Profile Guided Dataflow Transformation for FPGAs and CPUs

STEWART, Robert, BHOWMIK, Deepayan <http://orcid.org/0000-0003-1762-1578>, WALLACE, Andrew and MICHAELSON, Greg

Published version

STEWART, Robert, BHOWMIK, Deepayan, WALLACE, Andrew and MICHAELSON, Greg (2015). Profile Guided Dataflow Transformation for FPGAs and CPUs. Journal of Signal Processing Systems, 87 (1), 3-20.

Journal of Signal Processing Systems manuscript No.
(will be inserted by the editor)

Profile Guided Dataflow Transformation for FPGAs and CPUs

Robert Stewart · Deepayan Bhowmik ·
Andrew Wallace · Greg Michaelson

Received: date / Accepted: date

Abstract This paper proposes a new high-level approach for optimising field programmable gate array (FPGA) designs. FPGA designs are commonly implemented in low-level hardware description languages (HDLs), which lack the abstractions necessary for identifying opportunities for significant performance improvements. Using a computer vision case study, we show that modelling computation with dataflow abstractions enables substantial restructuring of FPGA designs before lowering to the HDL level, and also improve CPU performance. Using the CPU transformations, runtime is reduced by 43%. Using the FPGA transformations, clock frequency is increased from 67MHz to 110MHz. Our results outperform commercial low-level HDL optimisations, showcasing dataflow program abstraction as an amenable computation model for highly effective FPGA optimisation.

1 Introduction

FPGAs are successfully used in many application areas, including consumer electronics, the automotive industry, medical imaging, data centres and signal processing. The key advantage of FPGAs over conventional CPUs is configurability. Resource allocation and memory hierarchy on CPUs must perform well across a range of applications, whereas FPGA designs leave many of those decisions to the application designer to optimally use logic gates to implement one specific application. Moreover, they can be significantly faster as their nature supports fine-grained, massively parallel and pipelined execution. Programs for FPGAs are most often specified directly in hardware description languages such as Verilog.

R. Stewart & G. Michaelson
School of Mathematical & Computer Sciences
Heriot-Watt University, Edinburgh, UK
{R.Stewart,G.Michaelson}@hw.ac.uk

D. Bhowmik & A. Wallace
School of Engineering and Physical Sciences
Heriot-Watt University, Edinburgh, UK
{D.Bhowmik,A.M.Wallace}@hw.ac.uk

The lack of abstraction at this low level makes it very difficult to identify opportunities for restructuring FPGA designs to optimise performance. HDL synthesis tools focus on efficiently mapping a hardware design onto specific FPGA models using low-level techniques, but do not restructure algorithmic implementations.

This paper advocates dataflow program modelling as a high-level program representation that is amenable to restructuring using transformation techniques to target both FPGAs and conventional CPUs. The mean shift tracking algorithm [10] is used as a computer vision case study to demonstrate advantages of high level program transformation for FPGAs and CPUs. Mean shift is widely used in tracking segmentation applications, and the algorithmic properties represent interesting challenges for an FPGA implementation. Specifically, it includes a feedback loop and a convergence criteria of varying complexity. This contrasts with feed-forward algorithms composed of synchronised functional blocks with constant latency, which can more straightforwardly be mapped onto FPGAs.

It bridges the knowledge gap between software-competent domain experts who may know little about optimisation and paralellisation, and hardware engineers who may know little about abstract models of computation and software transformation techniques. Most software optimisations are compiler-embedded passes driven by heuristic knowledge of targeted processor architectures and by known combinations of passes that generally work well together in most cases. And whilst dataflow transformations have been embedded into dataflow language compilers *e.g.* [14], the key difference of our approach is that we apply transformations based on architecture specific simulation and trace-based profiling, with a current limitation of requiring manual application of the transformations. The profiles include FPGA clock frequency bottlenecks, FPGA resource utilisation, code traces on CPUs and CPU runtime. We use these profiles with a consideration for the trade-off between hardware space, runtime, redundant computation, clock frequency and throughput.

We show that it is possible to start from a common initial naive dataflow representation of an algorithm, and apply transformations to increase FPGA clock frequency, and separately to shorten CPU runtime. To the best of the authors knowledge, expressing computer visions algorithms as dataflow graphs and profile-driven dataflow transformations targeting FPGAs are both largely unexplored areas.

This paper is an elaboration of earlier work [4], which took the same case study expressed with the CAL dataflow language [12], and optimised it for shorter CPU runtime. Here, we significantly extend that work by first defining a set of dataflow transformations, and obtain profiles using methodical dataflow simulation and FPGA simulation. We then optimise the original dataflow implementation to increase FPGA clock frequency using these transformations. The contributions of this paper are:

- A categorisation of trace based and simulation based methods for dataflow profiling on FPGAs and CPUs.
- A definition of 8 portable dataflow transformations, with discussion on the likely benefits and drawbacks of applying them when targeting CPUs and FPGAs. Additionally, we define two FPGA specific computation transformations.
- Defining and applying various dataflow profiling techniques to optimise a naive dataflow implementation of the mean shift person tracking algorithm for FP-

GAs and CPUs. Actor-by-actor FPGA simulation, holistic dataflow profiling and trace-based profiling are used to identify opportunities for FPGA and CPU improvements.

The paper is structured as follows. Section 2 describes simulation and trace-based dataflow profiling methods. Section 3 presents 8 dataflow and 2 algorithmic transformations. Section 4 describes the mean shift algorithm and presents a dataflow implementation of it. Section 5 describes the mean shift profiling approaches on FPGAs and CPUs, and presents performance results after applying dataflow transformations. Section 6 concludes.

2 Dataflow Profiling

2.1 Dataflow Modelling

A dataflow graph models a program as a directed graph. The model is depicted in Fig. 1. *Tokens* move between asychronously communicating stateful and well defined functional blocks called *actors*. They transform input streams into output streams via *ports*. Ports are connected with *wires*. Inside an actor is a series of fireable sequences of instructions. These instructions are encapsulated within *actions*, and the steps an actor takes determine the which ports tokens are consumed and emitted and also which state-modifying instructions are executed. The conceptual dataflow model of explicit data streaming and functional units maps well onto FPGA design comprising explicit wires and basic building blocks [19].

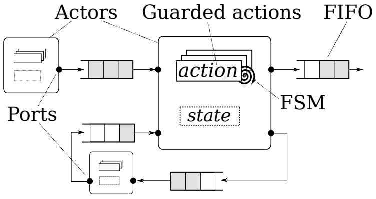
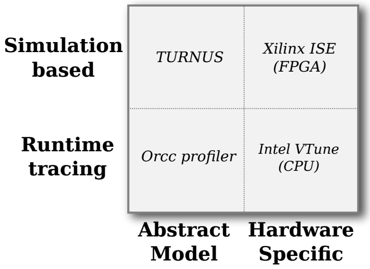


Fig. 1: The Dataflow Process Model

The mean shift implementation and optimisations in this paper are based on the CAL dataflow language [18]. The language includes a number of expressive constructs. A *guard* can be attached to an action to predicate its firing not only on the availability of a token on a given port, but also on its value. The language permits explicit *finite state machine* (FSM) transitions between actions, an implicit predication on firing actions as only actions reachable within one transition in the FSM declaration are fireable. Lastly, *priority* statements declare an inequality between two actions. When there exists more than one transition from a current state and more than one action is fireable, priority statements offer the opportunity for one action to fire over the other. They are often used in conjunction with guards.



	Abstract Model	Hardware Specific
Simulation based	*TURNUS*	*Xilinx ISE (FPGA)*
Runtime tracing	*Orcc profiler*	*Intel VTune (CPU)*

Fig. 2: Choices for dataflow profiling

2.2 Profiling

Profile guided optimisation is used to shorten execution runtimes and to improve hardware resource utilisation across different architectures. Profilers have been embedded into *compilers e.g.* for automatic function inlining [9], into profile guided *JIT compilers* that compile bytecode of frequently used methods to native code [1], into *runtime systems* to reduce memory access latency on NUMA architectures using automatic page placement [21] and into *IDE tool support* for semi-automatic parallel refactorings *e.g.* by introducing parallel algorithmic skeletons [6]. Profilers either *trace* a program's execution on a target hardware platform to identify runtime costs and resource utilisation, or they *predict* runtime costs or resource utilisation without executing machine code. Moreover, profilers can either be scoped to a specific model of computation such as the dataflow model, or can be hardware target specific, *e.g.* profiling FPGAs or CPUs (Fig. 2).

