Efficient Software Synthesis of Dynamic Dataflow Programs
Hervé Yviquel, Alexandre Sanchez, Pekka Jääskeläinen, Jarmo Takala, Mickaël Raulet, Emmanuel Casseau

To cite this version:
hal-00988003

HAL Id: hal-00988003
https://hal.archives-ouvertes.fr/hal-00988003
Submitted on 7 May 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
ABSTRACT
This paper introduces advanced software synthesis techniques that enhance the implementation of dynamic dataflow programs. These techniques have been implemented into opensource tools and demonstrated on well-known video decoders including one based on the new High Efficiency Video Coding (HEVC) standard. The results show an improvement of more than 100% of the frame-rate over previously proposed implementations, and achieve real-time decoding of high definition video sequences.

1. INTRODUCTION
The emergence of massively parallel architectures, along with the increasing complexity of applications, has revived the interest in dynamic dataflow programming. Indeed, dynamic dataflow programming offers a flexible development approach which is able to build complex and modular applications while expressing parallelism explicitly. Paradoxically, most of the studies stay focused on static dataflow programming, even if a pragmatic development process requires the expressiveness and the practicality offered by dynamic dataflow programming.

The main challenge that dynamic dataflow programs have to face is the demonstration of efficient implementations that can achieve performance constraints imposed by modern applications. For instance, video decoders have to provide real-time frame-rates for high-definition video sequences. While the efficiency of traditional language programs is the result of 50 years of work on compilers to mainly exploit memory locality, abandoning memory-oriented programming in favor of dataflow programming requires the development of new compilation techniques to fully benefit from the processor architecture.

As a result, this paper presents advanced software synthesis techniques that enhance the implementation of dynamic dataflow programs using their specific properties and the flexibility of software systems. These techniques have been implemented into open-source tools and demonstrated on well-known video decoders including one based on the new High Efficiency Video Coding (HEVC) standard.

The paper is organized as follows. First, the context of dynamic dataflow programming is described in Section 2. Then, we describe our methodology to enhance the software synthesis of dynamic dataflow programs in Section 3. Section 4 presents experimental results and compare them with previous works. Finally, we conclude in Section 5.

2. DYNAMIC DATAFLOW PROGRAMMING
Dynamic dataflow programming relies upon a model of computation called Dataflow Process Network (DPN) [1], which is closely related to Kahn Process Network (KPN). In this model, an application is represented as a directed graph wherein the vertices model computational units that are called actors and the unidirectional edges represent unbounded communication channels based on FIFO principle. The FIFO channels can be empty or can carry a possibly infinite sequence of atomic data called tokens.

Additionally to the KPN model, DPN introduces the notion of firing. An actor firing is an indivisible quantum of computation which corresponds to a mapping function of input tokens to output tokens applied repeatedly and sequentially on one or more data streams. This mapping is composed of three ordered and indivisible steps: data reading, then computational procedure, and finally data writing. These functions are guarded by a set of firing rules which specifies when the functions can be fired, i.e. the number and the values of tokens that have to be available on the input ports to fire the actor. An actor can fire when at least one of its firing rules is satisfied. When several firing rules are satisfied at the same time, a single one is chosen based on predefined priorities.

Few years ago, MPEG has introduced an innovative framework, called RVC [2], that can be considered as the first large-scale experimentation on dynamic dataflow programming. RVC has been initially introduced to overcome the lack of interoperability between the various video codecs deployed in the market. The framework allows the development of video coding tools, among other applications, in
a modular and reusable fashion thanks to the inclusion of a subset of CAL programming language [3], and the support of a complete development environment known as Orcc [4].

In general, communication and synchronization are the major sources of inefficiencies on every multi-core system. In particular, the implementation of dynamic dataflow programs faces two issues to achieve performance requirements: Scheduling and communication. Both are directly impacted by the application granularity, usually defined as the ratio of computation to the amount of communication. Video decoders are traditionally described at fine-granularity since the pixels are processed block after block. On the one hand, the scheduling is a well-known bottleneck of dynamic dataflow programs since their expressive power requires a large number of control structures. The literature has already introduced a large panel of methodologies to optimize the scheduling of dynamic dataflow programs in different manners [5, 6, 7, 8, 9]. On the other hand, the communication is the major bottleneck of all dataflow programs. Since the actors can only communicate through the FIFO channels, the execution requires a massive amount of data movements that can ultimately lead to poor performance. Restricted dataflow models usually solve this issue by grouping the data transfers, but this is not possible with dynamic dataflow models. As a result, this paper focuses on communication and computation aspects to enhance the software implementation of dynamic dataflow programs [10, 11] using the specific properties of the DPN model and the flexibility of software systems.

3. PROPOSED SOFTWARE IMPLEMENTATION OF DYNAMIC DATAFLOW PROGRAMS

In theory, the DPN model defines FIFO channels with unbounded capacity [1]. In practice, the FIFO channels are bounded to limit memory usage and avoid the overhead of dynamic memory allocation. Actually, bounded FIFO channels have been studied extensively, but the DPN model has specificities that make their implementation quite challenging. The DPN model defines action firing as an indivisible quantum of execution. Therefore, an action is fired if and only if its firing rule is valid. Thus, the implementation of FIFO channels for DPN-based programs requires the ability to check their state, i.e. the number of tokens available, during the execution, and to peek their tokens from input channels, i.e. checking values of incoming tokens without consuming them, to evaluate action fireability and thus break conventional FIFO principle.

