i.e., high resolution, small QP) to 2523 fps (i.e., small resolution, high QP). The overall performance improvements summary is depicted in Fig. 8.

Input Sequence			Base	1-SV-1C		SV-1C		SV-4C	
Name	Resolution	QP	fps	fps	speed-up	fps	speed-up	fps	speed-up
BQSquare	416x240	22	128	162	1.27	316	2.47	494	3.86
		27	159	191	1.20	579	3.64	925	5.82
		32	195	231	1.18	1056	5.42	1531	7.85
		37	223	255	1.14	1730	7.76	2523	11.31
BQMall	832x480	22	44	70	1.59	144	3.27	262	5.95
		27	50	78	1.56	227	4.54	424	8.48
		32	55	86	1.56	343	6.24	572	10.40
		37	59	90	1.53	496	8.41	807	13.68
ChinaSpeed	1024x768	22	21	44	2.10	69	3.29	130	6.19
		27	25	48	1.92	102	4.08	192	7.68
		32	27	53	1.96	150	5.56	287	10.63
		37	31	59	1.90	224	7.23	413	13.32
Johnny	1280x720	22	26	55	2.12	206	7.92	349	13.42
		27	30	62	2.07	391	13.03	575	19.17
		32	32	68	2.13	529	16.53	710	22.19
		37	34	70	2.06	622	18.29	811	23.85
BQTerrace	1920x1080	22	6	17	2.83	26	4.33	37	6.17
		27	8	28	3.50	49	6.13	89	11.13
		32	10	39	3.90	99	9.90	170	17.00
		37	11	40	3.64	146	13.27	237	21.55

TABLE I

RVC-CAL HEVC DECODER: BASE VERSION, 1 SHARED VARIABLE (1-SV-1C) AND COMPLETE SHARED VARIABLES (SV-1C ON 1 CORE, AND SV-4C ON 4 CORES) DESIGN RESULTS FOR DIFFERENT INPUT SEQUENCES (VARYING RESOLUTION AND QP VALUE).



(a) BQSquare - 416x240 (b) BQMall - 832x480

(c) ChinaSpeed - 1024x768 (d) Johnny - 1280x720

(e) BQTerrace - 1920x1080 (f) Average speed-up

Fig. 8. RVC-CAL HEVC decoder: base version, 1 shared variable (1-SV-1C) and complete shared variables (SV-1C on 1 core, and SV-4C on 4 cores) design results for different input sequences (varying resolution and QP value).

VI. Conclusions

This paper presents shared-variable synchronization approaches for dynamic dataflow programming languages. It combines an `@shared` RVC-CAL language extension with a shared memory controller (SMC) protocol for multiple actors to reuse one memory region for data sharing. The frames per second performance of an HEVC decoder has a $\times 21$ speedup using the approach.

Future work could investigate program analysis on dataflow programs that make use of the *@shared* primitive and the SMC protocol to ensure that 1) all actors using the same shared variable tags are connected to the same SMC, and 2) that these actors conform to the SMC protocol (Section III). We have developed a software implementation of the RVC-CAL language extension that supports shared dataflow variables. In future we intend on developing an FPGA hardware implementation of the *@shared* language construct and the SMC protocol to reduce memory requirements on low memory embedded architectures. The long-term ambition is to abstract the SMC from explicit use in programs i.e. remove the need to implement Section IV-B manually, and instead, infer those steps during compilation for all occurrences of *@shared* tags.

Acknowledgement

We acknowledge the support of the Engineering and Physical Research Council, grant reference EP/N028201/1 (Border Patrol: Improving Smart Device Security through Type-Aware Systems Design).

References

[1] Jeronimo Castrillon and Rainer Leupers, *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*, Springer Publishing Company, Incorporated, 2013.

[2] M. Mattavelli, "MPEG reconfigurable video representation," in *The MPEG Representation of Digital Media*, Leonardo Chiariglione, Ed., pp. 231–247. Springer New York, 2012.

[3] M. Mattavelli, J. Janneck, and M. Raulet, "MPEG reconfigurable video coding," in *Handbook of Signal Processing Systems*, S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds., pp. 43–67. Springer US, 2010.

[4] E. S. Jang, Mattavelli M., M. Preda, M. Raulet, and H. Sun, "Reconfigurable media coding: An overview," *Signal Processing: Image Communication*, vol. 28, no. 10, pp. 1215–1223, 2013.

[5] J. Dennis, "First version of a data flow procedure language," in *Symposium on Programming*, 1974, pp. 362–376.

[6] G. Kahn, "The Semantics of Simple Language for Parallel Programming," in *IFIP Congress*, 1974, pp. 471–475.

[7] E. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987.

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

[9] W. Johnston, J. Hanna, and R. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys (CSUR)*, vol. 36, no. 1, pp. 1–34, 2004.

[10] ISO/IEC 23001-4:2011, "Information technology - MPEG systems technologies - Part 4: Codec configuration representation," 2011.

[11] Tim Harris, James Larus, and Ravi Rajwar, *Transactional Memory, 2Nd Edition*, Morgan and Claypool Publishers, 2nd edition, 2010.

[12] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy, "Composable memory transactions," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2005, pp. 48–60.

[13] Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter, "Transactional Actors: Communication in Transactions," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems*, New York, NY, USA, 2017, SEPS 2017, pp. 31–41, ACM.

[14] V. Gajinov, S. Stipic, O. S. Unsal, T. Harris, E. Ayguadé, and A. Cristal, "Integrating dataflow abstractions into the shared memory model," in *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, Oct 2012, pp. 243–251.

[15] Amith R. Mamidala, Daniel Faraj, Sameer Kumar, Douglas Miller, Michael Blocksome, Thomas Gooding, Philip Heidelberger, and Gábor Dózsa, "Optimizing MPI Collectives Using Efficient Intra-node Communication Techniques over the Blue Gene/P Supercomputer," in *25th International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings*. 2011, pp. 771–780, IEEE.

[16] Praveen K. Murthy and Shuvra S. Bhattacharyya, "Shared Memory Implementations of Synchronous Dataflow Specifications," in *2000 Design, Automation and Test in Europe (DATE 2000), 27-30 March 2000, Paris, France*. 2000, pp. 404–410, IEEE Computer Society / ACM.

[17] S. Casale-Brunet, E. Bezati, and M. Mattavelli, "Programming models and methods for heterogeneous parallel embedded systems," in *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, Sept 2016, pp. 289–296.

[18] "Orcc," http://github.com/orcc/orcc, online, accessed May 2018.

[19] "Orc-Aps," http://github.com/orcc/orc-apps, online, accessed May 2018.

[20] "HEVC Anchor Bitstreams," ftp://ftp.kw.bbc.co.uk/hevc/hm-10.0-anchors/bitstreams/lp_main/, online, accessed May 2018.

[21] Simone Casale-Brunet, "Analysis and optimization of dynamic dataflow programs," 2015.

[22] S. Casale-Brunet and M. Mattavelli, "Execution trace graph of dataflow process networks," *IEEE Transactions on Multi-Scale Computing Systems*, pp. 1–1, 2018.