ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

StreamDrive: A Dynamic Dataflow Framework for Clustered Embedded Architectures

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
arthur stoutchinin, luca benini (2019). StreamDrive: A Dynamic Dataflow Framework for Clustered Embedded Architectures. JOURNAL OF SIGNAL PROCESSING SYSTEMS FOR SIGNAL, IMAGE, AND VIDEO TECHNOLOGY, 91(3-4), 275-301 [10.1007/s11265-018-1351-1].

Availability:
This version is available at: https://hdl.handle.net/11585/702075 since: 2019-10-11

Published:
DOI: http://doi.org/10.1007/s11265-018-1351-1

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

12 October 2024

This is a post-peer-review, pre-copyedit version of an article published in the *Journal of Signal Processing Systems*. The final authenticated version is available online at: https://doi.org/10.1007/s11265-018-1351-1

This version is subjected to Springer Nature terms for reuse that can be found at: https://www.springer.com/gp/open-access/authors-rights/aam-terms-v1

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

StreamDrive: A Dynamic Dataflow Framework For Clustered Embedded Architectures

Arthur Stoutchinin · Luca Benini

the date of receipt and acceptance should be inserted later

Abstract In this paper, we present StreamDrive, a dynamic dataflow framework for programming clustered embedded multicore architectures. StreamDrive simplifies development of dynamic dataflow applications starting from sequential reference C code and allows seamless handling of heterogeneous and application-specific processing elements by applications. We address issues of efficient implementation of the dynamic dataflow runtime system in the context of constrained embedded environments, which have not been sufficiently addressed by previous research. We conducted a detailed performance evaluation of the StreamDrive implementation on our *Application Specific MultiProcessor (ASMP)* cluster using the *Oriented FAST and Rotated BRIEF (ORB)* algorithm typical of image processing domain. We have used the proposed incremental development flow for the transformation of the ORB original reference C code into an optimized dynamic dataflow implementation. Our implementation has less than 10% parallelization overhead, near-linear speedup when the number of processors increases from 1 to 8, and achieves the performance of 15 VGA frames per second with a small cluster configuration of 4 processing elements and 64KB of shared memory, and of 30 VGA frames per second with 8 processors and 128KB of shared memory.

Keywords embedded, multicore, shared memory, dataflow, kahn process, heterogeneous, accelerator

Arthur Stoutchinin
ST Microelectronics,
Grenoble France
Tel.: +33-4-76586276
ORCiD: 0000-0002-7650-9570
E-mail: arthur.stoutchinin@st.com

Luca Benini
Electrical, Electronic, and Information Engineering Department,
University of Bologna, Italy,
and
Integrated Systems Laboratory,
Swiss Federal Institute of Technology (ETH), Zurich
ORCiD: 0000-0001-8068-3806
E-mail: luca.benini@unibo.it

1 Introduction

Advanced embedded computing platforms are often designed as *clustered* multi-cores [33, 43, 36, 10]. In a clustered architecture, processing elements are grouped in tightly-coupled clusters sharing a finite amount of resources such as local memory, a DMA, external access ports, etc. The main disadvantage of such platforms is that software engineers must explicitly deal with parallelism, with heterogeneous and application-specific computing elements, with limited in-cluster memory constraints, and with data transfers across the memory hierarchy. In this paper, we address the programming issues with clustered multi-core platforms.

The dataflow computing model aims at addressing the aforementioned programming challenges for embedded applications that exhibit streaming behavior, eg. image and video processing, multimedia, networking, etc. However, adoption of the dataflow programming model by industry has been hindered by two important issues: (i) the necessity to drastically modify the existing sequential software and software development flow, and (ii) an unappealing trade-off between model expressiveness and efficient implementation. Indeed, restricted dataflow models, such as the Synchronous Dataflow Model (SDF) [30], the Cyclo Static Dataflow Model [5], or the Parameterized Synchronous Dataflow Model [3] are amenable to analysis,

automation techniques, and efficient implementation, but are too constrained in expressiveness to meet the needs of many real-time industrial applications. On the other hand, dynamic dataflow models, such as Kahn Process Networks (KPN) [27], Boolean Dataflow [26] and Dynamic Dataflow (DDN) [6] are difficult to develop and often do not result in an efficient implementation [19, 50, 34].

In a preliminary publication [55], we presented the *StreamDrive* framework that supports parallelization of streaming applications and aims at reducing the effort required in doing this. StreamDrive supports two execution modes: *preemptive* for KPN execution, and *cooperative* for DDN execution. Based on the simultaneous support for these two execution modes, we propose an incremental transformation flow starting from a sequential reference application and moving towards an optimized dynamic dataflow implementation. Supporting the two execution modes simultaneously is essential for providing such incremental transformation flow because the initial transformation of a sequential algorithm into a KPN often requires minimal modification of the original code. Moreover, the process of transforming the KPN into a DDN by adding dataflow firing rules is relatively straightforward. Another benefit of this methodology is that application-specific hardware blocks, acting as KPN processes, can be seamlessly integrated together with software DDN actors at the application level.

The StreamDrive application programming interface (API) is built on top of the C language and relies on familiar standard development tools, and the resulting parallel code is not radically different from the initial reference software. Although several dataflow APIs have been proposed in the past (see section 2), none simultaneously combines the support for the KPN and DDN execution modes, while relying on standard C development tools without introducing language restrictions, and remains sufficiently lightweight for targeting constrained embedded platforms. The StreamDrive communication API allows actors to share the dataflow buffers and efficiently supports data-parallel actors. Finally, the StreamDrive provides a lightweight runtime execution environment where particular attention is paid to minimize the overhead for the run-time support in terms of both, execution cycles and memory footprint requirements. Specific challenges addressed in our work include a low-overhead scheduler, dealing with small memories and the memory hierarchy that needs to be explicitly managed by software. Generally, these issues are insufficiently addressed in existing run-time environments and embedded real-time operating systems.

In [55], we also presented a detailed performance analysis of the dynamic dataflow execution model using a real-life application in a context of a small-scale embedded platform. We have implemented StreamDrive on the embedded *Application Specific MultiProcessor (ASMP)* platform from ST Microelectronics [52]. We present the results of the evaluation carried out over the Oriented FAST and Rotated BRIEF (ORB) application use cases, which are commonly used in mobile and automotive camera image processing pipelines [49]. Our evaluation showed that the StreamDrive-based ORB implementation achieves real-time performance, low parallelization overhead, small memory footprint, scales near linearly from 1 to 8 processing cores, and maintains performance even with long external memory latency and limited available bandwidth. Compared to other reported publications, our runtime implementation has lower overhead, and our speedup is closer to linear due to efficient combination of two types of parallelism: functional and data parallelism.

This article extends the preliminary version [55] in the following ways. First, we explain in details the StreamDrive communication protocol. Second, we provide a detailed description of the incremental transformation flow starting from the reference code and ending with optimized dataflow implementation, using the ORB as a running example. We explain the details of all the key aspects of the proposed methodology. Finally, we quantify and analyze the performance improvement of a dataflow execution with respect to the KPN execution, and demonstrate that the KPN scheduling overhead is relatively important in a typical embedded multiprocessor context.

The article is organized as follows: related work is explained in section 2; in section 3 we give the overview of the ASMP platform, explain its shared memory architecture and its support for the application-specific hardware; in section 4 we present the StreamDrive API and its implementation choices, and we illustrate the StreamDrive incremental transformation flow with an example; finally, we discuss our evaluation results in section 5.

2 Related Work

The dataflow execution model is a popular research topic in the embedded domain because it is a good match for many applications and hardware platforms. Several approaches of dataflow programming have been proposed with the objective to balance conflicting concerns of expressiveness, analyzability, and implementability [54]. Table 1 summarizes selected representative related dataflow publications.

Framework	Model	Target	Programming
Ptholemy II	Most existing dataflow models	Simulation and design environment	Specialized Language
LWDF	CFDF	Modeling Framework	C API
StreamIT	SDF	Software development	StreamIT Language
Sesame	KPN	Simulation and design environment	C API
Kaapi	Dynamic dataflow	HPC	C API
TideFlow	Dynamic dataflow	HPC	C API
OpenStream	Control driven dataflow	HPC	OpenMP Extension
YAPI	KPN	Workstation	C API
Nornir	KPN	Workstation	C API
PREESM	PiSDF	Simulation and code generation embedded	Graphical GUI + C
Kalray MPPA	CSDF	Embedded	ΣC Language
CAL	SDF,CSDF, Dynamic dataflow	Embedded	Specialized Language
Shim	Restricted KPN	Embedded	C Extension
DOL	KPN	Embedded	C Restricted

Table 1 Selected related work summary

A large number of frameworks propose specialized languages and tools for developing dataflow applications (the reader is referred to [4] for a comprehensive survey). The premise of these frameworks is that an application can be specified at a high abstraction level and automatically transformed into an efficient implementation. However, in practice there has been a noticeable gap between a high-level description and the efficient implementation that the automated tools failed to close. As a result, often restricted dataflow models are used such as Synchronous Dataflow (SDF) [30], Cyclo Static Dataflow [5], Parameterized SDF [3], Heterochronous Dataflow [20], etc., while achieving the efficiency with more expressive Kahn Process Networks [27], or Dynamic Dataflow [26] remains difficult [19, 50, 34]. Another major inconvenience of these frameworks is that they requires significant changes to reference software and to the existing software development flow - reference applications are typically specified as sequential programs using imperative programming languages such as C/C++ or Matlab. The disruptive changes in software development flow and being able to only deliver efficient implementation for a restricted set of the dataflow models hinders adoption of these technologies by industry.

An alternative is to integrate coarse grain dynamic dataflow programming structures into familiar languages, using a lightweight API with an associated runtime environment. Several such KPN and dynamic dataflow APIs have been proposed in the literature. Many of them target large computing systems and often rely on off-the-shelf OSes. Kaapi [17], Sesame [40], Shim [12] are based on POSIX threads. The QUARK (QUeing And Runtime for Kernels) [61], TIDeFlow [38], and OpenStream [44], have been developed in the context of the High-Performance Computing (HPC) applications. YAPI [28] and Nornir [60] support the KPN execution model on workstation computers. These runtime environments come with heavy performance and memory footprint overheads. This is an acceptable choice for running applications in big-size computers. In the embedded domain we need a lightweight approach: the small memory and the high performance requirements preclude using the full OS, a kernel-level scheduler, and dynamic data structures.

One example of a minimalist dataflow API similar to StreamDrive is the *lightweight dataflow* (LWDF) [53]. The LWDF implements the *core functional dataflow* (CFDF) model [41]. In the CFDF, an actor has a set of valid modes in which it can execute. The actor specification is divided into separate *enable* and *invoke* functions. The *enable* is designed to be used as a "hook" for the dynamic scheduler to rapidly query actors at runtime, and check whether or not they are executable. The *invoke* function implements actor functionality and can generally change the mode of the actor for the next invocation. This is similar to the StreamDrive, where actors proceed deterministically to some "next mode" of execution while changing their firing rules. Plishker et al. [42] have presented an analysis method that can exploit the core functional dataflow to improve the scheduler. The LWDF focus is on providing a framework for modeling and exploring the scheduler strategies, and it does not address the implementation efficiency issues of dynamic dataflow applications.

Several publications addressed supporting static and quasi-static dataflow execution model on embedded platforms. For example, the Parallel and Real-time Embedded Executives Scheduling Method (PREESM) is a framework offering rapid prototyping and automatic

code generation for heterogeneous multi-core embedded systems. PREESM targets the TI's Keystone DSP architecture and supports the PiSDF model [39]. Another example is the Kalray MPPA system programmed using the specialized ΣC language [21] and implementing the Cyclo Static Dataflow [5].

In order to overcome the limitations of the static dataflow, the Scenario-Aware Dataflow (SADF) [56] views applications as collections of different SDF graphs. SADF is able to perform some worst-case and stochastic analyses, and to provide implementation with limited run-time overhead, while relaxing some of the limitations of the SDF. However, this approach is still limited to a range of applications that follow a sequence of fairly static scenarios. From a syntactic perspective, the SADF model resembles the Heterochronous Dataflow [20], and therefore requires complete re-write of application reference code and usage of specialized development tools.

The above approaches are different from the StreamDrive in that they restrict the computation model to a subclass of dataflow process networks and rely on static scheduling for achieving efficiency.

In order to support dynamic dataflow execution, several research leverage on CAL programming language [14, 15], and its ISO-standardized subset, RVC-CAL [31, 32]. The RVC-CAL provides a dataflow framework with high level of abstraction and modularity as a basis for platform independent description of dataflow programs for execution on multicore platforms.

In [18, 58], the *Actor Machine* is used to generate an application-specific runtime dataflow scheduler from CAL targeting the Epiphany architecture [37]. The generated scheduler is less efficient compared to the StreamDrive because the actor machine does not memorize actors blocking conditions and therefore reevaluates these blocking conditions multiple times, while explicit enumeration of actor states leaves large memory footprint. The communication library is tailored to Epiphany's distributed shared memory and does not support sharing of communication buffers. Finally, work in [18] does not support the dynamic mapping of actors on processing elements.

Yviquel et al [62, 63] use the RVC-CAL infrastructure for developing a dynamic dataflow framework targeting a shared memory multi-core platform. The framework pays particular attention to the efficient implementation of dataflow communication functions. In particular, the shared memory is used to implement the zero-copy communication channels. The essential difference from StreamDrive lies in how the race conditions are avoided: while StreamDrive defines a communication protocol that ensures the race-free execution, the work in [63] guarantees the atomicity of the sequence *read-input/execute/write-output* by postponing the update of dataflow channels state until the actor has finished execution. Apart from potential performance overhead, this would be incompatible with the KPN mode of execution. This framework also supports broadcasting single data to several target actors (if all concerned FIFO channels are mapped to the same shared memory bank) that reminds the StreamDrive *broadcast* operation. However, there is no equivalent to the *collect* operation and no support for the data parallelism. Finally, the work in [63] relies on static scheduling of actors to platform processing elements.

The RVC-CAL offers a standardized framework for developing dynamic dataflow applications. However, there are very few applications available in CAL, mostly video codecs, while the majority of new applications continue to be developed in standard languages. Thus, the most important difficulty of using CAL is that it requires a complete re-write of the reference applications. CAL tools are also less mature that standard development tools such as the gcc compiler, etc.

Shim [12, 13, 59] and the Distributed Operation Layer (DOL) [23, 22] addressed implementation issues of the KPN execution model in the embedded context. Shim implements a KPN restricted to support only synchronous (*rendezvous*) communication. This choice eases scheduling, and programs are, by definition, always executable in finite space because synchronous communication does not need buffering. Shim language is based on C (but is not a C subset) augmented with few constructs for concurrency, communication, and exceptions. Compared to the StreamDrive, Shim imposes many syntactic restrictions on the input language which makes porting existing reference applications more difficult. Furthermore, Shim does not address the runtime implementation issues, instead it relies on costly standard runtime support such as Pthreads library.