Simulation-based profiling provides contextual information, such as critical path analysis through actors, and abstract computational cost models and cost provenance of actors, actions, private state variables and FIFOs are preserved. Orcc [28] is primarily a compiler for the CAL dataflow language, though it also includes a CPU based dataflow simulator that traces action firings and workload on actors and connections. The Orcc profiler is used to identify bottlenecks in the context of high-level dataflow execution, by identifying actions on the critical path through the dataflow graph and finding where FIFOs are being starved of tokens. TURNUS [8] is a dataflow profiler that evaluates the computational load of actions inside actors in terms of executed operator and control statements and analysis of the critical path through an actor network. Xilinx ISE [27] is a software tool for synthesis, simulation and analysis of HDL designs targeting FPGAs. Synthesis tools can be used to provide simulation profiles for dataflow actors and graphs by compiling to HDL and synthesising separately and then all together. For a particular hardware design, *e.g.* derived from a CAL dataflow graph with a Verilog Orcc backend, it calculates clock frequency and device resource utilisation. In contrast to dataflow simulators, HDL synthesis tools do not preserve the dataflow attributes of actions, private state variables and FIFOs.

Runtime trace-based profiling records program execution on a target architecture. They provide insights into bottlenecks at the source code level, *e.g.* Intel VTune [17] traces C source code performance analysis on x86 CPUs. It reports stack sampling, thread profiling and hardware event sampling. Trace based profilers can assist a programmer in re-writing their source code, or compiler writers generating more efficient code if their source to source compiler targets a system in languages such as C as an intermediate language. We use the Intel VTune profiler to trace CPU clock cycles for every line of C code for both actors and the runtime scheduler that the Orcc compiler generates, and identify bottlenecks in the CPU scheduler and hotspots *within* action implementations.

Trace-based profiling of dataflow programs can complement dataflow simulation tools such as TURNUS. CAL statements such as loops and branching used in action bodies resemble a subset of C, and Orcc compiles CAL actions to very similar C code. CPU bottlenecks for executing actions can therefore be detected by trace-based profilers, and likewise bottlenecks in the C scheduler can be highlighted. The main drawbacks to trace-based profiling are that 1) these tools are unaware of the higher-level dataflow model so no holistic analysis is possible *e.g.* critical path analysis, 2) the code generated by a dataflow compiler is a moving target *e.g.* improved C source code generation from Orcc may shift CPU bottlenecks to other parts of the code, and 3) the traces depend on the input dataset.

3 Dataflow Transformations

Architecture specific compilers such as C compilers for x86 CPUs or HDL tools for synthesising FPGA hardware designs include fine tuned low level optimisations, such as peephole optimisations on CPU assembly or timing optimisations and register replication on HDL for FPGAs. In contrast, our high-level dataflow transformation approach takes place much earlier in the design process, and targets much more coarse grained program restructuring for significant performance improvement. Dataflow graphs are portable high level portable representations of programs and are amenable to coarse grained transformation to exploit conventional multi-threaded and SIMD architectures, and wire-exposed architectures such as FPGAs.

Our dataflow transformations involve composing or decomposing actors and wires between them, summarised in Table 1. They are influenced by previous work on dataflow optimisation and verification, data locality optimisation and parallelism optimisations. The high-level transformation approach is influenced by software restructuring approaches, *e.g.* parallel refactoring tools [7], and injecting parallel algorithmic skeletons [25] into sequential code, *e.g.* transforming a map function into a data-parallel farm. Actor fusion and fission have been proposed elsewhere as dataflow optimisations in [14], named there as *vertical fusion* and *horizontal fission* respectively. Actor pipelining and actor fusion have been proposed as box calculus transformation rules [16], named there as *VCompE* and *VCompI* respectively, for verified program parallelisation. Actor fusion is also influenced by task-coarsening techniques for other programming models, *e.g.* the *loop fusion* compiler pass on the array based SaC language [23], to increase shared-memory locality of previously pipelined loops. Our loop fission transformation is similar to the `for` directive in OpenMP [11] which splits different portions of a loop across multiple threads, though our loop fission transformation is more restrictive as memory access is also partitioned into memory-isolated actors.

	Transformation	Description
1	Actor fusion	Combines multiple actors into a single actor.
2	Actor fission	Replicates an actor multiple times, and incoming data is load balanced between each one.
3	Loop fission	Promotes a loop in an actor to multiple loops in separate actors, and incoming data is load balanced between each one.
4	Actor pipelining	Spreads the computation inside a single actor into set of actors connected in a pipeline.
5	Task parallelism	Separates an expression into sub expressions which is then distributed across multiple actors, The implicit dataflow in the original expression is transformed into explicit dataflow wires.
6	Loop elimination	Removes loops where it is possible to do so.
7	FSM simplification	Factors away states (*i.e.* actions) that always become unreachable beyond progression through an actors finite state machine.
8	Built-in constructs	Replaces code with equivalent built-in language constructs that that are better supported with optimised scheduling or implementation strategies.

Table 1: Dataflow Transformations Summary

Transformation 1 *Fusing two finite state machines into one actor*

```
ACTOR A In ⟹ Out :
  var z;
  act0: action In:[x] ⟹
    z := x-1;
  act1: action ⟹ Out:[z]
  fsm s0: s0 (act0) --> s1;
          s1 (act1) --> s0;

ACTOR B In ⟹ Out :
  var z;
  act0: action In:[x] ⟹
    z := x+2;
  act1: action ⟹ Out:[z]
  fsm s0: s0 (act0) --> s1;
          s1 (act1) --> s0;
```

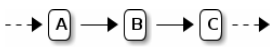
⟹

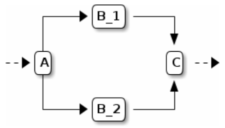
```
ACTOR A In ⟹ Out :
  var z, z';
  act0: action In:[x] ⟹
    z := x-1;
  act1: action ⟹
    z' := z+2;
  act2: action ⟹ Out:[z']
  fsm s0: s0 (act0) --> s1;
          s1 (act1) --> s2;
          s2 (act2) --> s0;
```

3.1 Graph Transformations

Actor fusion (T1) increases the granularity of actors by consolidating instructions distributed amongst a set of actors into a single actor. An example is shown in Transformation 1. The primary motivation is to increase the ratio of computation and communication time by fusing computationally inexpensive actors, as instruction sequences in an single actor can communicate via shared memory rather than with FIFOs. However, fusion increases the memory requirements and potentially reduces the extent of pipelined parallelism. Another drawback is code reuse, due to the aggregation of previously well defined modular functional blocks. In the context of dataflow, fusion is achieved by combining the finite state machines of multiple actors into a finite state machine of a single actor whilst preserving the causally ordered transition interactions between the original FSMs.

Transformation 2 *Data parallelism with actor fission*





```
𝒜𝒞𝒯𝒪ℛ A In ⟹ Out :
  action In:[x] ⟹ Out:[x]

𝒜𝒞𝒯𝒪ℛ B In ⟹ Out :
  action In:[x] ⟹ Out:[ℱ x]

𝒜𝒞𝒯𝒪ℛ C In ⟹ Out :
  action In:[x] ⟹ Out:[x]
```

$\Longrightarrow$

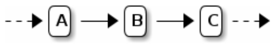
```
𝒜𝒞𝒯𝒪ℛ A A ⟹ O1,O2 :
  action In:[x1,x2] ⟹ O1:[x1], O2:[x2]

𝒜𝒞𝒯𝒪ℛ B_1 A ⟹ Out :
  action In:[x] ⟹ Out:[ℱ x]

𝒜𝒞𝒯𝒪ℛ B_2 A ⟹ Out :
  action In:[x] ⟹ Out:[ℱ x]

𝒜𝒞𝒯𝒪ℛ C A ⟹ Out :
  action In:[x] ⟹ Out:[x]
```

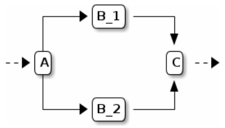
Actor fission (T2) is a data-parallelism optimisation that replicates actors and load-balances tokens between each replica. This transformation can reduce register pressure and exploit data-parallelism using multithreaded CPU runtime systems, and also exploit fine-grained parallel communication on FGPAs. An example is in Transformation 2. $\mathcal{ACTOR}B$ is replaced with $\mathcal{ACTOR}B\_1$ and $\mathcal{ACTOR}B\_2$ thanks to two determinate properties for $\mathcal{ACTOR}B$. First, the actor maps $\mathcal{F} : X \rightarrow Y$ for all input tokens $x : X$ and output tokens $y : Y$, where $\mathcal{F}$ is pure (*i.e.* side-effect free). Second, the single action implies a deterministic transition sequence through finite states.