DOL implements the KPN execution model using cooperative protothreads [11]. While cooperative scheduling eliminates context-switching overhead and simplifies the runtime stack handling, the protothreads impose a number of important language restrictions leading to additional performance overhead and to artificial changes to the sequential reference code. Using the protothreads precludes implementation of real-time systems that may require preemptive scheduling. DOL assignment of actors to processors is static reducing load balancing ability. Unlike StreamDrive, DOL only implements the KPN execution model which leads to additional performance penalty compared to the dataflow execution because there are no firing rules: the sched-

uler keeps trying to execute actors even when their blocking condition persists.

Like the work in [63] cited earlier, DOL takes advantage of shared memory in order to implement copy-free dataflow communication channels. In particular, DOL's *windowed FIFOs* [25] give actors direct access to data buffers to avoid copying. However, the windowed FIFO buffers cannot be shared between multiple actors, and do not support the broadcast and collect operations.

Gangwal et al [16] proposed the *query/claim/release* protocol, similar to the StreamDrive communication protocol. One important difference between the StreamDrive implementation and the *query/claim/release* protocol is that the StreamDrive synchronization counters count actual data bytes present in the communication buffers instead of tokens. This enables communicating actors to refer to different token sizes while still benefiting from an efficient synchronization support. Most importantly, the *query/claim/release* protocol do not allow buffers to be shared between multiple actors.

StreamDrive actors' ability to share data buffers is essential for reducing memory requirements and the communication overhead, but also for supporting the *data parallelism*. Zaki et al [64] developed *Partial Expansion Graphs (PEGs)* to help exploit the data parallelism in addition to functional/pipeline parallelism for SDF graphs. Similar to the StreamDrive, several instances of dataflow actors may be instantiated depending on relative load of the actor in terms of the execution time. Compared to StreamDrive, the PEG methodology has a number of important restrictions: (1) it is applicable to SDF graphs, (2) the data-parallel actors cannot have internal state, and (3) the data-parallel actor instances must execute in different processing cores. The work in [64] also develops a dynamic scheduling heuristics and shows that this scheduler performs better than the static one when actor execution loads are variable. However, their runtime implementation incurs higher performance overhead compared to StreamDrive: the synchronization and scheduling are ensured by a special *buffer manager* actor built on top of the underlying RTOS services, while StreamDrives' *broadcast* and *collect* synchronization is built into the scheduler itself. Finally, proposed *Particle Swarm Optimization (PSO)* approach for calculating the degree of the data parallelism for dataflow actors is general enough and can be applied on top of the StreamDrive as well.

Dynamic dataflow scheduling in the context of multicore systems has been studied by Michalska at al [34, 35]. The techniques proposed in these publications can be easily integrated with the StreamDrive distributed scheduler.

The above mentioned published research has not fully addressed the question of efficient execution of dynamic dataflow models, such as DDN and KPN, in small-scale clustered embedded architectures. Furthermore, they do not address the integration of specialized hardware blocks with programmable components in a single dataflow representation.

The integration of the application-specific hardware blocks has been previously addressed in the context of the high level synthesis (HLS) design flow. Several authors used RVC-CAL language as a single starting point for description of SW and HW components in a heterogeneous platform [47, 1, 2, 51]. Serot et Al [57] developed CAPH programming language for describing and implementing stream-processing applications on reconfigurable hardware, such as FPGAs. CAPH is based upon the dynamic dataflow model, supports an automated compilation producing VHDL code, and structurally reminds CAL.

These methodologies propose the software/hardware co-design flow that automates analysis, synthesis, optimization, and design space exploration for a given dataflow application. While targeting an application-specific solution, they pursue three design objectives: (1) higher degree of program analyzability and fast design cycle, (2) platform independent description that can be utilized for any implementation platform, and (3) rapid exploration of design alternatives. However, as explained earlier, CAL (and similarly, CAPH) is a specialized language with limited code base. The mainstream programming languages for heterogeneous computing are C and C-like languages. Porting a C application to CAL requires considerable effort and investment, comparable to developing a dynamic dataflow application from scratch.

On the other hand, we are interested in developing specialized hardware blocks for a particular target platform that can be reused accross a given application domain, eg. the convolution for image processing. This requires analyzing multiple applications and a "generic" hardware block development approach. The hardware-software partitioning and architecture exploration in our work has been conducted using higher-level simulation, with hardware block models derived from reference C functions with minimal modifications of original code. The specialized hardware blocks in our work have been designed using traditional RTL development flow. If desired, in order to speed-up the design cycle and to make the design technology-independent, the HLS tools such as CatapultC can also be used for these hardware blocks implementation.

3 Target Platform

StreamDrive targets small-scale clustered embedded architectures, where heterogeneous processing elements (PEs) are grouped into small clusters sharing a finite amount of resources including local memory. In this paper, we focus on programming a single cluster of such PEs. We are targeting image processing domain with real-time requirements of processing multiple image frames per second, we adopt the strategy of distributing computations of different image frames on different clusters. Thus, in a multicluster configuration, each cluster executes the same dataflow application applied to different image frame.

Fig. 1 shows the block diagram of our target architecture, the STMicroelectronics' ASMP cluster. It is composed by a number of programmable cores, *specialized hardware blocks (HWPEs)*, and a DMA, all connected together to a shared Tightly-Coupled Data Memory (TCDM). The HWPEs are essential for achieving the required performance while keeping the cost and the power consumption low. In order to even further optimize power-efficiency of the system, the HWPEs can run each in their own different dedicated clock domain, thus allowing for the adjustment of their frequency in accordance with application requirements. Seamless integration of the hardware blocks is one of the important advantages of the StreamDrive framework.

The key element of the ASMP cluster is its *logarithmic* interconnect [45] that allows multiple concurrent accesses to the multi-bank TCDM memory. In order to minimize the number of stalls due to conflicting simultaneous accesses to the same bank, the banking factor (i.e. the ratio between the number of TCDM memory banks and the number of access ports), needs to be correctly dimensioned. Such shared memory organization, although it has a limited scalability, corresponds well to the small-scale cluster architecture that we target. Our experience, confirmed by other studies on similar architectures [8], shows that this type of interconnect can support up to 32 access ports, each with a throughput close to 32-bits/cycle with latency compatible with the RISC core internal pipeline, under the embedded IP target frequencies.

The connection between the HWPEs and the shared memory is ensured by the *hardware block interface* (I/F 0, .. I/F K-1 in the figure) that serves as a bridge for streaming hardware blocks. The programmable cores, the hardware block interface, and the DMA, all support the StreamDrive communication protocol based on shared memory - this creates a common infrastructure for the core-to-core, the core-to-hardware-block, or the hardware-block-to-hardware-block communication.

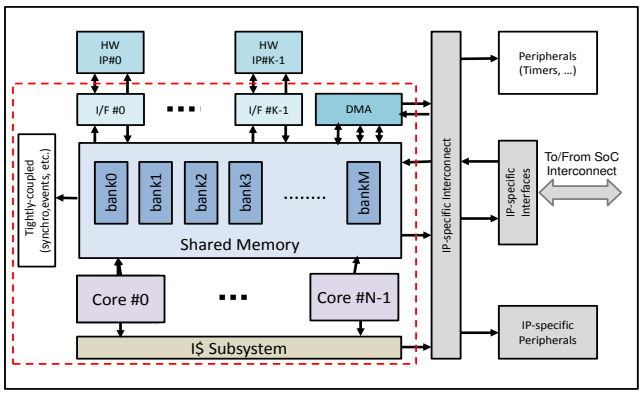

Fig. 1 The ASMP Cluster Block Diagram

The shared tightly-coupled memory organization is important for taking full advantage of the StreamDrive features.

The size of the TCDM has important impact on area-efficiency (GOPS/mm2) of the system: the larger the TCDM the lower area-efficiency. Generally, relatively small TCDM memory cannot fit the entire application working set. The StreamDrive cluster includes a DMA used for data movement between the TCDM and larger external off-chip memory. The DMA ensures additional function of synchronizing data transfers with the rest of the processing.

Finally, the ASMP cluster includes a small number of tightly-coupled peripherals aiming at accelerating synchronization, event handling, etc.

The current ASMP cluster implementation targets mobile image processing applications and includes 8 RISC processor cores running at relatively low frequency (500 MHz), with 32K of instruction cache each. The processor cores are extended with a small dedicated set of specialized instructions resulting in a 2-4X acceleration of important image processing functions [1] The cluster also includes two hardware blocks: the Gaussian filter and the Scaler interpolation block, - along with 256 KB of the TCDM memory. Overall, one ASMP cluster delivers up to 8Gops at 500 MHz not counting the hardware blocks, while the TCDM peak bandwidth reaches 32 GB per second. A predecessor of this architecture, featuring 4 clusters and 16 processors per cluster, with no hardware blocks, achieved power consumption of 2W in 28nm technology [33]. ASMP targets an even lower power and silicon area budget, position-

[1] This set includes relatively generic instructions, such as a `MAC4CLIP` which performs SIMD multiplication on bytes of two input operands, saturates the two 16-bit results, and accumulates them with the result operand; as well as instructions dedicated to specific image processing functions, such as a `XORSBCW`, used in Support Vector Machine (SVM), which calculate the Hamming distance between two vectors.

ing this platform well within the low-power, low-cost embedded profile.

Although the logarithmic interconnect technology constraints limit the scale of a tightly-coupled cluster to a couple of dozens of processing elements, multiple clusters can be put together allowing massive up-scaling in performance while maintaining the initial power- and area- efficiency. However, the multi-cluster aspects are out of this paper's scope.

As described in this section, our StreamDrive implementation leverages the tightly-coupled shared memory available in clustered platformes similar to ASMP. However, the StreamDrive framework can be retargeted to other architectures, for example, to distributed shared memory clusters. In this case, attention should be payed to efficiently implementing StreamDrive communication layer, in particular the *broadcast* and the *collect* connections (refer to next section). The StreamDrive scheduler is already distributed, however the global scheduling list will need to be implemented differently, perhaps using a work-stealing approach. As a bottom line, it is important to keep in mind that StreamDrive targets systems with small-scale clusters, not exceeding douzens of processing elements, as opposed to large-scale massively parallel systems.

4 StreamDrive Overview

In this section, we discuss the StreamDrive API and the runtime system, and illustrate the incremental transformation flow of a sequential code into a dynamic dataflow implementation.

In the dataflow model of computation, an application is represented as a graph of *actors* [2] connected by the *communication channels*, or *dataflow buffers*. The actors carry the actual computation while exchanging application-specific units of data, called *tokens*, over the communication channels. Tokens are *written* and *read* onto the communication channels in FIFO order. In StreamDrive, actors are connected to communication channels via *input and output ports*. Reading from an input port blocks the actor until all required tokens are available in the channel, and writing to an output port blocks the actor until enough empty room is available in the channel for the writing. For example, an actor performing an image filtering operation could be reading an input image line-by-line and writing the filtered image line-by-line. Then, the input and output communication tokens correspond to one line of the image.

[2] In this paper, we also use term actor for the KPN processes for the sake of coherence.

Two major dataflow models of computation supported by the StreamDrive are the Kahn Process Networks and the Dynamic Dataflow Networks.

4.1 The StreamDrive API

The StreamDrive API is based on the C programming language and provides methods for defining the dataflow actors, for constructing the dataflow graph, and for controlling the runtime scheduler.

Each StreamDrive actor defines its private variables and its input and output ports. Actor ports specify the size of tokens exchanged on the given port. The actor's private variables and its communication ports are accessible from inside the actor functions via the `THIS` pointer. StreamDrive actors also define four basic functions: `CONSTRUCTOR`, `DESTRUCTOR`, `INIT`, and `WORK`. The `CONSTRUCTOR` and the `DESTRUCTOR` perform all actions required at actor creation and release time, in particular the actor ports are created inside the actor constructor function. The `INIT` function configures the actor for execution by initializing actor's internal state. Finally, the `WORK` implements the actor functionality.

The StreamDrive uses a copy-free communication protocol, leveraging on the shared TCDM available in hardware. The API defines four communication functions: *reserve* and *push* for writing the data to an output channel, and *pop* and *release* for reading the data from an input channel. Before writing into an output channel, a source actor must acquire a pointer to an available empty buffer entry via the *reserve* call. The function is blocking if no room is available inside the given output buffer. When the data have been written to the buffer, the source actor signals the availability of new tokens via the *push* call. On the destination side, an actor must acquire a pointer to an input token via the *pop* call before reading the data. The *pop* function is blocking if there is not enough available tokens in the FIFO. The destination actor does not need to make a copy of the data but instead can use data directly from the shared communication buffer. When the destination actor no longer needs the data, it must signal the source actor that the buffer can be reused via the *release* function.

The StreamDrive API also provides methods for creating actors and their ports, and for connecting actors via communication buffers. The StreamDrive graph description can be parameterized in number of actor instances and their connections. The API supports the configuration of dataflow graphs between executions by disabling actors, actor connections and by changing actor parameters. It is important to note that the application graph does not need to change depending on

whether actors are implemented as software functions or as hardware blocks.

The StreamDrive model requires explicit management of the memory hierarchy, in particular of transferring data between the external and the local shared memory. Our experience is that streaming applications have regular memory access patterns and the advantages of explicit memory hierarchy management outweigh its inconveniences. The StreamDrive API implements a specific DMA support. A DMA function is similar to that of an actor: its implementation ensures that a synchronization is generated upon the DMA transfer completion in order to signal that a token is ready. In section 5 we show that using this mechanism, a very efficient hiding of external memory latency can be achieved.

Finally, the API provides a few functions for controlling the runtime scheduler. Of particular importance is the function for specifying the dataflow firing rules. A firing rule is specified by requiring certain number of free slots (output ports) or ready tokens (input ports) to be available in a given communication port before the next firing of the actor can take place. The firing rules can change for each new actor firing, supporting fully dynamic dataflow model. In the absence of firing rules, an actor behaves as a KPN process, possibly blocking during execution.

4.2 The Incremental Parallelization Flow

One important objective of the StreamDrive is to support the incremental transformation of a sequential reference code into an optimized dataflow form. In order to facilitate such transformation, the process is divided into a number of conceptually simple steps, each consecutive step is an incremental improvement over the previous one:

1. Identification of the dataflow part of the sequential application.
2. Identification of the dataflow actors and building the initial Kahn Process Network, KPN.
3. Refinement of the initial KPN by reducing actors granularity.
4. Identification and implementation of data parallel actors.
5. Conversion of the Kahn Process Network into the Dataflow Network by introducing the dataflow *firing rules*.
6. Optimization of the performance vs memory footprint trade-off.