Loop fission (T3) is a second data parallelism transformation that promotes a loop statement in an actor to multiple data-parallel actors. An example is in Transformation 3. The original $\mathcal{ACTOR}A$ consumes 50 tokens and finds the maximum value using a loop. In the transformed version, the loop is removed from $\mathcal{ACTOR}A$ which now fans out 25 tokens each to actors $\mathcal{ACTOR}B\_1$ and $\mathcal{ACTOR}B\_2$ who find the maximum value in their array partition. $\mathcal{ACTOR}C$ reduces those results by comparing the two numbers to find the largest. The control instructions around the original loop structure must be modified to account for the fewer iterations of each loop. Due to loop body code analysis, loop fission is a more complex transformation than actor fission, as code analysis must be used to ensure that loop-carrying operations do not cross between the separated loops in isolated actor memory, *e.g.* random access to an array.

Pipelining computation (T4) spreads the execution of multiple instructions across actors connected in a pipeline. Pipelining transformations can be applied at three levels of granularity. From largest to smallest: 1) an actor's FSM can be pipelined into numerous FSMs each containing fewer actions; 2) a sequence of statements in an action can be split into actions potentially in separate pipelined actors; and 3) instructions in a loop statement can be pipelined across multiple loops potentially in separate pipelined actors. Pipelining instructions can reduce memory requirements of an actor or ease register pressure in loops that contain many assignments. The transformation favours FPGAs for high-throughput fine-grained pipelined computation, and avoids utilising scarce FPGA memory. Computation

Transformation 3 *Data parallelism with loop fission*



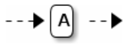


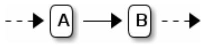
```
ACTOR A In ⟹ Out :
  var max := 0;
  action In:[arr] repeat 50 ⟹ Out:[max]
    for i in 1..50
      max := arr[i] > max ? arr[i] : max
```

⟹

```
ACTOR A In ⟹ Out1, Out2 :
  action In:[arr] repeat 25 ⟹
         Out1:[x] repeat 25
  action In:[arr] repeat 25 ⟹
         Out2:[x] repeat 25

ACTOR B_1 In ⟹ Out :
  var max := 0
  action In:[arr] repeat 25 ⟹ Out:[max]
    for i in 1..25
      max := arr[i] > max ? arr[i] : max

ACTOR B_2 In ⟹ Out :
  var max := 0
  action In:[arr] repeat 25 ⟹ Out:[max]
    for i in 1..25
      max := arr[i] > max ? arr[i] : max

ACTOR C In1, In2 ⟹ Out :
  action In1:[x], In2:[y] ⟹ Out:[z]
    z := x > y ? x : y
```

Transformation 4 *Loop pipelinig*





```
ACTOR A In ⟹ Out :
  var a, b;
  action ⟹ Out:[y] repeat 100
    for i in 1..100
      a    := $E_1$
      b    := $E_2$
      y[i] := a + b
```

⟹

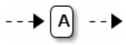
```
ACTOR A In ⟹ Out :
  action ⟹ Out:[a] repeat 100
    for i in 1..100
      a[i] := $E_1$

ACTOR B In ⟹ Out :
  action In:[a] repeat 100 ⟹
         Out:[y] repeat 100
    for i in 1..100
      y[i] := a[i] + $E_2$
```

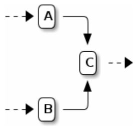
pipelining shifts the balance from computation to communication, which may hinder processing elements such as CPUs that would incur scheduling overheads of very frequent context switching of fine-grained actors.

Loop pipelining breaks a single loop into multiple loops. Each new loop has the same number of iterations as the original, but contains a subset of the statements of the original loop [20]. The technique is used to create subloops with fewer dependencies, improve instruction cache locality due to shorter loop bodies and reduce memory requirements by iterating over fewer arrays [2]. The loop pipelining transformation is permitted both in the absence of a flow dependence or for the example in Transformation 4 if there is a $E_1 \rightarrow E_2$ dataflow dependence in the loop, *i.e.* `a` appears in E_2.

Task parallelism (T5) separates an expression E into separate actors each containing one or more redexes e_i, where E is composed of reducible expressions (redexes) $\{e_1, .., e_n\}$. An example is in Transformation 5. The transformation extracts the implicit dataflow between redexes, and generates an explicit dataflow connection between the newly created actors. The primary benefit of this transformation is the

Transformation 5 *Decompose expression into task parallel actors*



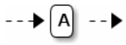


```
ACTOR A ⟹ Out :
  action ⟹ Out:[y]
    y := (e1 · e2 · e3)
```

⟹

```
ACTOR A ⟹ Out :
  action ⟹ Out:[e1]

ACTOR B ⟹ Out :
  action ⟹ Out:[e2 · e3]

ACTOR C In1, In2 ⟹ Out :
  action In1[x], In2:[y] ⟹ Out:[x · y]
```

Transformation 6 *Replace a loop with a streaming action*



```
ACTOR A In ⟹ Out :
  var arr[100]
  act0: action In:[x] repeat 100 ⟹
    arr := x;

  act1: action ⟹
    for i in 1..100
      arr[i] := arr[i-1] + i  + 4

  act2: action ⟹ Out:[x] repeat 100
    x := arr;

  fsm s0: s0 (act0) --> s1;
          s1 (act1) --> s2;
          s2 (act2) --> s0;
```

⟹

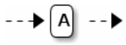
```
ACTOR A In ⟹ Out :
  var carry, idx = 0
  act0: action In:[x] ⟹
    idx := 0
    carry := x

  act1: action In:[x] ⟹ Out:[y]
  guard idx < 100
    y := carry + x + 4
    carry := x

  fsm s0: s0 (act0) --> s1;
          s1 (act1) --> s1;
          s1 (act1) --> s0;

  priority act1 > act0
```

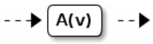
potential for parallel reduction of any redexes with no implicit data dependence between them. The transformation extracts explicit wires between implicit dataflow which can be pipelined as fine-grained computation on FPGAs, or mapped across CPU cores as coarse-grained threads. Thanks to the absence of side-effects when reducing each pure redex, parallel execution is thread-safe.

Loop elimination (T6) removes the outer-most for loop, which is replaced with a streaming action. An example is in Transformation 6. The cost of loops for FPGAs is the memory requirements of intermediate data structures, and the latency between token consumption and token emission caused by executing loop iterations, which may reduce throughput on FPGAs. The primary benefit of loop statements inside actors is the potential for SIMD vectorisation on CPUs & GPUs in the absence of flow dependence between statements inside the loop.

The transformation targets actors with multiple actions that 1) consume tokens into an intermediate data structure *e.g.* an array, 2) traverses the data structure, and 3) emits tokens. The transformation can be applied in the absence of loop-carrying dependence between loop iterations. It creates a single action that consumes a single token, applies each instruction in the original loop body, then emits a token. For the loop elimination example in Transformation 6, the loop in action `act1` is replaced with an action that computes and emits token values on-the-fly. The iteration dependence in `arr[i-1]` is carried in the scalar variable `carry`.

Transformation 7 *FSM minimisation factoring out eventually unreachable states*





```
ACTOR A In ⟹ Out :
 var v
 act0: action ⟹
   v := E

 act1: action In:[x] ⟹ Out:[y]        ⟹      ACTOR A(var v) In ⟹ Out :
   y := x + v                                   act0: action In:[x] ⟹ Out:[y]
                                                  y := x + v
 fsm s0: s0 (act0) --> s1;
         s1 (act1) --> s1;
```

Finite state machine minimisation (T7) identifies actions in a FSM that will eventually become unreachable. An actor's FSM is defined as $(\Sigma, S, s_0, \delta, F)$, where Σ is the non-empty input token set, S is a finite non-empty set of states, s_0 is the initial state, $\delta : S \times \Sigma \rightarrow \mathcal{P}(S)$ is non-deterministic state transition function to a power set S, and F is the set of final states. FSM minimisation transformation targets an action rule $\delta : S \times \Sigma \rightarrow \mathcal{P}(S')$ such that the linear temporal logic formula $(S' \rightarrow \Box \neg S)$ holds for all input token sequences $t_s \in \Sigma$ for S. That is, action S cannot be removed at compile-time with dead action elimination but eventually becomes unreachable after a transition sequence. An unreachable action is never executed and yet remains an overhead for scheduling, clock synchronisation and hardware resource. Statically analysing an actor to discover such actions could be used to factor them away from streaming-based FPGAs to other processing elements on heterogeneous architectures. An example is in Transformation 7, which factors out the one-time computation of expression E, *i.e.* action `act0`. Instead, $\mathcal{ACTOR}A$ is parameterised on the pre-computed value of E and the FSM is removed.

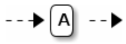
Built-in constructs (T8) replaces code with behaviourally equivalent built-in language constructs, that reduces scheduling or area requirements. Opting for one implementation technique over another can have dramatic effects on hardware space and runtime, because the implementation of each language feature is entirely target and compiler-backend specific. An example is shown in Transformation 8. It emits all values in a local array of 100 elements. A finite state machine could be used to visit an action that emits a single token 100 times, or use CAL's `repeat` construct *i.e.* the transformed version for atomic emission of 100 tokens. The first approach falls back to the scheduler 100 times whilst the second only falls back to the scheduler after broadcasting all array values.

3.2 Computational Transformation

We now consider two FPGA optimisation techniques. They imply changes to parts of the implementation of an algorithm, rather than transformations to the dataflow graph structure.

Floating point to integer based conversion replaces floating point operands of arithmetic instruction with integer based equivalents. Many scientific applications depends on both the dynamic range and high precision of IEEE double-precision

Transformation 8 *Emitting a token sequence atomically with* `repeat`



```
𝒜𝒞𝒯𝒪ℛ ⟹ Out :
  var i := 0, arr[100]
  act0: ⟹ Out:[arr[i]]
  guard i < 100
    i++

  act1: action ⟹
    die

  fsm s0: s0 (act0) --> s0,
          s0 (act1) --> s0;

  priority act0 > act1
```

⟹

```
𝒜𝒞𝒯𝒪ℛ ⟹ Out :
  var arr[100]
  act0: action ⟹ Out:[arr] repeat 100

  act1: action ⟹
    die

  fsm s0: s0 (act0) --> s1,
          s1 (act1) --> s1;
```

floating point to maintain numerical stability [26]. Hosting such applications with high precision requirements on FPGAs is an active research area *e.g.* [15]. Nevertheless, integer based arithmetic requires significantly less area to implement and run significantly faster than IEEE formats [13], and some FPGA-targeting compilers do not support floating point operations. The floating point to integer based arithmetic transformation is safe to apply when the precision loss incurred by integer conversion does not comprimise algorithmic robustness. This is the case for our mean shift case study where precision loss is tolerable, which is discussed in Section 5.1.

Lookup tables (LUTs) replace runtime computation with pre-calculated values. They can save on processing time or expensive hardware resource requirements, since retrieving a value from storage is often cheaper than executing the computation. LUTs can either be accumulated at runtime by memoizing values returned from functions with identical input parameters, or by computing values offline. LUTs can increase throughput on FPGAs, where expensive computation in an actor may limit clock frequency of the dataflow graph synthesis. The use of LUTs is limited to the predefined lookup keys. We use a lookup table optimisation in Section 5.1 for pre-computing square root, generated using a program that generates lookup tables in CAL from a user defined $Int \rightarrow Int$ lookup function, which is available online[1].

4 Dataflow Modelling of Meanshift Tracking

Mean shift [10] is a feature-space analysis technique for locating the maxima of a density function. An example of applying mean shift to image processing for visual tracking is shown in Fig. 3. The target is successfully tracked from the initial frame on the left, to the final frame on the right. The algorithm is a kernel based method normally applied using a symmetric Epanechnikov kernel within a pre-defined elliptical or rectangular window. The target region of an initial image is modelled with a probability density function (a colour histogram) and identifies a candidate position in the next image by finding the minimum distance between models using an iterative procedure. A summary is given in Algorithm 1.

[1] `https://github.com/robstewart57/cal-lookuptable-gen`

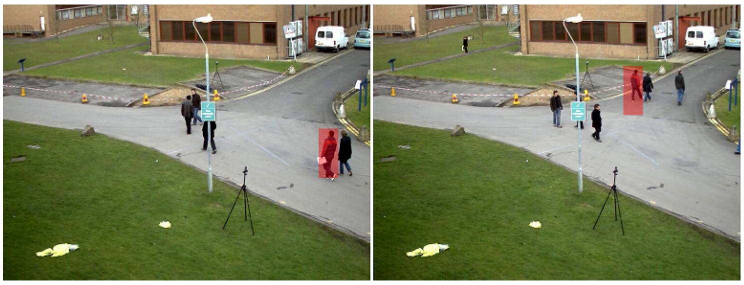


Fig. 3: Example of single target mean shift visual tracking.

```
Input: Target position $y_0$ on $1^{st}$ frame;
Compute Epanechnikov kernel;
Calculate target color model $q_u(y_0)$
        (e.g. using RGB color histogram);
repeat
    Input: Receive next frame;
    Calculate target candidate color model: $p_u(y_0)$;
    Compute similarity function $\rho(y)$ between $q_u(y_0)$ & $p_u(y_0)$;
    repeat
        Derive the weights $\omega_i$ for each pixel
                    in target candidate window;
        Compute new target displacement $y_1$;
        Compute new candidate colour model $q_u(y_1)$;
        Evaluate similarity function $\rho(y)$ between $q_u(y_0)$ & $p_u(y_1)$;
        while $\rho(y_1) < \rho(y_0)$ do
            Do $y_1 \leftarrow 0.5(y_0 + y_1)$;
            Evaluate $\rho(y)$ between $q_u(y_0)$ & $p_u(y_1)$;
        end
    until $|y_1 - y_0| < \epsilon$ (near zero displacement);
    Output: $y_1$ (Target position for current frame);
    Set $y_0 \leftarrow y_1$ for next frame;
until end of sequence;
```

Algorithm 1: Summary of Mean-shift tracking

4.1 Functional Decomposition with Dataflow & Actors

Our dataflow CAL implementation of mean shift is a port of an existing sequential implementation in C++ [5]. Coarse grained functional components were decoupled and mapped into separate actors, shown in Fig. 4.

The input frames are streamed through the *Stream_to_YUV_0* actor which separates the Y, U and V channels before applying the YUV to RGB color space conversion filter implemented in *YUV2RGB_TPP*, a nested network of actors as shown in Fig. 5. The Epanechnikov kernel and its derivatives are calculated in *kArray_evaluation* and *kArray_derv*, respectively. Their constant values are computed once because they depend only on the size of the target window. These values are passed as streaming tokens cyclically to the *update_model* and *displacement* actors. The *update_model* actor calculates the colour models $q_u(y_0)$ & $p_u(y)$, and the histogram as a collection of bins. The histogram function is used to assign a particular RGB value to a bin in the feature space using the 3 values as an index into a 3D space modelled using a 1D array. Each bin u in the model is a normalised sum of all kernel values for the pixels in that bin.