The initial transformation of a sequential reference code into KPN form is facilitated by the fact that

```
static unsigned int   n_levels = 8;   // RO
static size_t         n_features = 500;
...
int                 * n_features_per_level;
            // to be transformed
...
main (int argc, char **argv) {
  char * scene_obj = argv[1];
  char * scene_db = argv[2];
  Image_t img;
  Descr_t descr_db;
  Match_t match_db;
  Point_t * keypoints = (Point_t*)malloc(n_levels*
      sizeof(Point_t));
  orb_init(&img, scene_obj, &descr_db, scene_db, &
      match_db);
  orb_run(&img, keypoints, &descr_db);
  match (&descr_db, &match_db);
  ... show results ...
  orb_deinit(&img, keypoints, &descr_db, &match_db)
      ;
}
```

```
void orb_run (Image_t img, Point_t * keypoints,
    Descr_t * descriptors) {
  Image_t * img_pyramid = (Image_t *) malloc(
      n_levels*sizeof(Image_t));
  ... compute rescaled image pyramid from the img
      ...
  computeKeyPoints(img_pyramid, keypoints);
  for (level = 0; level < n_levels; ++level) {
    Point_t * keyp = &keypoints[level];
    computeOrientation(level, &img_pyramid[level],
        keyp);
    Descr_t * descr = &descriptors[level];
    Image_t * blur_img = (Image_t *) malloc(sizeof(
        Image_t));
    computeGaussianFilter(level, &img_pyramid[level
        ], blur_img, ...);
    computeDescriptors(level, blur_img, keyp, desc)
        ;
    free (blur_img);
  }
  free (img_pyramid);
}
```

Fig. 2 Extract from the reference ORB application

streaming applications are typically structured into a sequence of processing kernels that roughly correspond to parallel Kahn processes. Transforming a sequential kernel into a Kahn process often requires minimal modifications to the code, consisting mostly of inserting KPN communication statements at appropriate places.

The biggest effort goes into achieving good performance vs memory footprint trade-off beyond the initial *basic* level. In this respect, the StreamDrive is not different from other parallelization approaches - usually a good understanding of the model is required in order to achieve high performance levels.

Importantly, all transformation steps can be performed incrementally, one actor at a time, allowing at each stage to debug and verify functional correctness of the transformation. In order to gain a more precise idea of the StreamDrive incremental parallelization flow, we illustrate the process using a real-life example. Figures 2 and 3 refer to the code for the *Oriented*

```
void computeKeyPoints (Image_t * img_pyramid,
    Point_t * keypoints) {
  ...
  for (level = 0; level < n_levels; ++level) {
    Point_t * keyp = &keypoints[level];
    int cornerCount;
    Keyp_t * results = fast9_nonmax(level, &
        img_pyramid[level], ..., &cornerCount);
    keyp->data = (Keyp_t *)malloc(cornerCount *
        sizeof(Keyp_t));
    copy (keyp->data, results, cornerCount);
    computeHarrisResponse(level, img_pyramid[level
        ], keyp, ...);
    cullKeypoints(level, keyp, n_features_per_level[
        level], ...);
    free (results);
  }
}
```

```
Keyp_t * fast9_nonmax (int level, Image_t * img,
    ..., int * n_corners) {
  int nCorners;
  Keyp_t * corners = fast9_detect (img, ..., &
      nCorners);
  int * scores = fast9_score (corners, nCorners,
      ...);
  Keyp_t * nonmax = nonmax_suppress (corners,
      scores, nCorners, ..., n_corners);
  return nonmax;
}
```

```
Keyp_t * fast9_detect (Image_t img, ..., int *
    num_corners) {
  int xsize = img->width;
  int ysize = img->height;
  int rsize=512;
  *num_corners = 0;
  Keyp_t * corners = (Keyp_t*)malloc(sizeof(Keyp_t)
      *rsize);
  for (y = edge_threshold; y < ysize -
      edge_threshold; y++) {
    for (x = edge_threshold; x < xsize -
        edge_threshold; x++) {
      ... compute keypoint or not ...
      if (corner) {
        if (*num_corners == rsize) {
    rsize*=2;
    corners = (Keyp_t*)realloc(corners, sizeof(
        Keyp_t)*rsize);
  }
        corners[*num_corners] = *corner;
  *num_corners++;
      }
    }
  }
  return corners;
}
```

```
Keyp_t * nonmax_suppress (Keyp_t * corners, int *
    scores, int num_corners, ..., int * num_nonmax)
    {
  *num_nonmax=0;
  Keyp_t * nonmax = (Keyp_t*)malloc(num_corners *
      sizeof(Keyp_t));
  ... compute nonmax corners ...
  free (corners);
  return nonmax;
}
```

Fig. 3 Compute_Keypoints function from the ORB application

Fast and Rotated Brief (ORB) application. The ORB algorithm identifies a set of objects inside an image and matches their descriptors to the descriptors of objects in a trained database. The objects are identified by detecting *keypoints* of interest via the *FAST algorithm* [48]. The corner keypoints are selected via the *nonmax suppression* and then sorted using *Harris response* measure [24] to retain only the "best" keypoints. For these keypoints, the algorithm computes object *orientation*, and objects' BRIEF *descriptor* of the object associated with each keypoint. The descriptor computation requires the *Gaussian filtered* image. In order to be independent from the distance-to-object, processing is repeated over a series of images representing scaled down original image, the *pyramid*. ORB puts to evidence several important parallelization challenges: (1) ORB computation is irregular - some parts of the image may not have any keypoints, while others contain many; progressively reduced pyramid image sizes; (2) the *nonmax* and the *sort* computations are inherently serial; (3) the working set footprint is larger than can fit with the small L1 level memory, therefore our implementation extensively. uses DMA for transferring data to and from the external memory.

```
void orb_run (Image_t img, Point_t * keypoints,
    Descr_t * descriptors) {
  Image_t * img_pyramid = (Image_t *) malloc(
      n_levels*sizeof(Image_t));
  ... compute rescaled image pyramid from the img
      ...
  GraphBuild_t build_parm;
  GraphExec_t exec_parm;
  build_parm.img_pyramid = img_pyramid;
  build_parm.n_levels = n_levels;
  ...
  Build_Graph (build_parm);
  Exec_Graph (exec_parm);
  Term_Graph ();
  free (img_pyramid);
}
```

Fig. 4 The `orb_run` function modified to execute under the StreamDrive runtime.

As a preparation step, the ORB application has been transformed from the floating-point version to the fixed-point suitable for an embedded implementation. The shown extract has been slightly amended for the purpose of the illustrating important points.

4.2.1 Identification of the dataflow graph

The ORB `main` function (line 6 in the top listing) receives the names of the image to process and of the objects database as arguments. Inside the `main` function, the `orb_init` loads the input image from a file; loads the trained objects database initializing the `match_db` for matching image objects vs the database objects; and initializes some global parameters. The `orb_run` computes the keypoints and the object descriptors. The `match` function compares the `descr_db` vs the `match_db`

classifying the objects found inside the input image. Finally, the `orb_deinit` releases resources allocated during the processing.

The very first step for transforming this code is to identify the part of the code which will become a dataflow graph. We will focus on the `orb_run` function where the compute intensive processing is required. The `match` function, accounting for about half of the processing requirements of the application, is another good candidate but is more communication than compute bound. In our actual implementation, the `orb_run` and the `match` have been implemented as two separate dataflow graphs. In order to simplify the illustration, we do not include building the scaled image pyramid (lines 2-3 inside the `orb_run` function) as part of the graph. Thus, we assume that the pyramid has been built during a pre-processing step and the results have been stored in external memory [3]. We divide the ORB application into two parts: (1) the *main part* which takes care of the input and output, user interactions, allocating and freeing resources, etc., and (2) the *dataflow part* which corresponds to the compute intensive part of the application. This dataflow part is executed as a dataflow graph under the control of the StreamDrive dataflow scheduler.

Figure 4 shows the transformed `orb_run` function from the main part of the application, The ORB computation has been replaced with the StreamDrive API calls for building, executing and releasing the dataflow graph. The `Build_Graph` API call takes an application-specific structure as a parameter used to pass construction time arguments to the graph such as the pointer at the image pyramid, number of pyramid levels, image dimensions, etc. A dataflow graph, once built, can be executed multiple times, for example looping over several input images. In this case, the `Exec_Graph` parameter can be used to pass different execution parameters for each graph execution.

The Figure 5 shows the dataflow part of the application. The single `ORB` actor is defined via two files, the `orbActor.h` shown in the top of the figure, and the `orbActor.c`, second listing from the top. The `.h` file defines actor's private data structure and actor ports. This actor has neither input, no output ports. The `.c` file defines four functions: the actor *constructor* and *destructor*, called at actor construction and termination time, respectively; the *init* function called before the graph execution starts; and the *work* function which performs the actors' workload. Notice that for the initial `ORB` actor, the constructor, destructor and init functions remain empty, while the work function is a copy-paste of the code from sequential reference. The only slight change from the reference code is that arguments such as pointers to the `img_pyramid`, to the `keypoints`, etc. are read from the global `cfg` structure. This structure

```
typedef struct {
  int32_t dummy;
} orb_t;

STREAM_DECLARE_ACTOR_TYPE(ORB,orb_t);
#define ORB_ACTOR_PORT_COUNT          0
```

```
#define ACTOR_NAME   ORB

#include <stream.h>

STREAM_CONSTRUCTOR (void * arg) {}
STREAM_DESTRUCTOR (void) {}
STREAM_INIT () {}
STREAM_WORK () {
  uint32_t n_levels = cfg->n_levels;
  Image_t * img_pyramid = cfg->img_pyramid;
  Point_t * keypoints = cfg->keypoints;
  Descr_t * descriptors = cfg->descriptors;
  computeKeyPoints(img_pyramid, keypoints);
  for (level = 0; level < n_levels; ++level) {
    Point_t * keyp = &keypoints[level];
    computeOrientation(level, &img_pyramid[level],
        keyp);
    Descr_t * descr = &descriptors[level];
    Image_t * blur_img = (Image_t *) malloc(sizeof(
        Image_t));
    computeGaussianFilter(level, &img_pyramid[level
        ], blur_img, ...);
    computeDescriptors(level, blur_img, keyp, desc)
        ;
    free (blur_img);
  }
  return 0;
}
```

```
STREAM_DECLARE_ACTOR(ORB,ORB_ACTOR_PORT_COUNT,2048)
    ;

static ORB_t * orbActor;

GlobalParam_t cfg;

int32_t Build_Graph (GraphBuild_t * arg) {
  ... initialize the cfg from arg ...
  orbActor = STREAM_ACTOR_MAKE(ORB, "orb", NULL);
}

int32_t Exec_Graph (GraphExec_t * arg) {
  uint32_t timeout = arg->timeout;

  STREAM_ACTOR_ENABLE (orbActor);
  STREAM_ACTOR_SET_PRIORITY(orbActor, 0);

  STREAM_GRAPH_SET_TIMEOUT (timeout);
}

int32_t Graph_Term () {
  STREAM_ACTOR_TERM(orbActor);
}
```

Fig. 5 The ORB code wrapped into dataflow graph with single actor

[3] In actual implementation, we have implemented two variants of the ORB: (1) with a rescaler tightly-coupled HW block and where the pyramid construction is part of the dataflow graph, and (2) with the pyramid construction as a pre-processing step.

is setup during the dataflow graph construction from `Build_Graph` arguments (line 8 in the bottom listing).

The three dataflow graph function corresponding to the StreamDrive API, the `Build_Graph`, the `Exec_Graph`, and the `Term_Graph` are shown in the bottom listing of the Figure 5. The `STREAM_DECLARE_ACTOR` function declares an actor, with the last parameter specifying how much stack room this actor needs to have allocated. The `STREAM_ACTOR_MAKE` function from the `Build_Graph` allocates resources for the actor and calls its constructor. Similarly, the `STREAM_ACTOR_TERM` function from the `Term_Graph` calls actor destructor before releasing actor resources. The `Exec_Graph` function configures the dataflow graph by enabling actors - it is possible to only enable a subset of all actors for any particular execution. Each actor is given a scheduling priority, the default is 0. Finally, there are no graph connections in our initial single-actor dataflow graph.

Notice two important points about our transformed application: (1) apart from little "syntactic sugar", the code changes to the initial sequential version are minimal; (2) the dataflow part runs under the control of our StreamDrive scheduler and executed in the KPN mode (there are no firing rules yet). It is clear that the process of identification of the main part and the dataflow part of the application is application-specific. The global variables that are used in the dataflow part need to be identified and declared with the `cfg` structure.

4.2.2 Building the initial dataflow graph

As next transformation step, the initial dataflow graph is built: we need to (1) identify sections of code (kernels) which can be transformed into dataflow actors, and (2) introduce the communication channels connecting the actors. This step is facilitated by the fact that streaming applications are typically structured into a sequence of processing kernels that are a natural choice for parallel Kahn actors. For example, from the Figure 2, the `ComputeOrientation`, `ComputeGaussianFilter`, and `ComputeDescriptors` seem to be good candidates for dataflow actors. The `ComputeKeyPoints` shown in Figure 3 is itself composed of kernels, the `fast9_detect`, the `nonmax_suppress`, the `ComputeHarrisResponse`, and the `CullKeypoints`. These will be our initial choice of the dataflow actors.