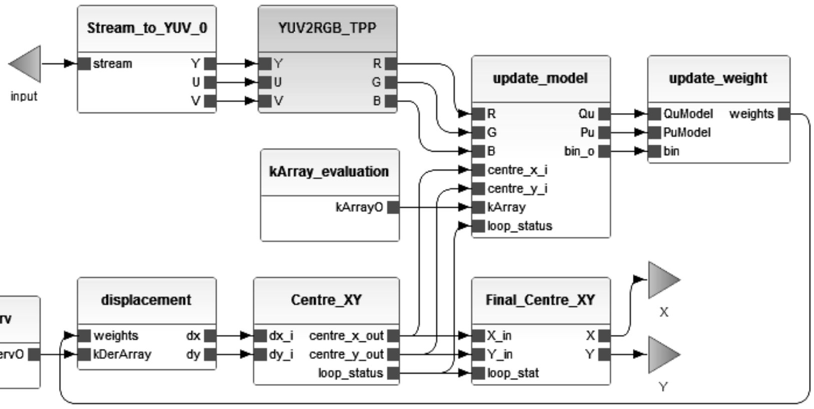


Fig. 4: Dataflow expression of mean-shift tracking

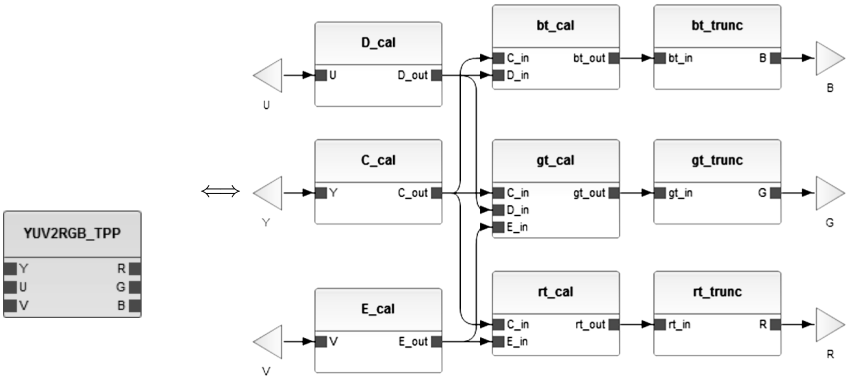


Fig. 5: Expanded actor network

Once the model q_u is calculated for the initial centre position on the first frame, subsequent frames are used to calculate the displacement y_1 on each frame using a feedback loop representing steps in Algorithm 1. The *update_weight* actor derives the weights w_i for each pixel in the target candidate window, while the *displacement* actor computes the displacement y_1 in Eq. (1), where N is the number of pixels in the target window, x is each pixel's relative position, its weight w_i and $g()$ is the kernel derivative function.

$$y_1 = \frac{\sum_{i=1}^{N} x_i w_i g()}{\sum_{i=1}^{N} w_i g()}, \tag{1}$$

This is iterated by actors *Centre_XY* and *Final_Centre_XY* until the convergence criteria ($|y_1 - y_0| < \epsilon$) is met using a feedback loop controlled by boolean tokens passed to the *loop_status* port in the *update_model* actor. Finally *Final_Centre_XY* emits the (x, y) location of the tracked window in consecutive frames.

5 Target Specific Mean Shift Optimisation

This section combines the profiling techniques from Section 2 with the transformations from Section 3, to increase FPGA clock frequency and reduce CPU runtime from the same naive mean shift tracking implementation as a common starting point. The frame size and tracking window size are statically defined, at 176×144 and 20×26 respectively. Orcc [28] is used to compile each CAL dataflow graph to Verilog for FPGAs and C for CPUs, as shown in Fig. 6. The results in this section and the final optimised mean shift implementation, is available in an open access dataset archive [24].

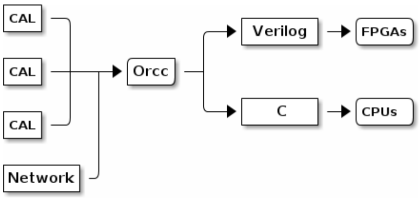


Fig. 6: Compiling CAL to FPGAs & CPUs

5.1 FPGA Optimsations & Results

Supporting floating point arithmetic using IEEE standard libraries on FPGAs comes at the cost of both area and speed (Section 3.2). Moreover, the Orcc backend we are using to generate Verilog does not support floating point operations, so an integer-based adaptation was required. Thankfully, the accuracy loss using integer based operations for mean shift is tolerable for our sample data, so objects can still be tracked. Fig. 7 shows the loss of tracking precision due to floating to integer conversion with a tracking window of 20 × 26. The error is measured by calculating the Euclidean distance between tracking positions from integer based implementation against benchmark floating point implementation. The maximum actual distance for the sample dataset is 8.06 and stays within this tolerable range for the remaining frames. The **floating point to integer** based transformation has been applied to all FPGA based mean shift implementations.

To target FPGAs, the mean shift derivations are compiled with Orcc using Xronos [3], a Verilog backend that generates an FPGA hardware design from the application. The Xilinx ISE software is used to synthesise the Verilog for the Xilinx XC72100-2FFG1156 board, which has 554800 Slice Registers, 277400 Slice LUTs, 755 BRAMs and 2020 DSPs.

FPGA maximum clock frequency is obtained from Xilinx ISE after synthesising the design. The FPGA synthesis results of the naive mean shift implementation is in Table 2, showing both the overall program clock frequency ad also actor-by-actor clock frequency. The range in clock frequencies is 721.5MHz and 55.4MHz. The clock frequency of the entire program is 55.4MHz, and is constraint by the slowest actor. The actors we targeted were prioritised starting from actors with the slowest clock frequency.

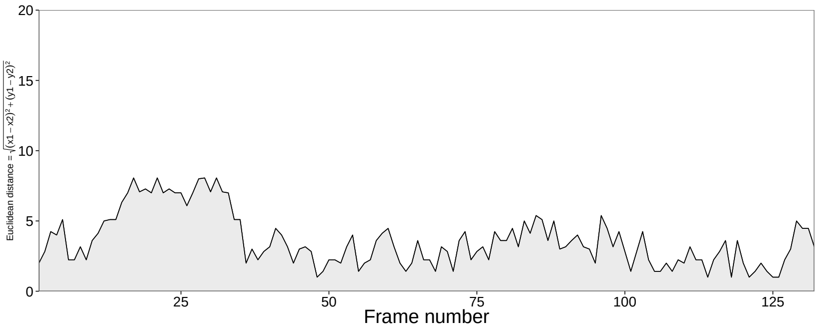


Fig. 7: Tracking Precision Loss after the Floating Point to Integer Transformation

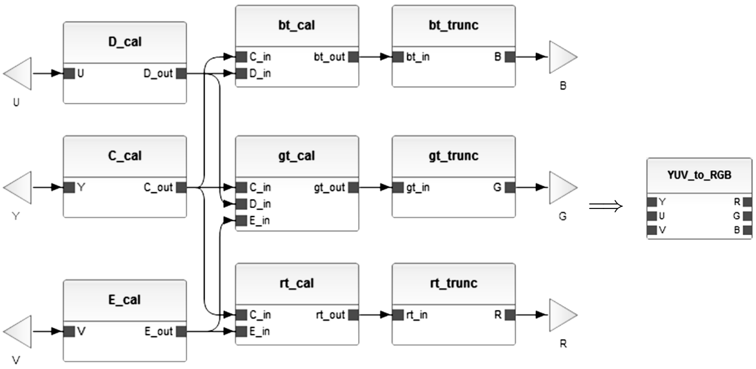
	Slice LUT	Slice registers	Block RAM /FIFO	DSP48E	FMax (MHz)
Naive	**3664**	**8777**	**88**	**49**	**55.41**
Final_XY	76	80	0	0	721.48
Centre_XY	182	199	0	0	530.81
Stream_to_YUV	90	287	24	0	420.07
update_model	1042	2399	30	0	148.74
YUV2RGB	300	957	7	0	126.71
displacement	545	1326	2	9	73.40
update_weight	556	1544	14	4	66.46
kArray_derv	437	1074	1	18	55.44
kArray_evaluation	460	1148	1	18	55.41

Table 2: FPGA Synthesis Results for the Naive Version

5.1.1 Applying FPGA Transformations

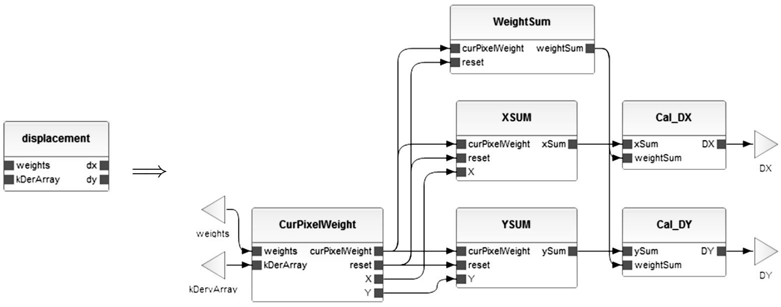
The **loop elimination** transformation is applied to *Stream_to_YUV*. The synthesis of the original representation used 90 registers, 287 slice LUTs, 24 BRAMs and ran at 420MHz. Originally, the stream for an entire frame was completely consumed and organised with a loop into three two dimensional arrays corresponding to the frame's shape for the Y, U and V channels before propagating through the graph. In the transformed version, the loop was eliminated by switching to FSM scheduling to propagate stream values into three separate output ports for each of the three colour channels. This eliminates all intermediate data structures from the actor. The change in number is 27 registers (down 70%), 85 slice LUTs (down 70%), and no BRAM. Although the clock frequency is reduced to 386.7MHz (down 7.9%), this remains higher than the algorithm's overall final clock frequency of 110MHz.

The **actor fusion** transformation is applied to 9 actors that together convert YUV pixels to RGB pixels. The original version is in Fig. 8a, and the transformed version is in Fig. 8b. The synthesis of the original representation used 300 registers, 957 slice LUTs, 7 BRAMs and ran at 420MHz. The transformation fused the finite state transitions between these actors into a single action in the new *YUV_to_RGB* actor that consumes the Y, U and V values and emits the computed R, G and B values on-the-fly. The new FPGA numbers are 99 registers (down 67%), 353 slice LUTs (down 63%), no BRAM and a clock frequency of 182.8MHz (up 44%). This fusion transformation contrasts with the commonly used task and pipelined



(a) before (b) after

Fig. 8: Applying **actor fusion** to mean shift

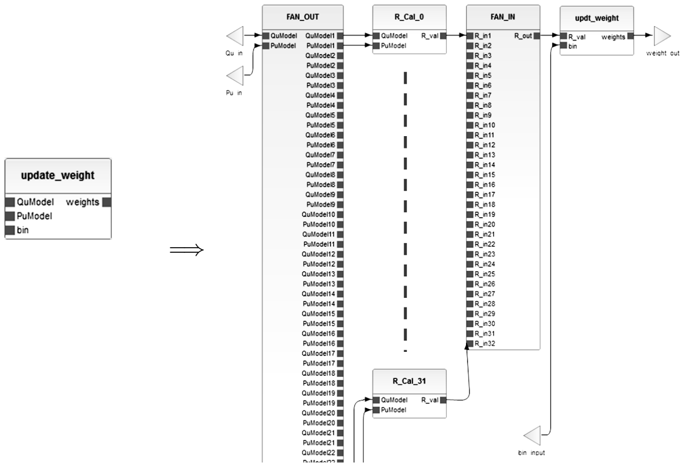


(a) before (b) after

Fig. 9: Applying **task parallelism** to mean shift

parallelism optimisations targeting FPGAs. However, in this case the costs of clock synchronisation, increased memory requirements, and the requirement of implementing 10 additional wires connecting 9 lightweight actors outweighs the benefits of parallel execution of actors with no flow depending path. Therefore, it is important to strike a balance between parallel task granularity and communication overheads.

The **task parallelism** transformation is applied to *Displacement*. The original version is in Fig. 9a, and the transformed version is in Fig. 9b. The synthesis of the original representation used 545 registers, 1326 slice LUTs, 2 BRAMs, 9 DSPs and ran at 73.4MHz. The transformation decomposes Eq. (1) into six redexes in six separate actors. *CurPixelWeight* maintains the interface with preceding actors using ports *weights* and *kDervArray*, and broadcasts the current pixel weight, and X and Y values to three new actors *WeightSum*, *XSUM* and *YSUM*. *XSUM* and *YSUM* compute the numerator of Eq. (1) in the X and Y direction respectively. The *WeightSum* calculates the denominator of the equation used by actors *Cal_DX* and *Cal_DY* to compute the displacement y_1. The new FPGA numbers are 791



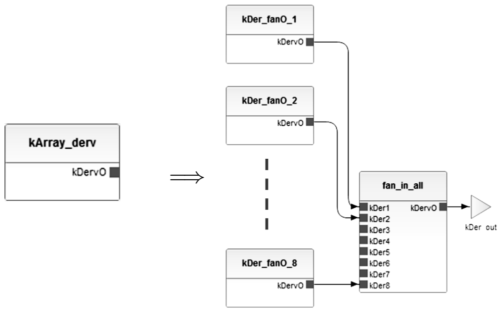
(a) before (b) after

Fig. 10: Applying **actor fission** to mean shift

registers, 1210 slice LUTs, 7 BRAMs, and 9 DSPs. Clock frequency is increased to 110MHz (up 50%).

The **fission** and **lookup table** transformations are applied to *Update_Weight*. The original version is in Fig. 10a, and the transformed version is in Fig. 10b. The synthesis of the original representation used 556 registers, 1544 slice LUTs, 14 BRAMs, 4 DSPs and ran at 66.5MHz. The new FPGA numbers are 12352 registers, 19878 slice LUTs, 55 BRAMs and 128 DSPs. The clock frequency is 72.5MHz (up 9%). The fission transformation creates 32 new actors, whose computation include a square root calculation. Each of those actors use 346 registers, 548 slice LUTs, no BRAM and 4 DSPS. The clock frequency is 72.5MHz. The execution is dominated by the square root calculation. We apply the **lookup table** transformation to replace the computation code consisting of a nested while loop with an array-based square root lookup table. The new FPGA numbers for each instance of this actor are 139 registers, 227 slice LUTs, and no DSPs. The clock frequency is 368.2MHz, up by 408%, at the expense using 32 BRAMs. Combining the fission and lookup table transformation, the FPGA numbers for implementing the *update weights* mean shift functionality are 7907 registers, 38544 slice LUTs, 1028 BRAM and no DSPs. The new clock frequency is 225.9MHz, an increase of 240%.

The **loop promotion** transformation is applied to the *kArray_Derive* actor. The original version is in Fig. 11a, and the transformed version is in Fig. 11b. The synthesis of the original representation used 437 registers, 1074 slice LUTs, 1 BRAM, 18 DSPs, and the clock frequency was 55.4MHz. There was a loop that traversed over all $20 \times 26 = 520$ positions in the rectangular tracking window in the *kArr_derv* actor. There is no iteration dependence in this loop, *i.e.* all values are computed independently, and the transformation partitions the two dimensional space of positions into 8 data-parallel actors. A new actor *fan_in_all* reduces all computed values before propagating through the graph. The new FPGA numbers are 4447 registers, 12484 slice LUTs, 5 BRAM, 144 DSPs, and the clock frequency



(a) before (b) after

Fig. 11: Applying **loop promotion** to mean shift

Functionality	Transformation	Registers	Slice LUTs	BRAM	DSP	Clock (MHz)
Stream to YUV	None	90	287	24	0	420.0
	Loop elimination	27	85	0	0	386.7
YUV to RGB	None	300	957	7	0	126.7
	Actor fusion *(Fig 8)*	99	353	0	0	182.8
Displacement	None	545	1326	2	9	73.4
	Task parallelism *(Fig 9)*	791	1210	7	9	110.0
Update weight	None	556	1544	14	4	66.5
	Fission *(Fig 10)*	12352	19878	55	128	72.5
	Just square root (none)	346	548	0	4	72.5
	Square root Lookup	139	227	32	0	368.2
	Combined	7907	38544	1028	0	225.9
k-array derive	None	437	1074	1	18	55.4
	Loop promotion *(Fig 11)*	4447	12484	5	144	52.7

Table 3: Transformation effects on FPGA results

was 52.7MHz. The slight clock frequency reduction may be due to the data partition size of 65 positions for each data-parallel actor may be too small to benefit from parallel execution.