The channel introduction requires identifying, for the *input channels*, of the data that are read by this actor but written outside of it, and for the *output channels*, the data that are written by the actor and read elsewhere. This remains a manual task, although tools, such as Sprint [7] may be considered in future work.

```
  typedef struct {                                          1
    uint32_t        _cFastThreshold;                        2
  } fast_t;                                                 3
  STREAM_DECLARE_ACTOR_TYPE(FAST,fast_t);                   4
                                                            5
  #define FAST_IN_ESIZE             (sizeof(Image_t))       6
  #define FAST_OUT_ESIZE            (sizeof(Keyp_t))        7
  #define FAST_PORT_IN              0                       8
  #define FAST_PORT_OUT             1                       9
  #define FAST_PORT_COUNT           2                       10
```

```
  STREAM_CONSTRUCTOR (void * arg) {                         1
    STREAM_ACTOR_MAKE_PORT_IN(FAST_PORT_IN, "in_p",         2
        FAST_IN_ESIZE);
    STREAM_ACTOR_MAKE_PORT_OUT(FAST_PORT_OUT, "out_p        3
        ", FAST_OUT_ESIZE);
  }                                                         4
  STREAM_DESTRUCTOR (...) {                                 5
    STREAM_ACTOR_TERM_PORT_IN(FAST_PORT_IN);                6
    STREAM_ACTOR_TERM_PORT_OUT(FAST_PORT_OUT);              7
  }                                                         8
  STREAM_INIT() {}                                          9
  STREAM_WORK() {                                           10
    int16_t fastThreshold = THIS->_cFastThreshold;          11
    uint8_t n_levels = cfg->n_levels;                       12
    Image_t * img_pyramid = cfg->img_pyramid;               13
    uint32_t level;                                         14
                                                            15
    for (level = 0; level < n_levels; ++level) {            16
      int cornerCount;                                      17
      fast9_detect(level, &img_pyramid[level], ..., &       18
          cornerCount);
    }                                                       19
}                                                           20
```

```
  void fast9_detect (Image_t * img, ..., int *              1
      num_corners) {
    int xsize = img->width;                                 2
    int ysize = img->height;                                3
    //int rsize=512;                                        4
    Keyp_t * header = (Keyp_t*)STREAM_OUT_RESERVE (         5
        FAST_PORT_OUT, 1);
    *num_corners = 0;                                       6
    //Keyp_t * corners = (Keyp_t*)malloc(sizeof(            7
        Keyp_t)*rsize);
    for (y = edge_threshold; y < ysize -                    8
        edge_threshold; y++) {
      for (x = edge_threshold; x < xsize -                  9
          edge_threshold; x++) {
        ... compute keypoint or not ...                     10
        if (corner) {                                       11
          Keyp_t * token = (Keyp_t*)                        12
              STREAM_OUT_RESERVE (FAST_PORT_OUT, 1);
          //if (*num_corners == rsize) {                    13
    //  rsize*=2;                                           14
    //  corners = (Keyp_t*)realloc(corners, sizeof(         15
        Keyp_t)*rsize);
    //}                                                     16
          //corners[*num_corners] = *corner;                17
          *token = *corner;                                 18
    *num_corners++;                                         19
        }                                                   20
      }                                                     21
    }                                                       22
    header->... = *num_corners;                             23
    STREAM_OUT_PUSH(FAST_PORT_OUT, *num_corners+1);         24
    return;                                                 25
  }                                                         26
```

Fig. 6 Initial FAST dataflow actor

Once again, due to the intrinsic structure of the streaming applications, the channel introduction turns often to be relatively straightforward. From the ORB example, the `fast9_detect` kernel takes one image from the

image pyramid in its input channel, and produces the array of `cornerCount` `FAST` keypoints with their `FAST` scores. The `nonmax_suppress` takes the keypoints generated by the `fast9_detect` as input and generates the set of corner keypoints by removing "uninteresting" keypoints from the set. The `ComputeHarrisResponse` reads these corners and computes the *Harris response* for each of them, which is a measure of "relevance" of each keypoint. The `ComputeHarrisResponse` output is the set of keypoints with their associated Harris response. The `CullKeypoints` performs the sorting of the keypoints with respect to their Harris response and reduces the keypoint set further by retaining at most the `n_features_per_level` best keypoints. From these remaining keypoints, the `ComputeOrientation` computes each keypoint orientation. The `ComputeOrientation` has two input channels, the keypoints generated by the `CullKeypoints` and the scaled input image from corresponding image pyramid level. The output of the `ComputeOrientation` is a set of keypoints with their associated orientation measure. The `ComputeDescriptors` takes two input channels as well, the output keypoints from the `ComputeOrientation` and the Gauss filtered input image. The `ComputeDescriptors` output is the final set of keypoints and their descriptors.

In order to introduce new actors, each processing kernel needs to be *wrapped* into the StreamDrive syntactic structure similar to the earlier `ORB` actor. Figure 6 shows as example the `FAST` actor corresponding to the `fast9_detect` kernel. Inside the actor `.h` file, the actor ports are declared, where the `XXX_ESIZE` defines port token size. The `FAST` input tokens are of type `Image_t` and output tokens are corners of type `Keyp_t`. Inside the `.c` file, the actor constructor and destructor functions create and destroy, respectively, the actor ports. The change to the original `fast9_detect` function is minimal and consists in inserting the StreamDrive communication primitives for writing the output data to the output port. At this point we do not use the input port yet, because we did not create an actor that can write the data to this port, therefore our `FAST` actor keeps reading the `img_pyramid` directly from external memory. The `STREAM_OUT_RESERVE` and the `STREAM_OUT_PUSH` implement the StreamDrive communication protocol. Thus, a *reserve* is called for every new corner and at the end all corners are *pushed* to the output channel. In order to communicate the number of corners to the downstream actor, the `FAST` *reserves* one *header* token at the beginning of the processing. When the number of corners is known at the end of the outermost loop, the *header* is pushed to the output channel

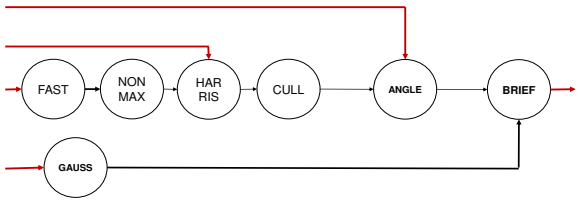

Fig. 7 Initial ORB dataflow graph: The actors correspond to the original kernels; `FAST`, `GAUSS`, `HARRIS`, and `ANGLE` read input image data directly from the external memory, `BRIEF` writes result descriptors directly to the external memory.

together with the corner tokens [4]. It is interesting to notice that using the streaming style communication allows us to get rid of inefficient `malloc`, `realloc`, and `free` calls.

The dataflow graph refinement is performed incrementally, one actor at a time, verifying the transformation correctness at each graph transformation. At the end of the process, the initial `ORB` actor is no longer needed and is removed from the graph. The resulting ORB graph is drawn in the Figure 7.

Figure 8 shows the corresponding `Build_Graph` function with seven actors. The `STREAM_BIND` function connects the output port of a source actor to the input port of the destination actor, while specifying the communication buffer depth and the memory level at which the buffer needs to be allocated. So far we have not addressed the memory size and actor granularity issues, therefore all buffers have been allocated in large external memory.

Notice that several ports remain unused in the current graph. These are ports that do not have actors to connect to. For example, the `FAST` actor input port which reads the input image from memory does not have a matching output port to connect to. Similarly, the `GAUSS` input port, the `HARRIS` and `ANGLE` ports that read input images from memory, as well as the `BRIEF` output port that writes final descriptors out to memory, do not have matching ports to connect to. All these ports correspond to input and output channels to the dataflow graph. This data initiate inside the external memory and need to be copied from this external memory to the L1 memory for processing.

We use the DMA engine to copy the external data to the L1 memory. For this, we use the StreamDrive DMA API. For the input channels, we introduce the new `SRC` actor which implements the DMA transfers. The `SRC` actor does not have input ports and has one output port to which data from the DMA transfer are sent. For transferring results from the `BRIEF` actor to the external memory, we add the StreamDrive DMA

[4] any field of the `Keyp_t` structure can be used to communicate the number of corners.

```
STREAM_DECLARE_ACTOR(GAUSS,GAUSS_ACTOR_PORT_COUNT,1024);
STREAM_DECLARE_ACTOR(FAST,FAST_ACTOR_PORT_COUNT,1024);
STREAM_DECLARE_ACTOR(NONMAX,NONMAX_ACTOR_PORT_COUNT,1024);
STREAM_DECLARE_ACTOR(HARRIS,HARRIS_ACTOR_PORT_COUNT,1024);
STREAM_DECLARE_ACTOR(CULL,CULL_ACTOR_PORT_COUNT,1024);
STREAM_DECLARE_ACTOR(ANGLE,ANGLE_ACTOR_PORT_COUNT,1024);
STREAM_DECLARE_ACTOR(BRIEF,BRIEF_ACTOR_PORT_COUNT,1024);

static GAUSS_t  * gaussActor;
static FAST_t   * fastActor;
static NONMAX_t * nonmaxActor;
static HARRIS_t * harrisActor;
static CULL_t   * cullActor;
static ANGLE_t  * angleActor;
static BRIEF_t  * briefActor;

GlobalParam_t cfg;

int32_t Build_Graph (GraphBuild_t * arg) {
  ... initialize the cfg from arg ...

  gaussActor = STREAM_ACTOR_MAKE(GAUSS, "gauss", NULL);
  fastActor = STREAM_ACTOR_MAKE(FAST, "fast", NULL);
  nonmaxActor = STREAM_ACTOR_MAKE(NONMAX, "nonmax", NULL);
  harrisActor = STREAM_ACTOR_MAKE(HARRIS, "harris", NULL);
  cullActor = STREAM_ACTOR_MAKE(CULL, "cull", NULL);
  angleActor = STREAM_ACTOR_MAKE(ANGLE, "angle", NULL);
  briefActor = STREAM_ACTOR_MAKE(BRIEF, "brief", NULL);

  STREAM_BIND (fastActor, FAST_PORT_OUT, nonmaxActor, NONMAX_PORT_IN, FAST_OUT_DEPTH, MEM_EXT);
  STREAM_BIND (nonmaxActor, NONMAX_PORT_OUT, harrisActor, HARRIS_PORT_IN, NONMAX_OUT_DEPTH, MEM_EXT);
  STREAM_BIND (harrisActor, HARRIS_PORT_OUT, cullActor, CULL_PORT_IN, HARRIS_OUT_DEPTH, MEM_EXT);
  STREAM_BIND (cullActor, CULL_PORT_OUT, angleActor, ANGLE_PORT_IN, CULL_OUT_DEPTH, MEM_EXT);
  STREAM_BIND (angleActor, ANGLE_PORT_OUT, briefActor, BRIEF_PORT_IN, ANGLE_OUT_DEPTH, MEM_EXT);
  STREAM_BIND (gaussActor, GAUSS_PORT_OUT, briefActor, BRIEF_PORT_BLUR, GAUSS_OUT_DEPTH, MEM_EXT);
}
```

Fig. 8 Listing of the `Build_Graph` function that constructs the initial ORB graph.

API calls inside the `BRIEF` actor. Figure 9 shows the ORB graph with the `SRC` actor broadcasting the input image to several ORB actors.

At the end of this step, the initial dataflow graph is built with several dataflow actors identified. Following important points facilitate this transformation step: (1) the actor *granularity of execution* of the original application has been preserved; (2) we have avoided dealing with limited memory constraints by allocating all communication buffers in sufficiently large external memory; (3) the actor execution order corresponds to that of the original application because we have preserved the sequential code *granularity* and dependencies.

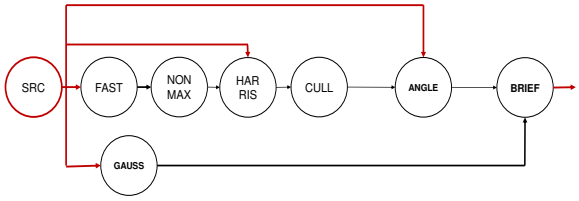

Fig. 9 ORB dataflow graph with the DMA actor added: the input image data are read via the DMA by the `SRC` actor and are broadcast to the `FAST`, `GAUSS`, `HARRIS`, and `ANGLE` actors. The `BRIEF` actor uses the StreamDrive DMA API for writing result descriptors to the external memory.

4.2.3 The dataflow graph refinement

The next transformation step is the dataflow graph refinement by reducing actor *granularity* so that dataflow communication buffers fit with limited L1 memory. The dataflow actor *granularity* refers to the amount of data that the actor needs for executing without being blocked, and is directly related to the size of actor input and output *tokens*.

# Actor	Port	Token size
FAST	IN	One image line
	OUT	One keypoint
NONMAX	IN	One keypoint
	OUT	One keypoint
HARRIS	IN	One keypoint
	REF	One image line
	OUT	One keypoint
CULL	IN	One keypoint
	OUT	One keypoint
ANGLE	IN	One keypoint
	REF	One image patch
	OUT	One keypoint
GAUSS	IN	One image line
	OUT	One image line
BRIEF	IN	One keypoint
	BLUR	One image patch
	OUT	One descriptor

Table 2 Granularity of actors in the ORB application

Table 2 shows refined token sizes for the ORB graph actors. Notice that we have chosen to fetch the `ANGLE` and the `BRIEF` image data one *patch* at a time: a patch is a small area around each keypoint. Because patches for different keypoints may overlap, we end up fetching same image pixels multiple times. However, the alternative of keeping the keypoints in raster scan order and

```
typedef struct {
  uint32_t   cFastThreshold;
  uint8_t *  line_p[3];
} fast_t;
STREAM_DECLARE_ACTOR_TYPE(FAST,fast_t);

// Ports
#define FAST_IN_ESIZE              (sizeof(Line_t))
#define FAST_OUT_ESIZE             (sizeof(Keyp_t))
#define FAST_PORT_IN               0
#define FAST_PORT_OUT              1
#define FAST_PORT_COUNT            2
```

```
STREAM_CONSTRUCTOR (void * arg) {
  STREAM_ACTOR_MAKE_PORT_IN(FAST_PORT_IN, "in_p",
      FAST_IN_ESIZE);
  STREAM_ACTOR_MAKE_PORT_OUT(FAST_PORT_OUT, "out_p
      ", FAST_OUT_ESIZE);
}
STREAM_DESTRUCTOR (...) {
  STREAM_ACTOR_TERM_PORT_IN(FAST_PORT_IN);
  STREAM_ACTOR_TERM_PORT_OUT(FAST_PORT_OUT);
}
STREAM_INIT() {}
STREAM_WORK() {
  int16_t fastThreshold = THIS->_cFastThreshold;
  uint8_t n_levels = cfg->n_levels;

  Keyp_t * header = (Keyp_t*)STREAM_OUT_RESERVE (
      FAST_PORT_OUT, 1);

  uint32_t level;

  for (level = 0; level < n_levels; ++level) {
    int xsize = cfg->img_width[level];
    int ysize = cfg->img_height[level];
    int cornerCount = 0;

    // Build FAST window
    for (i = 0; i < 3; i++) {
      THIS->line_p[i] = (Line_t*)STREAM_IN_POP(
          FAST_PORT_IN, 1);
    }

    for (y = edge_threshold; y < ysize -
        edge_threshold; y++) {
      int count;
      fast9_detect (level, THIS->line_p, ..., &
          count);
      cornerCount += count;
      // Rotate FAST window
      STREAM_IN_RELEASE (FAST_PORT_IN, 1);
      for (i = 0; i < 2; i++) {
        THIS->line_p[i] = THIS->line_p[i+1];
      }
      THIS->line_p[2] = (Line_t*)STREAM_IN_POP(
          FAST_PORT_IN, 1);
    }

    STREAM_IN_RELEASE (FAST_PORT_IN, 2);

    header->... = cornerCount;
    STREAM_OUT_PUSH(FAST_PORT_OUT, cornerCount+1);
  }
}
```

Fig. 10 The FAST actor KPN definition.

fetching reference image line by line led to poor performance.

Choosing actor granularity represents an important trade-off: finer granularity reduces the actor memory

```
void fast9_detect (int xsize, Line_t *line[3], ...,
      int * num_corners) {
  *num_corners = 0;
  for (x = edge_threshold; x < xsize -
      edge_threshold; x++) {
    ... compute keypoint or not ...
    if (corner) {
      Keyp_t * token = (Keyp_t*)STREAM_OUT_RESERVE
          (FAST_PORT_OUT, 1);
      *token = *corner;
*num_corners++;
    }
  }
  return;
}
```

Fig. 11 The FAST actor fast9_detect function.

footprint while increasing the synchronization overhead [5]; coarser granularity suffers very little synchronization overhead but may require too much memory. Although granularity vs. performance is an application-specific trade-off, the parallelization should preserve application's *natural* granularity. In this context *natural* means as close to the intrinsic algorithm structure as possible. In an image processing application, choosing one image line as a dataflow token is natural because it corresponds to the second level in the image processing nested loop: (1) frame, (2) line, (3) pixel. As an alternative, sets of lines, tiles, or similar, are less *natural* in a sense that they are algorithm-specific, require some non-intuitive changes to the initial application code, and result is often radically different from the sequential algorithm.

Refining actors' granularity requires changing its `WORK` function. Figures 10 and 11 show the `FAST` actor with refined input granularity: the input token corresponds to one image line. Compared to the previous actor version from Figure 6, the outermost *level* loop has been moved to the `WORK` function. Since the `fast9_detect` works on three lines at a time, we pass it a window of three lines, `THIS->line_p`, which is rotated on every iteration of the `WORK` function. The `STREAM_IN_POP` and the `STREAM_IN_RELEASE` implement the consumer side StreamDrive communication protocol.

Once the granularity of the actors has been reduced, the communication channels can be moved to the L1 memory. However, some channels may need to buffer too many tokens to fit with the L1 memory. For example, the `SRC` actor transfers the input image from the external to L1 memory one line at a time, while the `FAST` and the `HARRIS` actors consume these lines also one at a time. However, the `ANGLE` cannot start con-

[5] The synchronization overhead includes actions required to verify the token availability, and the associated scheduler actions

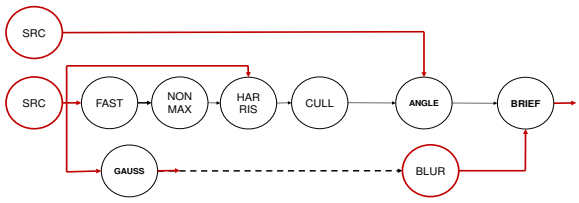

Fig. 12 Refined ORB dataflow graph: most communication buffers have been moved to the L1 memory. The `SRC` to `HARRIS`, and the `GAUSS` to `BRIEF` buffers do not fit with the L1 memory, therefore two additional DMA actors are added, the second `SRC` and the `BLUR`.

suming the input image lines until the entire image has been seen and processed by the `CULL` actor. Therefore, the communication channel needs to buffer the entire image and is, thus, too big to fit the L1 memory. In such cases, the communication channel buffer is allocated in the external memory with the DMA actors ensuring data transfer between the external and the L1 memories. The refined ORB dataflow graph is shown in Figure 12.

The refinement transformation step enables parallel execution for the first time: actors can execute their work-functions in parallel, synchronizing at *reserve* and at *pop* points.

4.2.4 Adding application-specific hardware blocks

Before further optimization and introduction of the firing rules, it is convenient to perform software-hardware partitioning at this point. For example, Table 3 shows the breakdown of ORB kernel's execution time from the original application (first image pyramid):

Kernel	Execution Time, Mcycles
fast9_detect	3,77
fast9_score	0,39
nonmax_suppress	0,32
ComputeHarrisResponse	0,81
CullKeypoints	0,34
ComputeOrientation	0,55
ComputeGaussFiltering	7,66
ComputeDescriptors	2,09
Total	15,93

Table 3 The ORB execution time breakdown.

The Gaussian filtering kernel largely dominates the application execution time and, considering that filtering is a quite common function in image processing, is a good candidate for being implemented as an application-specific hardware block.

With StreamDrive, integration of application-specific hardware blocks does not require changing the dataflow graph. Instead, it is sufficient to change actor declaration from `STREAM_DECLARE_ACTOR` to the one declaring a hardware block, the `STREAM_DECLARE_HWBLK`. The StreamDrive runtime will transparently handle the hardware block actor during the execution.

4.2.5 Data Parallelism

The above transformation steps build a dataflow graph by identifying and exposing the *functional parallelism*, where multiple actors form execution pipeline over the input stream of data. Another important type of parallelism is the *data parallelism*. In data parallelism, multiple instances of the same actor are simultaneously created. The data parallelism leads to efficient execution when the computations are not data dependent and regular: (1) it is easy to identify and to expose, (2) it has lower parallelization overhead compared to the functional parallelism.

In the context of the StreamDrive, the data parallelism also allows to balance actors workload facilitating the work of the runtime scheduler. In the ORB application, the `fast9_detect`, the `ComputeHarrisResponse`, the `ComputeOrientation`, and the `ComputeDescriptors` kernels are regular and are easy to data parallelize. Parallelizing these kernels into a number of data-parallel instances has several advantages: (1) it balances the workloads of graph actors, and (2) it creates more actors for the scheduler to choose from. From the Table 3, the workload of the `fast9_detect`, is few times that of the `nonmax_suppress` or of the `CullKeypoints` kernels, and dividing its workload among several data parallel instances helps balancing the workload of all these actors.

Unlike the standard dataflow implementations, the StreamDrive includes the *broadcast* and the *collect* connections for efficiently supporting the data parallel actors (see section 4.4). The *broadcast* enables sharing of the input tokens by the data-parallel actors, while the *collect* allows sharing the output tokens. Using these connections, it is very easy to build data-parallel actors. Two data sharing strategies can be considered:

1. The *single token* data parallel actors work all on the same input or output token, but each on different part of it, for example a different part of an image line.
2. The *multiple token* data parallel actors work each on one of `N` tokens in parallel and synchronize on all `N` tokens simultaneously.

The data sharing strategies apply to individual input or output ports, and therefore it is perfectly possible to mix different data parallel strategies within the

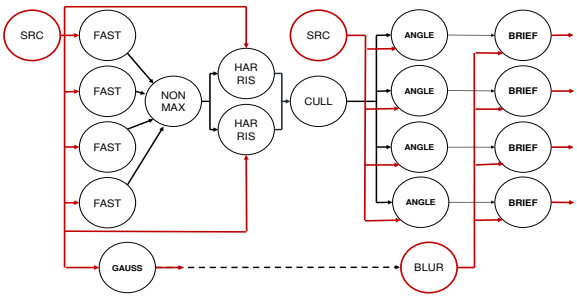

Fig. 13 The ORB dataflow graph with data-parallel actors: the `FAST`, the `ANGLE` and the `BRIEF` actors are replicated 4 times, and the `HARRIS` actor is replicated 2 times.

same actor, at the same time having channels which do not implement any data parallel sharing.

The Figure 13 shows the ORB dataflow graph with data-parallel actors. The *broadcast* connections are used to share input tokens of these actors. The *collect* connections are mostly optimized away, only the `HARRIS` actor data-parallel instances use the *collect* connection to share its output tokens. The `FAST` actor does not use the *collect* connection for its output because the downstream `NONMAX` actor needs the `FAST` corners to arrive in the raster scan order of the image. However, since the number of the corners in each image line is not known in advance, it is impossible for them to share the communication channel. As a solution, the `NONMAX` actor has 4 input ports, one for each upstream `FAST` actor, and reads them in a round-robin order ensuring that the `FAST` corners arrive in the raster scan order of the input image. Similarly, instead of using the *collect* connection for the output of the `ANGLE` actor, and then re-*broadcast* it to the `BRIEF` actors, we connect each `ANGLE` actor directly to the corresponding `BRIEF` actor, thus gaining efficiency.

The dataflow graph in Figure 13 shows the version with 4 `FAST`, 4 `ANGLE`, and 4 `BRIEF` data-parallel instances, as well as 2 `HARRIS` instances. Our implementation is parameterized in terms of the number of actors, their granularities, and the communication buffer sizes: it can be configured for 1 PE with one instance of each actor up to 8 PEs with 8 instances of the `FAST`, `ANGLE`, and `BRIEF` actors.

Figure 14 shows the data parallel `FAST` actor. The new `THIS->idx` private field corresponds to the index of this data parallel instance among the 4 data parallel instances. This index is initialized via the actor constructor. The `FAST` actor implements the *multiple token* data parallel sharing in its input port. The changes to the actor's `WORK` function are minimal: the actors handle a shared rotating input window of 6 lines instead of 3 lines, while each actor processes only those lines that correspond to this actors' index. The actor output is not changed since every `FAST` data parallel

```
typedef struct {
  uint8_t    idx;
  uint32_t   cFastThreshold;
  uint8_t *  line_p[6];
} fast_t;
STREAM_DECLARE_ACTOR_TYPE(FAST,fast_t);

// Ports
#define FAST_IN_ESIZE            (sizeof(Line_t))
#define FAST_OUT_ESIZE           (sizeof(Keyp_t))
#define FAST_PORT_IN             0
#define FAST_PORT_OUT            1
#define FAST_PORT_COUNT          2
```

```
STREAM_CONSTRUCTOR (void *arg) {
  uint32_t idx = (uint32_t)arg;
  STREAM_ACTOR_MAKE_PORT_IN(FAST_PORT_IN, "in_p",
      FAST_IN_ESIZE);
  STREAM_ACTOR_MAKE_PORT_OUT(FAST_PORT_OUT, "out_p
      ", FAST_OUT_ESIZE);
  THIS->idx = idx;
}
STREAM_DESTRUCTOR (...) {
  STREAM_ACTOR_TERM_PORT_IN(FAST_PORT_IN);
  STREAM_ACTOR_TERM_PORT_OUT(FAST_PORT_OUT);
}
STREAM_INIT() {}
STREAM_WORK() {
  int16_t fastThreshold = THIS->_cFastThreshold;
  uint8_t n_levels = cfg->n_levels;
  uint32_t level;

  for (level = 0; level < n_levels; ++level) {
    int xsize = cfg->img_width[level];
    int ysize = cfg->img_height[level];

    Keyp_t * header = (Keyp_t*)STREAM_OUT_RESERVE (
        FAST_PORT_OUT, 1);

    int cornerCount = 0;

    // Build FAST window
    for (i = 0; i < 4+2; i++) {
      THIS->line_p[i] = (Line_t*)STREAM_IN_POP(
          FAST_PORT_IN, 1);
    }

    for (y = edge_threshold; y < ysize -
        edge_threshold; y++) {
      if (y % 4 == THIS->idx) {
        int count;
        fast9_detect (level, &THIS->line_p[THIS->
            idx], ..., &count);
        cornerCount += count;

        // Rotate FAST window
        STREAM_IN_RELEASE (FAST_PORT_IN, 4);
        for (i = 0; i < 2; i++) {
          THIS->line_p[i] = THIS->line_p[i+1];
        }
        for (i = 0; i < 4; i++) {
          THIS->line_p[i+2] = (Line_t*)
              STREAM_IN_POP(FAST_PORT_IN, 1);
        }
      }
    }

    STREAM_IN_RELEASE (FAST_PORT_IN, 4+2);

    header->... = cornerCount;
    STREAM_OUT_PUSH(FAST_PORT_OUT, cornerCount+1);
  }
}
```

Fig. 14 Data-parallel version of the FAST actor

```
typedef struct {
  uint8_t    idx;
  uint8_t    state;
  uint8_t    level;
  uint32_t   cFastThreshold;
  uint8_t *  line_p[6];
  Keyp_t * header;
  uint16_t cornerCount;
  uint16_t y;
} fast_t;
STREAM_DECLARE_ACTOR_TYPE(FAST,fast_t);
// States
#define FAST_IDLE          0   // actor initial state
#define FAST_LEVEL         1   // one iteration of the outermost loop
#define FAST_LEVEL_END     2   // iteration control
#define FAST_LINE          3   // one iteration of the second level loop
#define FAST_LINE_END      4   // iteration control
// Ports
#define FAST_IN_ESIZE              (sizeof(Line_t))
#define FAST_OUT_ESIZE             (sizeof(Keyp_t))
#define FAST_PORT_IN               0
#define FAST_PORT_OUT              1
#define FAST_PORT_COUNT            2
```

Fig. 15 ORB FAST dataflow actor definition.

instance handles its own (not shared) output channel. The `fast9_detect` function remains unchanged.

It is worth noticing that the StreamDrive offers great flexibility in connecting and synchronizing the data parallel actors. By buffering the input and output tokens, actors data parallel instances do not need to start and stop processing simultaneously, thus benefiting from the efficiency of pipelined execution. Finally, creating a few data parallel actors, we remain within the scope of a small-scale data parallelism (as opposed to the massive data parallelism with hundreds or thousands of parallel instances) matching well with the scale of the target architecture cluster.

4.3 Optimizing Scheduling via Firing Rules

Execution of the refined and parallelized dataflow graph can be optimized by introducing dataflow *firing rules*.

In *KPN execution mode*, software actors require the ability to suspend an actor on a blocked *pop* (or *reserve*), and to resume its execution when sufficient tokens (or empty FIFO entries) are available. Suspending and resuming actors implies costly context-switching. In the *dataflow execution mode* the *firing rules* give preconditions for actor execution by ensuring that there are enough input tokens (or room in output FIFOs) for the actor not to be blocked. Thus, dataflow mode allows the context-switch free, *cooperative*, scheduling.