The computation related to the Epanechnikov kernel, *i.e.* its evaluation and derivation, is required only once before computing mean shift on the first frame. The *kArray_derv* and *kArray_evaluation* actors therefore do not constitute the streaming part of the naive dataflow graph. The **FSM simplification** transformation is applied to factor out the Epanechnikov kernel computation from the implementation, leaving a significant part of the mean shift algorithm (Algorithm 1) on the FPGA.

5.1.2 Applying All FPGA Transformations

The effects of FPGA resource utilisation and clock frequency for each transformation are shown in Table 3. The loop elimination and actor fusion transformations have reduced the FPGA resource utilisation. The actor fusion, task parallelism, fission and lookup table transformations increase clock frequency by 56.1MHz, 36.6MHz, 6MHz and 295.7MHz respectively. For our case study, loop promotion

	Slice LUT	Registers	BRAM	DSP48E	Clock (MHz)
Naive version	2751	6582	86	13	66.5
Optimising HDL for speed	2751	6635	86	13	66.5
Optimising HDL for area	2748	6610	87	13	66.5
Dataflow optimisations	10786	51267	1026	9	110.0

Table 4: Comparison of Dataflow and HDL Level Optimisation Results

and loop elimination reduces clock frequency marginally. However, the loop elimination slowdown does not effect overall clock frequency and the loop promotion transformation is factored away with the FSM simplification transformation. The goal of our transformations is performance, by increasing clock frequency so that overall image data throughput increases as well. Applying all transformations increased the area used for the FPGA configuration. We attribute the size increase to a change in the dataflow structure, *e.g.* transformations requiring more space to accommodate more actors. In spite of the bigger size, clock frequency increased to 110MHz. Clock frequency of a synthesised dataflow graph is determined by the longest combinatorial path in the FPGA configuration. This critical path is determined by the computational complexity and implicit data dependencies within actions inside actors, whose impact on frequency has been reduced by the transformations.

The effectiveness of the transformations in Section 3 are compared with lower level optimisations in the HDL synthesis tool Xilinx ISE version 14.7. The Xilinx ISE results are obtained by applying two HDL optimisations to the Verilog generated by the Orcc compiler from the naive version, one targeting increased speed and another targeting reduced area. A significant partition of the naive mean shift version is used as a starting point for the HDL optimisations and the dataflow optimisations in Table 3, with the Epanechnikov kernel computations excluded and assumed computable with a known tracking window size offline. The ISE optimisations are unable to improve either resource utilisation or the clock frequency which remains at 66.5MHz. In contrast, our higher level dataflow transformations increase clock frequency to 110MHz, an improvement of 65%.

5.2 CPU Optimsation & Results

The goal of dataflow optimisations for CPU targets was shortening runtime for tracking a single target over 130 YUV444 frames from a standard tracking sequence from PETS dataset [22] (S2.L1). The nature of CPUs is very different to FPGAs, namely that modern CPUs have up to 64 cores that all share memory. Our dataflow transformations aim for more coarsely granulated parallel actors to match the degree of multicore CPU parallelism, rather than the fine-grained parallelism of FPGAs. CPUs are targeted with the Orcc compiler's C backend, which generates C for each actor and also emits a multi-threaded scheduler, also implemented in C. It creates one Operating System thread per CPU core and assigns each an actor pool, and partitions the actors across these actor pools. Trace-based profiling with Intel VTune is used on both actor implementations and the Orcc runtime system — enabling the possibility of optimising both. All CPU runtimes are on an Intel

Core 2 Quad CPU at 2.8GHz with 6Gb DDR3 memory, running the 64bit Linux 3.15 kernel, and the C was compiled with gcc 4.8.3.

The mean runtime of computing the naive mean shift version was 24.6s. The profile reported 41% of overall runtime was executing an actor responsible for writing YUV frames to file. We modified the IO operations in the Orcc **runtime system** by replacing costly `fseek` and `fwrite` calls with a `putc` call, and the Orcc compiler was patched with this fix. The mean runtime using this fix is 2.97s, a speedup of 8.3 reducing runtime by 88%. All remaining runtimes are measured using this scheduler optimisation.

In the runtime system, FIFOs are implemented as lock-free C `struct`s shared between two actors. These become bottlenecks if the two actors reside in the same actor pool, or if the computational granularity of the actors is too small. The naive version uses three separate actors to *1)* draw a tracking rectangle, *2)* convert RGB values to the YUV colour space and *3)* merge individual YUV channels into a single stream. The profiler reports runtimes of 0.25s, 0.26s and 0.29s respectively — a total of 0.8s. The **actor fusion** transformation is applied to eliminate the two intermediate FIFOs. These three actors are fused into a new single actor, and profiling reports a runtime of 0.33s for this actor — an actor runtime reduction of 59%. It reduced program runtime by 10%.

By observing the dependencies between actors, and between actions within actors, the *displacement* actor was identified as an optimisation candidate. This actor computes the displacement function in Eq. (1). The **task parallelism** transformation is the same as the one applied targeting FPGAs (Section 5.1). This transformation introduced six additional actors connected with 12 additional connections, which is a potential CPU bottleneck and shared memory contention on RAM which is reflected in a 7% longer program runtime, *i.e.* the additional communication cost appears to outweigh multi-threaded parallelism in this case.

Trace based profiling showed intensive scheduling of actors that have only a small number of computationally inexpensive actions. For example, the workload of the *kArray_evaluation* actor was profiled at using 12.2% of overall CPU runtime, despite there being only two actions in the actor, one of which computed the Epanechnikov kernel with no token passing and the other action repeatedly transmitted the kernel values to colour model actors using state transitions to itself. The latter was initially implemented as a transmission action looping over the kernel size. The **built-in constructs** transformation was applied to replace the original action with CAL's `repeat` construct. The profiler reports a workload reduction of 82% for the *kArray_evaluation* actor. It reduced reduced program runtime by 21%.

A FIFO size of 32768 was needed to stream two consecutive YUV frames through the naive mean shift dataflow graph. Attempting to pass more frames through the graph deadlocked execution, suggesting that the naive version does not fully support the streaming model which is required for continuous tracking. For example, in order to pass 42 frames through the graph required a FIFO size of 1048576 and to pass 130 frames through required a FIFO size of 16777216. The Orcc profiler identified a starvation of tokens in the FIFO between the `R`, `G` and `B` ports and the *update_model* actor, because the tokens were not being consumed at the same rate by an actor that only computed $q_u(y_0)$ for the first frame and *update_model* that computed $p_u(y)$ for all consecutive frames.

The **FSM minimisation** transformation removed the FIFO size bottleneck. Unlike the FSM minimisation for FPGAs which removed the non-streaming Epanech-

Functionality	Transformation	Runtime	FPS
Naive version		2.97s	43.8
Updating the model	FSM simplification	2.06s	63.1
Displacement	Task parallelism	3.17s	41.0
Compute tracking window	Loop elimination	2.96s	43.9
RGB to YUV	Actor fusion	2.67s	48.7
k-array evaluation	Language use	2.34s	55.6
Combined		**1.70s**	**76.5**

Table 5: Transformation effects on CPU results

nikov kernel computation, the CPU transformation merged the kernel computation the $q_u(y_0)$ and $p_u(y)$ computations into *update_model*, using a modified FSM that visits the action that computes $q_u(y_0)$ only once. In contrast to FPGAs, once the computation has been executed the corresponding machine instructions can be removed from memory. The optimisation reduces the FIFO size to 32768 to process any number of frames, and the algorithm now supports the streaming model.

The naive version had an actor (not shown in Fig. 4) that received X and Y values for the tracked subject, and was responsible for drawing a rectangular window around the tracked target. This actor stored the R, G and B values for an entire frame before using the tracking location to determine which pixels must be highlighted. The **loop elimination** transformation was used to track location to highlight pixels *on-the-fly* if their position is within the tracking window criteria.