In the dataflow mode, actor's `WORK` function is subdivided into a sequence of *firings* [9, 29]. During a firing, the actors *reserve* and *pop* tokens similar to the KPN mode, but the *firing rules* ensure that the actor is never

```
STREAM_CONSTRUCTOR (void *arg) { ... }
STREAM_DESTRUCTOR (...) { ... }
STREAM_INIT() {
  SET_PORT_QUOTA (FAST_PORT_IN, 4+2);
  SET_PORT_QUOTA (FAST_PORT_OUT, MAX_IMAGE_WIDTH/2);
  THIS->state = FAST_IDLE;
}
STREAM_WORK() {
  int16_t fastThreshold = THIS->_cFastThreshold;
  uint8_t n_levels = cfg->n_levels;
  switch (THIS->state) {
  case FAST_IDLE:
    THIS->level = 0;
    SET_PORT_QUOTA (FAST_PORT_IN, 4+2);
    THIS->state = FAST_LEVEL;
    // Fallthrough to LEVEL
  case FAST_LEVEL:
    THIS->header = (Keyp_t*)STREAM_OUT_RESERVE (FAST_PORT_OUT, 1);
    for (i = 0; i < 2; i++) {
      THIS->line_p[i] = (uint8_t*)STREAM_IN_POP(FAST_PORT_IN, 1);
    }
    THIS->cornerCount = 0;
    THIS->y = edge_threshold;
    SET_PORT_QUOTA (FAST_PORT_IN, 4);
    THIS->state = FAST_LINE;
    break;
  case FAST_LINE:
    int xsize = cfg->img_width[THIS->level];
    int ysize = cfg->img_height[THIS->level];
    int count;
    fast9_detect (level, &THIS->line_p[THIS->idx], ..., &count);
    THIS->cornerCount += count;
    // Rotate FAST window
    STREAM_IN_RELEASE (FAST_PORT_IN, 4);
    for (i = 0; i < 2; i++) {
      THIS->line_p[i] = THIS->line_p[i+1];
    }
    for (i = 0; i < 4; i++) {
      THIS->line_p[i+2] = (Line_t*)STREAM_IN_POP(FAST_PORT_IN, 1);
    }
    THIS->y += 4;
    if (THIS->y < ysize) break;
    // Fallthrough to LINE_END
  case FAST_LINE_END:
    STREAM_IN_RELEASE (FAST_PORT_IN, 4+2);
    THIS->header->... = THIS->cornerCount;
    STREAM_OUT_PUSH(FAST_PORT_OUT, THIS->cornerCount+1);
    THIS->level += 1;
    if (level < n_levels) {
      SET_PORT_QUOTA (FAST_PORT_IN, 4+2);
      THIS->state = FAST_LEVEL;
      break;
    }
    // Fallthrough to LEVEL_END
  case FAST_LEVEL_END:
    SET_PORT_QUOTA (FAST_PORT_IN, 0);
    STREAM_EXIT();
  }
  STREAM_YIELD();
}
```

Fig. 16 ORB FAST actor with dataflow firing rules

blocked during the firing. When a firing is completed, actor returns control to the scheduler without requiring a context switch via the `STREAM_YIELD` call. The dataflow actor `WORK` function is "fired" by the scheduler until the `STREAM_EXIT` call signals the scheduler that actor completed its execution and does not require anymore firings.

Introducing firing rules requires to once more change actor's `WORK` function. Figures 15 and 16 show the ORB `FAST` actor converted to the dataflow mode. The KPN version of the actor from the Figure 14 consisted of a loop nest with the outermost loop iterating over the image pyramid levels, the second level over the input image lines, and the innermost level iterating over the image pixels. First, we need to choose what one actor firing should be: the firing workload determines how many tokens this firing requires for the execution and therefore directly impacts actor's memory footprint. Thus, for our `FAST` actor we choose the second level loop, iterating over input image lines, as a firing unit. Next, all loops in the loop-nest above the chosen level need to be converted to a state machine. This conversion is relatively straightforward. The state machine states correspond to the loop-nest levels of the KPN actor: the `FAST_IDLE` corresponds to the initial state, the `FAST_LEVEL` and `FAST_LEVEL_END` to the outermost level loop, and the `FAST_LINE` and `FAST_LINE_END` to the second level loop. Before the `WORK` function yields the control to the scheduler, a transition to the next state needs to be specified by setting the private `THIS->state` variable. In addition, the firing rules can be given for the next firing via the `SET_PORT_QUOTA` call. The `SET_PORT_QUOTA` function takes two arguments, the input or output port id and the number of tokens to expect in that port before the firing can take place. The initial state and the initial firing rules can be specified inside the actors' `STREAM_INIT` function. Notice that by default, unless set by the actor, the firing rules are not set and the actors' *reserve* and *pop* calls become blocking similar to the KPN execution mode.

One important point about converting the graph into the dataflow form is that all variables live across multiple actor firings need to be saved by the actor before the end of the firing and restored in the next firing. For this, such variables need to be added to actors' private state, similar to local variables `level`, `header`, `y`, and `cornerCount` from the `FAST` actor.

4.3.1 Further refinement and optimization

In the embedded domain, the cost of the system and the power consumption are directly related to the system memory size, and therefore reducing application memory footprint is very important. The dataflow program memory footprint depends on the communication buffers size and is finally related to the actors' granularity. The coarser the actor granularity, the bigger is the memory footprint. On the other hand, when the granularity of a program is very fine, the intrinsic overhead of the runtime has a high impact on efficiency. Thus, the optimization objective consists in finding the best trade-off between the communication buffer sizes and the parallelization overhead.

This step is the most time-consuming of the entire transformation process since the developer needs to choose from many different possibilities leading to different trade-off results. For example, we have noticed that processing the `NONMAX`, `HARRIS`, or `CULL` one keypoint per firing is inefficient because the amount of work per keypoint is small relative to the actor invocation overhead. One possibility that we explored was to combine the three actors together thus creating larger workload per keypoint. While this works well with smaller number of processing resources (less than 4 processing elements), when the number of processing resources increases, the resulting bulky actor is difficult to efficiently schedule and balance with other actors. On the other hand, we have noticed that several keypoint are usually simultaneously available for processing by the above actors. Therefore, we use the StreamDrive *multi-token* version of the communication API for increasing the firing working set of the actors and to reduce the parallelization overhead. Notice that the StreamDrive flow treats increasing the working set granularity as an optimization task within a well defined reference frame - number of tokens per actor firing. At the same time, preserving tokens *natural* granularity allows optimized application keep algorithmic description close to the original code.

As a general rule, the optimization process should first search for the possibility to combine actors together - this has additional benefits of reducing the overall buffer requirements since intermediate buffers between the combined actors can often be eliminated, and of reducing the schedulers' workload since fewer actors are active in the system. Then, the optimization should work to increase the number of tokens used in actors firings until an acceptable trade-off between the performance and the memory footprint is found.

This sections' example illustrates several important points from the StreamDrive:

- The StreamDrive incremental transformation flow facilitates parallelization of sequential applications into the dataflow implementation. For example, the original `fast9_detect` code incrementally undergoes relatively simple modifications during the transformation process: using the rotating window of image lines instead of the full image; addition of the StreamDrive communication primitives.
- The StreamDrive does not impose any specific language restrictions on reference code in order to be parallelized.

- Unlike canonical dataflow, the StreamDrive allows usage of shared global variables. Shared variables are very efficient way of communicating in a shared memory environment and it facilitates porting existing sequential reference code. In our example, the `FAST` actor relies on global `cfg` for retrieving parameters such as image width and height, etc. These parameters are also used by other actors and, instead of duplicating the set of these parameters for each actor, they are implemented as a shared data structure. The coherent use of the shared data remains developer's responsibility.
- The StreamDrive runtime simultaneously supports two execution modes, the KPN and the dataflow execution. This is essential for enabling our incremental transformation flow.

Following sections describe important points of the StreamDrive communication layer and runtime system implementation.

4.4 The StreamDrive Communication Layer

In order to gain higher efficiency, StreamDrive relies on a fixed-buffer implementation, i.e. token sizes and buffer sizes need to be specified at graph construction time and cannot change during graph execution. The drawback of this is that deadlocks cannot be resolved at runtime. However, the experience is that most applications exhibit a regular communication behavior that allows software developer to quantify the capacity of the FIFO buffers such that deadlock will not occur. Nevertheless, the StreamDrive provides the runtime timeout service that allows detecting the deadlock condition. Upon detecting a deadlock, the StreamDrive gives debug information about the state of the dataflow graph which helps the developer to eliminate the deadlock.

A standard dataflow FIFO implementation where data must be copied from a source actor to the communication buffer and then from the communication buffer to the destination actor, causes a significant execution overhead. Instead, the StreamDrive implementation leverages the cluster shared memory and gives actors direct access to shared communication buffers avoiding memory copy operations. The direct access to communication buffers is enabled by using the StreamDrive communication protocol described in section 4.1.

The dataflow model of computation defines a single source and a single destination communication FIFO buffers. This is an essential requirement for ensuring the dataflow execution properties and correctness. On the other hand, this also creates a significant execution overhead: when a source actor is connected to multiple destination actors, a special `copy` actor needs to be inserted between them in order to *copy-forward* the data from the source to each destination. As a result, several copies of the same data must be made and several copy operations executed, one for each destination actor.

Instead, the StreamDrive API defines a special *broadcast* connection which allows one source actor and multiple destination actors to share a single FIFO buffer. A *release* operation on such buffer is valid when all destination actors have released the buffer.

Finally, the baseline dataflow model does not provide efficient support for data-parallelism. Typically, some sort of `split` and `join` actors need to be inserted around a data-parallel actor to *copy-forward* tokens in a round-robin order to multiple data-parallel actor instances. This leads to significant overhead: the memory overhead for holding multiple copies of the same token; the performance overhead for performing multiple copy operations and for scheduling the `split` and the `join` actors.

In StreamDrive, we avoid having these additional `split` and `join` actors by leveraging on the above *broadcast* connection and its symmetric *collect* connection. The *collect* allows multiple source actors to be connected to a single destination actor and share a communication buffer. A *push* operation on such buffer is valid when all source actors have signaled a ready token. A data-parallel actor, then, can be constructed by connecting multiple parallel actor instances via the *broadcast* connection to source actors and via the *collect* connections to destination actors. Sharing communication buffers gives a choice of data-parallel implementation: data-parallel actors may choose to process a sub-part of a single token each, or to process a different token each, whichever results in lower parallelization overhead.

The *broadcast release* and the *collect push* operations are internally supported by the StreamDrive runtime system. Therefore, there is no need to schedule these operations - the runtime knows when the *broadcast release* or the *collect push* needs to be executed. For example, a *broadcast release* will only be executed if the broadcasting output port is blocked on FIFO full condition. Such dedicated support to the *broadcast* and the *collect* connections ensures optimal runtime execution.

4.5 The StreamDrive Runtime System

The StreamDrive runtime system provides application with a communication layer and the dynamic scheduler. It is implemented as a user-level library avoiding costly system calls and enabling optimization such as inlining function calls, etc.

The runtime system is fully distributed with regard to the processors - there is no one process dedicated to the runtime system duties. Instead, each processor concurrently (1) performs its own scheduling and (2) handles synchronization actions related to the actor being executed by the processor. As a result, the StreamDrive dataflow scheduler is fully dynamic: it assigns and schedules actors for execution dynamically at runtime. The scheduler uses a simple round-robin heuristic for selecting next actor to execute. The StreamDrive runtime system is still centralized from the point of view of the memory because the runtime system uses a single, global scheduling list. Our evaluation in section 5 shows that we achieved an efficient distributed implementation with respect to Amdahl's upper bound.

As explained earlier, StreamDrive supports two execution modes, the dynamic dataflow and the KPN. In the dynamic dataflow mode, actors *run-to-completion* and therefore one runtime stack per processing element can be used during the execution, and these runtime stacks are reasonably small and fit inside the shared TCDM memory. In the KPN mode, actor execution can be suspended on a blocking condition (they do not *run-to-completion*) - therefore each actor requires its own dedicated runtime stack. Placing too many actor stacks inside the TCDM memory raises an important difficulty because this memory is relatively small. For example, the 8 processing elements version of ORB application (see section 5) has 30 actors. Given a stack size of 2K per actor, the total stack memory requirement would be 60KB, which corresponds to almost a quarter of the total available TCDM memory in the ASMP cluster.

The individual actor stacks should be allocated in the larger external memory. On the other hand, placing the runtime stack inside the external memory with long access latency, leads to a very inefficient, low performance execution. We address this difficulty by implementing a stack *spilling* strategy. For this, we allocate one runtime stack per processing element inside the shared TCDM memory in both execution modes (the total number of these stacks is independent of the number of application actors). The individual actor stacks are also allocated in the larger external memory. During the execution, actors use the TCDM allocated runtime stack. When an actor gets blocked during the execution, a context switch occurs, where the actors' register context is saved to a location inside the external memory. Together with saving actors' register context, the current runtime stack is also *spilled* to the actors' external stack location. When a blocked actor resumes execution, its register context is restored, and also its stack content is *reloaded* to the runtime TCDM stack from the external stack location. Notice that in the dataflow execution mode, actors do not get blocked and no context switch and stack spilling are necessary.

Using the above stack-spilling strategy increases the cost of a context-switch: in addition to usual saving and restoring registers, the stack contents need to be saved and restored as well. In order to alleviate the problem, StreamDrive optimizes the KPN execution as follows: (1) our runtime scheduler minimizes the context-switch occurrences, (2) we have implemented optimized, hand-crafted code for the context switch routines. Moreover, in our experiments we have observed that, typically, the total number of bytes of stack that need to be spilled is quite small, and penalty for stack spilling is comparable to that of register context switching (see discussion in section 5.5). As a result, this strategy remains more efficient compared to executing a program with runtime stack inside the external memory.

One side effect of the stack spilling approach is that actor assignment to processing elements in KPN mode is no longer dynamic. Indeed, because different processing elements' runtime stacks point at different addresses in the TCDM memory, a suspended actor can only resume its execution in the same processing element (address space) where it has been suspended. As a result, each KPN actor keeps execution in the same processing element where it has begun its execution. In order to optimize assignment of KPN actors to processing elements, StreamDrive provides a special API for assigning processor affinities to dataflow actors. However in general, we consider the KPN execution as an intermediate step during the incremental transformation process from a sequential reference code into a dataflow implementation with no context switching.

5 Performance Evaluation

To gain insight into the performance of the StreamDrive framework, we present the detailed analysis of the ORB application [49] described in detail in section 4. The first StreamDrive implementation is targeting small mobile camera systems with resolutions not exceeding VGA quality, 640x480 pixels. We present results obtained with a single, non-scaled, ORB pyramid frame containing 2,651 `FAST` keypoints and limiting the number of sorted keypoint to 764. This configuration is representative of the most demanding processing requirements for this scenario.

For our measurements we used a high level multi-threaded simulator dedicated for modeling the ASMP platform (see section 3) with the objective to evaluate architectural trade-offs in earlier stage. Our simulation platform models the number and type of processing elements, how they communicate and how the

memory is organized. The simulation platform is based on proprietory multithreaded design with a scheduling kernel which is responsible for the interaction with the architecture model to drive the simulation process. The platform integrates time-approximate simulators for programmable cores, which run as dedicated POSIX threads. The programmable cores are simulated with execution time approximation accuracy error below 10% compared to a cycle-accurate execution. All other components use a single POSIX thread and are scheduled in a cooperative fasion in order to speed-up the simulation. Finally, our simulation platform also models memory and interconnect conflicts giving a very reasonable simulation accuracy at a decent simulation speed.

The ORB actors' execution times are largely unbalanced, eg. the BRIEF requires double processing time compared to the ANGLE, the FAST is almost double processing time of the BRIEF, while NONMAX, HARRIS and SORT represent together less than 10% of the application processing time. Therefore, our parallelization strategy focused on balancing the actors execution times by creating multiple data-parallel actor instances, and pipelining actors in order to achieve the efficient execution as we described in section 4.

Moving from a sequential reference C code to the initial dataflow graph is a smooth process. Starting with the initial dataflow graph, we have incrementally derived the optimized dataflow implementations for the application shown in the Figure 13. The biggest effort went into optimizing the dataflow graph with the objective to strike the optimal trade-off between the parallelization overhead and the memory footprint.