The overall effects on program runtime is shown in Table 5. It shows the runtime of 2.97s for the naive mean shift version, the effect on runtime for each transformation separately, and then for all optimisations combined. CPU runtimes are reduced in four out of five cases. Combining all transformations gives a runtime of 1.70s, a 43% runtime improvement over the naive version.

6 Conclusions

This paper showcases the dataflow program abstraction as an amenable model for highly effective FPGA optimisation. It defines 8 dataflow transformations, 2 FPGA specific computational transformations and various approaches to profiling dataflow graphs. These are combined to optimise a mean shift person tracking algorithm expressed in the CAL dataflow language. The trace-based profiles identify opportunities for transformations targeting CPUs. For a video of 130 frames, applying all CPU targeting transformations shortens runtime from 2.97s to 1.7s. The FPGA clock frequency for each actor identifies transformation opportunities. We increase the FPGA clock frequency with five transformations, and with all transformations applied increase overall clock frequency from 66.5MHz to 110MHz. As future work we will investigate all factors that affect frames-per-second throughput performance. They include: algorithmic parameters *e.g.* tracking window size, FPGA performance *e.g.* clock frequency and computation complexity as a function on inputs *e.g.* tracked object velocity and the background scene. We plan on embedding the dataflow transformations as compiler optimisations guided by FPGA simulation and CPU traced-based profiling, to assess the generality of our approach across different application domains.

Acknowledgements 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). The authors thank Blair Archibald for helpful feedback.

References

1. Adl-Tabatabai, A., Cierniak, M., Lueh, G., Parikh, V.M., Stichnoth, J.M.: Fast, Effective Code Generation in a Just-In-Time Java Compiler. In: Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998, pp. 280–290. ACM (1998)
2. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler Transformations for High-Performance Computing. ACM Comput. Surv. **26**(4), 345–420 (1994)
3. Bezati, E., Mattavelli, M., Janneck, J.W.: High-level Synthesis of Dataflow Programs for Signal Processing Systems. In: International Symposium on Image and Signal Processing and Analysis (ISPA), Trieste, Italy September 4-6, 2013, pp. 750–754. IEEE (2013)
4. Bhowmik, D., Wallace, A.M., Stewart, R., Qian, X., Michaelson, G.J.: Profile Driven Dataflow Optimisation of Mean Shift Visual Tracking. In: 2014 IEEE Global Conference on Signal and Information Processing, GlobalSIP 2014, Atlanta, GA, USA, December 3-5, 2014, pp. 1–5 (2014)
5. Bonenfant, A., Chen, Z., Hammond, K., Michaelson, G., Wallace, A., Wallace, I.: Towards Resource-Certified Software: A Formal Cost Model for Time and Its Application to an Image-Processing Example. In: Proc. ACM Symposium on Applied Computing, pp. 1307–1314 (2007)
6. Brown, C., Danelutto, M., Hammond, K., Kilpatrick, P., Elliott, A.: Cost-Directed Refactoring for Parallel Erlang Programs. International Journal of Parallel Programming **42**(4), 564–582 (2014)
7. Brown, C., Loidl, H., Hammond, K.: ParaForming: Forming Parallel Haskell Programs Using Novel Refactoring Techniques. In: R. Peña, R.L. Page (eds.) Trends in Functional Programming, 12th International Symposium, TFP 2011, Madrid, Spain, May 16-18, 2011, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 7193, pp. 82–97. Springer (2011)
8. Brunet, S.C., Alberti, C., Mattavelli, M., Janneck, J.W.: Turnus: A Unified Dataflow Design Space Exploration Framework for Heterogeneous Parallel Systems. In: 2013 Conference on Design and Architectures for Signal and Image Processing, Cagliari, Italy, October 8-10, 2013, pp. 47–54. IEEE (2013)
9. Chang, P.P., Mahlke, S.A., Chen, W.Y., mei W. Hwu, W.: Profile-Guided Automatic Inline Expansion for C Programs. Software, Practice & Experience **22**(5), 349–369 (1992)
10. Comaniciu, D., Ramesh, V., Meer, P.: Kernel-Based Object Tracking. IEEE Transactions on Pattern Analysis and Machine Intelligence **25**(5), 564–577 (2003)
11. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
12. Eker, J., Janneck, J.W.: CAL Language Report Specification of the CAL Actor Language. Tech. Rep. UCB/ERL M03/48, EECS Department, University of California, Berkeley (2003). URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2003/4186.html`
13. Floating-Point Working Group, IEEE Computer Society: IEEE Standard for Binary Floating-Point Arithmetic (1985). Note: Standard 754–1985
14. Gordon, M.I., Thies, W., Karczmarek, M., Lin, J., Meli, A.S., Lamb, A.A., Leger, C., Wong, J., Hoffmann, H., Maze, D., Amarasinghe, S.P.: A Stream Compiler for Communication-Exposed Architectures. In: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), San Jose, California, USA, October 5-9, 2002., pp. 291–303 (2002)
15. Govindu, G., Zhuo, L., Choi, S., Prasanna, V.K.: Analysis of High-Performance Floating-Point Arithmetic on FPGAs. In: 18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA. IEEE Computer Society (2004)
16. Grov, G., Michaelson, G.: Hume Box Calculus: Robust System Development Through Software Transformation. Higher-Order and Symbolic Computation **23**(2), 191–226 (2010)

17. Intel: Intel VTune Performance Analyzer. URL https://software.intel.com/en-us/intel-vtune-amplifier-xe
18. Janneck, J.W., Mattavelli, M., Raulet, M., Wipliez, M.: Reconfigurable Video Coding: A Stream Programming Approach to the Specification of New Video Coding Standards. In: W. Feng, K. Mayer-Patel (eds.) Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems, MMSys 2010, Phoenix, Arizona, USA, February 22-23, 2010, pp. 223–234. ACM (2010)
19. Janneck, J.W., Miller, I.D., Parlour, D.B., Roquier, G., Wipliez, M., Raulet, M.: Synthesizing Hardware from Dataflow Programs - An MPEG-4 Simple Profile Decoder Case Study. Signal Processing Systems **63**(2), 241–249 (2011)
20. Kuck, D.J.: A Survey of Parallel Machine Organization and Programming. ACM Comput. Surv. **9**(1), 29–59 (1977)
21. Marathe, J., Mueller, F.: Hardware Profile-Guided Automatic Page Placement for cc-NUMA Systems. In: J. Torrellas, S. Chatterjee (eds.) Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006, pp. 90–99. ACM (2006)
22. of Reading, U.: Performance Evaluation of Tracking and Surveillance (PETS 2009) Dataset (2009). URL http://www.cvg.rdg.ac.uk/PETS2009/
23. Scholz, S.: Single Assignment C: Efficient Support for High-Level Array Operations in a Functional Setting. J. Funct. Program. **13**(6), 1005–1059 (2003)
24. Stewart, R., Bhowmik, D., Michaelson, G., Wallace, A.: Open access dataset for "Profile Guided Dataflow Transformation for FPGAs and CPUs". http://dx.doi.org/10.17861/7925c541-42d9-4ded-9a01-5ac652d51353 (2015)
25. Trinder, P.W., Hammond, K., Loidl, H.W., Peyton Jones, S.L.: Algorithm + Strategy = Parallelism. J. Funct. Program. **8**(1), 23–60 (1998)
26. Underwood, K.D.: FPGAs vs. CPUs: Trends in Peak Floating-Point Performance. In: R. Tessier, H. Schmit (eds.) Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays, FPGA 2004, Monterey, California, USA, February 22-24, 2004, pp. 171–180. ACM (2004)
27. Xilinx: ISE Design Suite. URL http://www.xilinx.com/products/design-tools/ise-design-suite
28. Yviquel, H., Lorence, A., Jerbi, K., Cocherel, G., Sanchez, A., Raulet, M.: Orcc: Multimedia Development Made Easy. In: ACM Multimedia Conference, MM '13, Barcelona, Spain, October 21-25, 2013, pp. 863–866. ACM (2013)