As explained earlier, our ORB graph implementation is parameterized in terms of the number of actors, actor granularities and the communication buffer sizes. In the parameterized dataflow graph, the number of data-parallel `FAST`, `HARRIS`, `ANGLE`, and `BRIEF` instances is configured depending on the number of processing cores available in the target platform. For example, the smallest ORB configuration targeting a single PE instantiates a single instance of each actor, while the biggest one targeting 8 PEs instantiates eight parallel instances of each, the `FAST`, `ANGLE` and `BRIEF` actors, and two parallel instances of the `HARRIS` actor. A choice of a particular dataflow graph configuration is dictated by the optimization objectives and should take into account the ease of developing, maintaining and evolving the parallelized code. For example, the dataflow graph in Fig. 13 resulted in best optimization trade-off for a cluster containing 4 processing elements and 64KB of the TCDM memory, while actor implementation remains very close to the initial reference

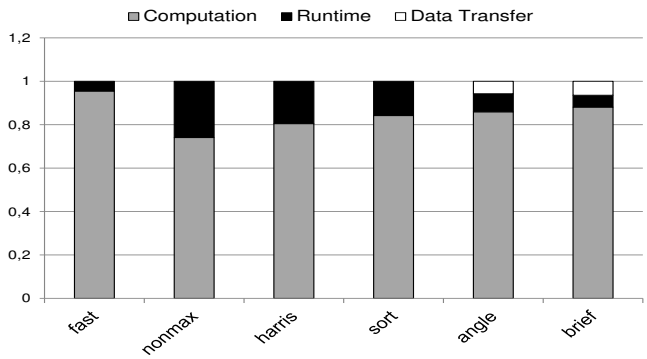

Fig. 17 ORB parallelization overhead: ratio of time spent in computation vs. data transfer and runtime tasks

C code. The capacities of communication buffers are a trade-off between the performance gain and the available TCDM memory size.

The order of application execution is deterministic, therefore debugging the dataflow code is similar to debugging the sequential one. One important difference concerns the dataflow deadlock that can occur if the communication buffers are incorrectly dimensioned. In our experience, dataflow applications exhibit a regular communication behavior so that quantifying the correct capacity of the buffers is relatively straightforward. A more formal approach for determining the dataflow buffer sizes has been proposed in [46], for example.

The following subsections analyze in details the StreamDrive parallelization overhead, memory footprint, and performance scaling under the optimistic assumption of external memory latency of 1 processor cycle and available external memory bandwidth of 8 bytes per processor cycle. We then show that StreamDrive maintains robust performance when we increase the external memory latency and reduce the available external memory bandwidth.

5.1 Parallelization Overhead

The parallelization overhead is a penalty paid for parallelizing an application. The StreamDrive parallelization overhead results from the runtime overhead including the `RESERVE`, `PUSH`, `POP`, and `RELEASE` functions, and from the DMA management for moving the data between external memory and the TCDM. This parallelization overhead does not include scheduling, which we evaluate later in this section. The parallelization overhead is *scalable*, i.e. from Amdahl's law perspective it contributes to the parallelizable part of the application.

In order to evaluate the StreamDrive parallelization overhead, we measure the performance of the ORB graph configured for 1 PE. The Figure 17 shows the

breakdown of ORB actors execution into the computation, the runtime, and the data transfer management time.

The `FAST` performs heavy computation for each image pixel and therefore its parallelization overhead is small, 4.7% of actor's execution time. The `NONMAX` actor, on the other hand, has very little computation per pixel and suffers the heaviest parallelization overhead of all, 35.0%. Similar to `NONMAX`, the `HARRIS` and the `SORT` actors perform relatively little computation per token and suffer from higher parallelization overheads, 24.2% and 18.7% respectively. One possibility that we explored is to merge the three actors into a single bigger actor. However, this only works well when parallelization degree is low (less than 4 processors) because the `NONMAX` and the `SORT` require sequential processing and the resulting actor is difficult to load balance with the rest of the application. We decided to favor better load balancing after having observed that the concerned actors' combined processing time represents less than 10% of the total application time (not counting the Gaussian filter). Finally, the `ANGLE` and the `BRIEF` actors include both the runtime and the data transfer management overhead, because they manage the DMA for transferring reference windows around each keypoint from external memory to the TCDM. Their runtime overhead is 9.2% and 5.9%, while the data transfer management overhead is 6.2% and 6.7% respectively. Relatively high data transfer management overhead corresponds to many rather small transfer requests at the chosen dataflow actor granularity.

5.2 Memory Footprint

Application memory footprint determines how much memory the application needs for execution. The StreamDrive application memory footprint includes the application data, the run-time system footprint including the run-time stack, and the dataflow buffers.

In terms of the run-time system memory requirements, the debug version of the StreamDrive library uses 944 bytes of static data. It also needs 64 bytes of memory per actor in addition to actor private data, and up to 60 bytes per communication channel, depending on channel type. For comparison, an image line of a VGA image has a size of 640 bytes, while the smallest ORB keypoints buffer requires almost 300 bytes. Altogether, the ORB graph with 30 actors configured for 8 PEs required in total less than 8 KB of memory for the run-time system. The stack contribution is application-specific and depends on the size of biggest stack that any one actor may require. As explained in section 4.5, the StreamDrive implementation allocates one runtime stack per processing element inside the TCDM. In ORB, we limit to 2KB the stack space per actor, eg. the 8 PE ORB configuration required 16KB of stack space for the 8 processing elements.

The application buffering requirements are determined by the actor granularity along with the size of the communication buffers. Every dataflow channel requires a minimal FIFO buffer size that ensures a deadlock free execution [6]. Additional buffer capacity beyond such minimal size helps improve performance by reducing scheduler overhead and by absorbing communication peaks when actor computation is irregular and unpredictable.

# PEs	min	64KB	128KB	256KB
1	37132	1.04	1.09	1.10
2	45968	1.07	1.11	1.11
4	63964	1.00	1.11	1.14
8	99084		1.00	1.18

Table 4 Dataflow Performance Gain vs. Memory Footprint

Table 4 summarizes the ORB implementation memory footprint versus performance improvement associated with increasing the available memory size. For different ORB graph configurations from 1 PE to 8 PEs, the *min.* column gives the minimal memory footprint, while other columns show the performance gain (ratio relative to the min.) that can be obtained by increasing the total memory size. From the table, it can be seen that performance gain due to adding more memory rapidly leads to diminishing returns. For example, with 4 PEs, the performance increase is 11% between 64KB and 128KB, and only 3% when moving to 256KB. Note that the 8 PE version of ORB does not fit in 64KB memory with the minimal requirement close to a 100KB, due to large total number of actors leading to increased number of buffers.

5.3 Performance Scaling

The performance scaling of a parallel application indicates how much performance increases when more processing elements are added. In order to quantify the StreamDrive performance scaling, we measured the performance of the ORB graph configured for 8 PEs while varying the number of PEs. Figure 18 plots the resulting Amdahl's curve. We are observing the speedup very close to the theoretically optimal point. In the figure, the second line from the top corresponds to

[6] Unless there is uncontrolled accumulation of tokens in a channel

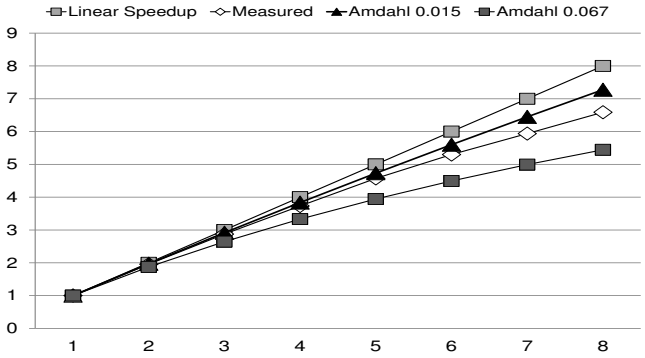

Fig. 18 ORB performance scaling: speed-up vs. number of PEs.

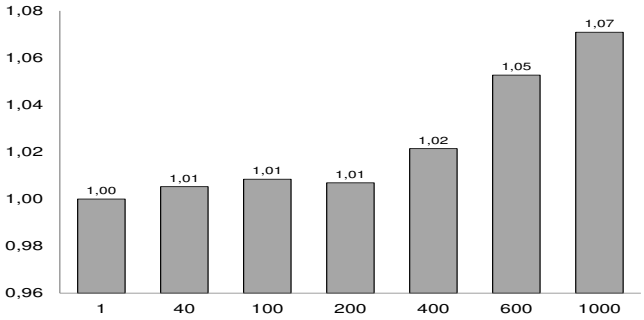

Fig. 19 ORB execution time increase vs. increased external memory latency

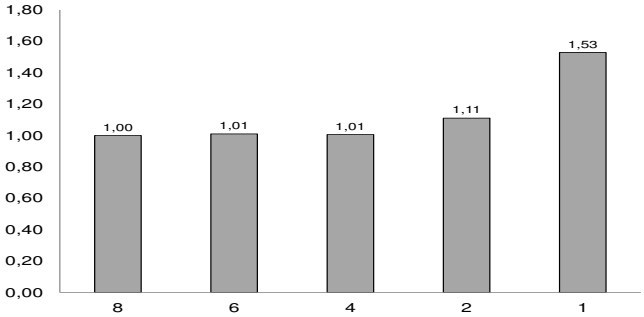

Fig. 20 ORB execution time increase vs. reduced external memory bandwidth

the Amdahl's speedup taking into account the 1.5% of non-parallelizable `SORT` part in the ORB application. The bottom line in the figure corresponds to the Amdahl's speedup taking into account additional 5.2% of the scheduler overhead measured in a single PE, if it were non-parallelizable. The measured ORB speedup lies in between these two Amdahl's curves, showing that the StreamDrive scheduler is efficiently distributed over multiple PEs such that its non-parallelizable fraction is of the order of 1.5% of the total application execution time.

In order to quantify the efficiency of our `broadcast` and `collect` implementation, we have measured the time that application spends inside these functions. The `broadcast` and `collect` processing represent 4% and 3%, respectively, of the StreamDrive scheduling overhead for scheduling 30 actors, i.e. the time for handling a `broadcast` or a `collect` is less than scheduling an actor, not even executing it.

Overall, we are observing the speedup very close to the theoretically optimal point. Note that this results have been achieved with a relatively small actor granularity. With larger input image sizes, the contribution of the `SORT` part would decrease as well as the scheduler overhead, resulting in speedups even closer to the linear.

It is interesting to compare our results with similar runtime environments. Compared to [22], the StreamDrive custom scheduler implementation is more efficient: among 30 ORB actors, we observe the average actor scheduling time of 161 cycles, versus 300 cycles for scheduling only 2 actors reported in [22]. The StreamDrive shared FIFO access is faster: less than 40 cycles versus 150. Finally, our ORB implementation with 30 actors required less than 8KB of runtime system memory versus 9KB for 2 small synthetic actors reported in [22].

Yviquel [63] reported performance scaling numbers of their dataflow implementation for MPEG-4 video decoder. For comparison, with 10 processors their reported speedup is less than 6 times. The authors explain this relatively low speedup numbers by the limit of functional parallelism in the application. This confirms our experience showing the importance of the data parallelism.

The comparison with the work in [18] was not possible since the authors did not report their efficiency numbers.

5.4 Performance w/r to External Memory

The above evaluation of the StreamDrive implementation has been carried based upon an optimistic assumption of external memory latency of 1 processor cycle and available external memory bandwidth of 8 bytes per cycle. In a System-on-Chip (SoC) environment, where multiple IPs compete for the access to the DDR memory, this assumption is not valid. In order to evaluate the StreamDrive performance with varying external memory latency and bandwidth, we used the 8PE ORB configuration. This is the most demanding configuration in terms of external memory bandwidth because it requires data to be available simultaneously for a large number of actors. Notice that the relatively small ORB actor granularity results in many modest size DMA transfers, such as one image line of only 640

bytes, or the `ANGLE` computation reference window of 31x31 bytes.

Figure 19 plots ORB performance change versus growing external memory latency from 1 to 1000 processor cycles. The figure shows that there is almost no performance degradation when external latency is smaller than 400 processor cycles. Furthermore, even when the latency is 1000 processor cycles, the performance degradation is only 7% versus the 1 cycle latency. Figure 20 plots the performance change when the available external memory bandwidth is reduced from 8 down to 1 byte per processor cycle. The performance starts to degrade visibly when the available bandwidth drops below 2 bytes per processor cycle. At 500 MHz, this corresponds to less than 2GB per second, which is quite low for a typical SoC external memory.

Our evaluation results confirm that the StreamDrive performance holds well even under long external memory latency and limited available external memory bandwidth.

5.5 KPN vs. Dataflow Trade-off

Considering that the biggest parallelization effort is required by optimizing the dataflow graph after having introduced the firing rules, we have also compared the performance achievable with the KPN execution vs. the optimized dataflow execution.

In the KPN execution mode, the number of context switches during a program execution is proportional to the available buffer sizes: the bigger are dataflow FIFO buffers, the fewer are there context switches in the KPN mode. On the other hand, our dataflow scheduling heuristic is trying to fire a given actor as long as it remains enabled. Similarly to KPN, the number of times that the scheduler switches actors is also proportional to the dataflow buffer sizes. Therefore, we observe similar *diminishing return* behavior with the KPN execution: there is a point at which adding more buffer size to the dataflow graph leads to a negligible performance gain. Unlike the dataflow execution, the KPN performance results under the minimal buffer sizes are considerably worse than the performance at the *diminishing return* point. Table 5 illustrates this point:

# PEs	min	64KB	128KB	256KB
1	37132	1.45	1.48	1.50
2	45968	1.24	1.42	1.44
4	63964	1.02	1.32	1.36
8	99084		1.05	1.22

Table 5 KPN Performance Gain vs. Memory Footprint

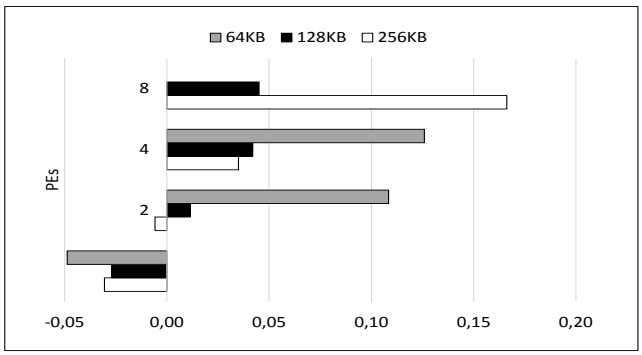

Fig. 21 ORB KPN vs. dataflow execution time, external memory latency 1 cycle

Unlike the DDN execution, the KPN execution is very sensitive to the external memory latency (and bandwidth). Figures 21 and 22 show the ratio of KPN vs. DDN execution cycles for ORB processing of one non-scaled VGA image when external memory latency is of 1 processor cycle (the external memory as as efficient as the TCDM) and 40 processor cycles, respectively. The Figures show measurements performed in different ASMP cluster configurations: TCDM memory size of 64, 128, and 256 KB, and using 1, 2, 4, and 8 processing elements. While with external latency of 1 cycle (and few processing elements), the KPN performance may even be slightly better, the DDN clearly outperforms the KPN execution in all ASMP configurations when external memory latency is 40 processor cycles. The explanation is straightforward: the cost of the KPN context switch is directly related to the external memory access time because it is not possible to hide the context saving and restoring by performing it in parallel with other computation work. For example, there were 202 context switches during the KPN execution of the ORB in 1 processing element. With external latency of 1 processor cycle, they account for less than 2% of the total execution time. When the external memory latency increases to 40 cycles, these context switches account for 10% of the total execution time. Because the KPN performance is much more affected by the external memory latency than the performance of the DDN execution, the DDN would be a better choice for real embedded systems, where the external memory latency is often a bottleneck.

The KPN performance scales worse than the DDN performance when the number of processing elements increases. In Figure 21, the KPN performance is even slightly better than the DDN, less than 5%, with 1 processing element. When 8 processing elements are used, the DDN outperforms the KPN by up to 18%. The explanation is twofold: (1) the dynamic assignment of dataflow actors to processing elements outperforms the

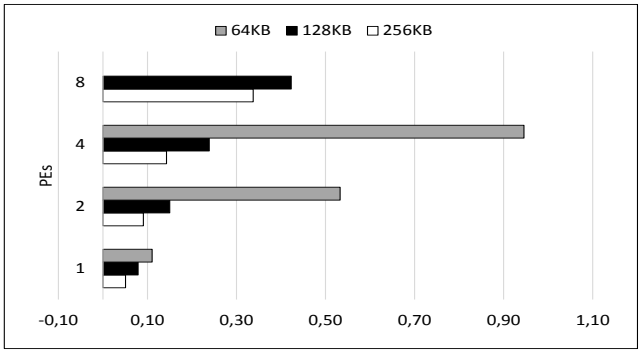

Fig. 22 ORB KPN vs. dataflow execution time, external memory latency 40 cycles

fixed KPN assignment, and (2) the relative contribution of the KPN scheduler is increasing faster than the contribution of the dataflow scheduler with more parallel execution.

To put the above performance measurements in prospective, notice that the execution time for processing one non-scaled VGA frame should not exceed 1,6M cycles at 500MHz operating frequency in order to achieve the real-time performance of 30 frames per second [7]. This level of performance can only be achieved with 8 PEs and under the dataflow execution mode. Thus, the effort spent in optimizing the dataflow graph is certainly necessary in order to achieve the target real-time objective.

6 Conclusion and Future Work

The StreamDrive framework implements the dynamic dataflow computing model. Two main contributions of the StreamDrive framework are: (1) simultaneous support for the KPN and the Dataflow execution modes, which enables the incremental parallelization flow starting with sequential reference code, and (2) an efficient runtime implementation in a resource-constrained embedded computing platform. StreamDrives' distributed runtime system provides low overhead, good scalability, and is robust versus limiting external memory bandwidth. The experience with the ORB application shows that StreamDrive is an efficient approach for parallelizing and executing embedded streaming applications.

The StreamDrive is a work in progress. The aspects of StreamDrive that need to be further investigated are primarily related to automating the optimization of the dataflow graph, and improvements to the runtime scheduler.

[7] This real-time requirement also takes the `match` part of the application into account

Acknowledgements This research was partially funded by the H2020 Project Opecomp (CA 732631) and by the ERC-ADG Project Multitherman (CA 291125). Authors would also like to thank the ST Microelectronics' Embedded Computing Systems management for supporting this research.

References

1. Bezati E (2015) High-level synthesis of dataflow programs for heterogeneous platforms: design flow tools and design space exploration. PhD thesis, COLE POLYTECHNIQUE FDRALE DE LAUSANNE
2. Bezati E, Brunet SC, Mattavelli M, Janneck JW (2016) High-level system synthesis and optimization of dataflow programs for mpsocs. In: Matthews MB (ed) ACSSC, IEEE, pp 417–421
3. Bhattacharya B, Battacharyya S (2001) Parameterized dataflow modelling for dsp systems. IEEE Transactions on Signal Processing 49(10):2408 – 2421
4. Bhattacharyya SS, Deprettere EF, Leupers R, Takala J (eds) (2013) Handbook of Signal Processing Systems. Springer
5. Bilsen G, Engels M, Lauwereins R, Peperstraete JA (1995) Cyclo-static data flow. In: ICASSP, vol 5, pp 3255–3258
6. Buck JT (1994) A dynamic dataflow model suitable for efficient mixed hardware and software implementations of dsp applications. In: HSCD Workshop, pp 165–172
7. Cockx J, Denolf K, Vanhoof B, Stahl R (2007) Sprint: A tool to generate concurrent transaction-level models from sequential code. EURASIP Journal on Applied Signal Processing 1:213
8. Dehyadegari M, Marongiu A, Kakoee M, Benini L, Mohammadi S, Yazdani N (2012) A tightly-coupled multi-core cluster with shared memory hw accelerators. In: ISCAMOS, pp 96–103
9. Dennis J (1974) First version data flow procedure language. Tech. Rep. MAC TM61, MIT Laboratory for Computer Science
10. de Dinechin BD, Ayrignac R, Beaucamps PE, Couvert P, Ganne B, de Massas PG, Jacquet F, Jones S, Chaisemartin NM, Riss F, Strudel T (2013) A clustered manycore processor architecture for embedded and accelerated applications. In: HPEC, IEEE, pp 1–6
11. Dunkels A, Schmidt O, Voigt T, Ali M (2006) Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In: Sensys, pp 29–42

12. Edwards SA, Tardieu O (2006) Shim: A deterministic model for heterogeneous embedded systems. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 14(8):854–867
13. Edwards SA, Vasudevan N, Tardieu O (2008) Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling shim to pthreads. In: Sciuto D (ed) DATE, ACM, pp 1498–1503
14. Eker J, Janneck J (2002) Caltrop—language report (draft). Technical memorandum, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, http://www.gigascale.org/caltrop
15. Eker J, Janneck JW (2012) Dataflow programming in cal – balancing expressiveness, analyzability, and implementability. In: Asilomar Conference on Signals, Systems and Computers, pp 1120–1124
16. Gangwal OP, Nieuwland A, Lippens PER (2001) A scalable and flexible data synchronization scheme for embedded hw-sw shared-memory systems. In: Hermida R, Aboulhamid EM (eds) ISSS, ACM / IEEE Computer Society, pp 1–6
17. Gautier T, Besseron X, Pigeon L (2007) Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PASCO, pp 15–23
18. Gebrewahid E, Yang M, Cedersjö G, Abdin ZU, Gaspes V, Janneck JW, Svensson B (2014) Realizing efficient execution of dataflow actors on manycores. In: EUC, pp 321–328
19. Geilen M, Basten T (2003) Requirements on the execution of kahn process networks. In: Degano P (ed) ESOP, Springer, Lecture Notes in Computer Science, vol 2618, pp 319–334
20. Girault A, Lee B, Lee EA (1999) Hierarchical finite state machines with multiple concurrency models. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 18(6):742–760
21. Goubier T, Sirdey R, Louise S, David V (2011) ΣC: A programming model and language for embedded manycores. In: ICA3PP, pp 385–394
22. Haid W (2010) Design and performance analysis of multiprocessor streaming applications. PhD thesis, ETH, Zurich
23. Haid W, Schor L, Huang K, Bacivarov I, Thiele L (2009) Efficient execution of kahn process networks on multi-processor systems using protothreads and windowed fifos. In: ESTIMedia, pp 35–44
24. Harris C, Stephens M (1988) A combined corner and edge detector. In: Proceedings of the 4th Alvey Vision Conference, pp 147–151
25. Huang K, Grunert D, Thiele L (2007) Windowed fifos for fpga-based multiprocessor systems. In: ASAP, pp 36–41
26. JT B (1993) Scheduling dynamic dataflow graphs with bounded memory using the token flow model. PhD thesis, Department of Electrical Enginnering and Computer Science, University of California at Berkeley
27. Kahn G (1974) The semantics of a simple language for parallel programming. In: IFIP Congress
28. de Kock EA, Smits W, van der Wolf P, Brunel JY, Kruijtzer W, Lieverse P, Vissers KA, Essink G (2000) Yapi: Application modeling for signal processing systems. In: DAC, pp 402–405
29. Lee E (1997) A denotational semantics for dataflow with firing. Memorandum UCB/ERL M97/3, Electronics Research Laboratory, U. C. Berkeley
30. Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proceedings of the IEEE 75(9):1235–1245
31. Mattavelli M, Amer I, Raulet M (2010) The reconfigurable video coding standard [standards in a nutshell]. IEEE Signal Processing Magazine 27(3):159–167
32. Mattavelli M, Raulet M, Janneck JW (2013) Mpeg reconfigurable video coding. In: Bhattacharyya SS, Deprettere EF, Leupers R, Takala J (eds) Handbook of Signal Processing Systems, Springer, pp 281–314
33. Melpignano D, Benini L, Flamand E, Jego B, Lepley T, Haugou G, Clermidy F, Dutoit D (2012) Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications. In: DAC, pp 1137–1142
34. Michalska M, Bezati E, Brunet SC, Mattavelli M (2016) A partition scheduler model for dynamic dataflow programs. In: Connolly M (ed) ICCS, Elsevier, Procedia Computer Science, vol 80, pp 2287–2291
35. Michalska M, Zufferey N, Boutellier J, Bezati E, Mattavelli M (2016) Efficient scheduling policies for dynamic data flow programs executed on multicore. In: 11th International Meeting on Logistics Research
36. NVIDIA (2010) Next generation cuda compute architecture: Fermi - white paper. http://www.nvidia.com
37. Olofsson A, Nordström T, Ul-Abdin Z (2014) Kickstarting high-performance energy-efficient manycore architectures with epiphany. In: Asilomar Conference on Signals, Systems and Computers, IEEE, pp 1719–1726

38. Orozco D, Garcia E, Pavel R, Khan R, Gao G (2011) Tideflow: The time iterated dependency flow execution model. In: Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM), pp 1–9
39. Pelcat M, Desnos K, Heulot J, Guy C, Nezan JF, Aridhi S (2014) Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In: EDERC, pp 36–40
40. Pimentel AD (2008) The artemis workbench for system-level performance evaluation of embedded systems. International Journal of Embedded Systems 3(3):181–196
41. Plishker W, Sane N, Kiemb M, Anand K, Bhattacharyya SS (2008) Functional dif for rapid prototyping. In: IEEE International Workshop on Rapid System Prototyping, IEEE Computer Society, pp 17–23
42. Plishker W, Sane N, Bhattacharyya SS (2009) A generalized scheduling approach for dynamic dataflow applications. In: Benini L, Micheli GD, Al-Hashimi BM, Mller W (eds) DATE, IEEE, pp 111–116
43. Plurality (2011) Plurality hypercore. http://www.plurality.com
44. Pop A, Cohen A (2013) Openstream: Expressiveness and data-flow compilation of openmp streaming programs. ACM Transactions on Architecture and Code Optimization 9(4):53
45. Rahimi A, Loi I, Kakoee MR, Benini L (2011) A fully-synthesizable single-cycle interconnection network for shared-l1 processor clusters. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011, IEEE, pp 1–6
46. Rahman AAHA, Brunet SC, Alberti C, Mattavelli M (2014) A methodology for optimizing buffer sizes of dynamic dataflow fpgas implementations. In: ICASSP, IEEE, pp 5003–5007
47. Rahman AAHBA (2014) Optimizing dataflow programs for hardware synthesis. PhD thesis, COLE POLYTECHNIQUE FDRALE DE LAUSANNE
48. Rosten E, Porter R, Drummond T (2010) Faster and better: A machine learning approach to corner detection. Pattern Analysis and Machine Intelligence, IEEE Transactions on 32(1):105–119
49. Rublee E, Rabaud V, Konolige K, Bradski G (2011) Orb: An efficient alternative to sift or surf. In: ICCV, pp 2564–2571
50. Sane N, Hsu CJ, Pino JL, Bhattacharyya SS (2010) Simulating dynamic communication systems using the core functional dataflow model. In: ICASSP, IEEE, pp 1538–1541
51. Sau C, Meloni P, Raffo L, Palumbo F, Bezati E, Brunet SC, Mattavelli M (2016) Automated design flow for multi-functional dataflow-based platforms. Signal Processing Systems 85(1):143–165
52. Schwambach V, Cleyet-Merle S, Issard A, Mancini S (2015) Estimating the potential speedup of computer vision applications on embedded multiprocessors. CoRR abs/1502.07446
53. Shen C, Plishker W, Bhattacharyya SS (2012) Dataflow-based design and implementation of image processing applications. Multimedia Image and Video Processing pp 609–629
54. Sriram S, Bhattacharyya SS (2009) Embedded multiprocessors: Scheduling and synchronization. CRC press
55. Stoutchinin A, Benini L (2017) Stream drive: A dynamic dataflow framework for clustered embedded architectures. In: Conf. Computing Frontiers, ACM, pp 1–8
56. Stuijk S, Geilen M, Thelen B, Basten T (2011) Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In: International Conference on Embedded Computer Systems, pp 404–411
57. Srot J, Berry F, Bourrasset C (2016) High-level dataflow programming for real-time image processing on smart cameras. Journal of Real-Time Image Processing 12(4):635–647
58. Ul-Abdin Z, Yang M (2015) A radar signal processing case study for dataflow programming of manycores. Journal of Signal Processing Systems pp 1–14
59. Vasudevan N, Edwards SA (2009) Celling shim: Compiling deterministic concurrency to a heterogeneous multicore. In: ACM Symposium on Applied Computing, pp 1626–1631
60. Vrba Z, Halvorsen P, Griwodz C, Beskow P, Espeland H, Johansen D (2013) The nornir run-time system for parallel programs using kahn process networks on multi-core machines - a flexible alternative to mapreduce. The Journal of Supercomputing 63(1):191–217
61. YarKhan A (2012) Dynamic task execution on shared and distributed memory architectures. PhD thesis, the University of Tennessee, Knoxville
62. Yviquel H, Sanchez A, Jskelinen P, Takala J, Raulet M, Casseau E (2014) Efficient software synthesis of dynamic dataflow programs. In: ICASSP, IEEE, pp 4988–4992
63. Yviquel H, Sanchez A, Jskelinen P, Takala J, Raulet M, Casseau E (2015) Embedded multi-core systems dedicated to dynamic dataflow programs. Signal Processing Systems 80(1):121–136

64. Zaki GF, Plishker W, Bhattacharyya SS, Fruth F (2017) Implementation, scheduling, and adaptation of partial expansion graphs on multicore platforms. Signal Processing Systems 87(1):107–125