3.1. Branch-Free Communications

In software, FIFO channels are traditionally implemented by a circular buffer allocated in a shared memory. Read and write are then achieved by accessing the buffer according to read and write indexes that are updated afterwards. Moreover, the comparison of the indexes is sufficient to know the state of the FIFO channel. Finally, a peek is a read without the update of the read index, but any token can be peeked thanks to the full accessibility of the shared memory. Using circular buffer to implement FIFO channels avoids side shuffles of data after each reading, but implies an advanced management of memory indexes that can ultimately lead to poor performance. For instance, the update of the indexes may require checking if the end of the buffer is reached to go back to the beginning.

Avoiding checks on the position of the indexes is however possible using absolute indexes with the cost of additional modulo operations. Thus, performing read and write increases the indexes infinitely until the overflow of the variables. Since computing the modulo is costly on most processor architectures, it is translated to a simple right shift by forcing the size of the buffer to a power of two. Paradoxically, such a constraint on the size of the communication channels does not have a large impact on the memory usage, especially compared to the large needs of video decoders. Indeed, the initial sizes of our FIFO channels being reasonable, the round-up to the next power of two is relatively small.

```
transp: action
  IN:[ src ] repeat 16 // Input pattern
  -->
  OUT:[ dst ] repeat 16 // Output pattern
  var
  int(size=16) dst[16] =
    [ src[ 4 * column + row ] :
      for int row in 0 .. 3,
        for int column in 0 .. 3
    ]
  end
```

Listing 1. Transposition of a 4x4 block in CAL

3.2. Copy-Free Communications

One of the high-level features of CAL is its ability to describe multi-rate actions [3], i.e. actions reading and writing pools of data at each firing, such as the transposition of 4x4 block presented in Listing 1 that reads and writes 16 tokens by firing. In fact, multi-rate actions are common for video coding since the pictures are usually processed block after block. Following this semantic, the body of a multi-rate action, such as the one described in Listing 1, is translated into a function composed of 3 steps as follows [12, 10]: 1) Reading: Incoming tokens are read in order from the input FIFO channels and stored into the local variables referenced by the input pattern. E.g., in Listing 1, 16 tokens are read from the input port IN and stored in the local array src. 2) Processing: The action is processed, as defined in its CAL description, using the local variables referenced into the input and output patterns as interfaces. As a consequence, the processing of data is not necessarily described in order. 3) Writing: Outgoing tokens are written in order from local variables referenced by the output pattern into the output FIFO channels. E.g., in List-
ing 1, 16 tokens are written successively from the local array dst to the output port OUT. While this implementation stays respectful of the FIFO principle, with the exception of the peeking, it also involves two additional copies between the circular buffer, fact, the firing rules are evaluated successively according to the partial order defined within the actor (priorities and FSM). Thus, offers and the local variables (knowing that only one copy is mandatory).

```c
void transp() {
    int indSrc, indDst;
    for(int row = 0; row<=3; row++) {
        for(int col = 0; col<=3; col++) {
            indSrc = (IN->rdInd + (4*col+row)) % IN->SIZE;
            indDst = OUT->wrInd + (row*4+col)) % OUT->SIZE;
            OUT->buff[indDst] = IN->buff[indSrc];
        }
    }
    IN->rdInd += 16;
    OUT->wrInd += 16;
}
```

Listing 2. Copy-free and branch-free action

Since our FIFO channels are implemented in shared memory without access restriction, we can remove all the additional copies to local buffers by accessing directly to the content of the FIFO channels within the processing of the action. So, accesses to input and output variables, such as src and dst, are replaced by direct accesses to FIFO channels, such as IN and OUT respectively. Unfortunately, race conditions, i.e. synchronization issues, can occur when the action processing does not ensure that the FIFO accesses are performed in order (such as the accesses to src). But, the DPN model defines an action firing as a quantum of execution [1], in other words an action firing is an atomic step that cannot be interrupted. Thus, the FIFO indexes can be updated just once at the end of the action without changing the semantic of the application, such as presented in Listing 2. Then, the implementation stays respectful of the FIFO principle of the DPN model. Indeed, other processors cannot access the FIFO rooms involved by this processing since the FIFO indexes are not updated until the action is entirely processed.

To summarize, the three first steps of action firing (Reading, processing, and writing) can be merged together, reducing the memory footprint and the number of instructions to implement the action, as long as the FIFO indexes are updated after the action processing, and thus let the other actors using newly produced data and newly released rooms.

3.3. Aligned Communications

Our branch-free implementation prevents potential optimizations due to absolute indexes. In fact, the compiler cannot know if the access are aligned in the memory or if the end of the circular buffer is reached during the execution of the current action. Thus, we generate two versions of all actions, standard (Listing 2) and aligned (Listing 3), that are executed according to the current position in circular buffers. The aligned version of the action is called whenever the tokens are linearly accessible in the buffer. So, the relative indexes can be computed only once at the beginning of the action. Additionally, the aligned accesses to the circular buffer are vectorizable since the width of the FIFO channels within our applications are often inferior to the bus width (8 or 16 bits are common values in video processing). As a result this optimization is very powerful for processors that exploits instruction-level parallelism and word-level parallelism.

```c
void transp_aligned() {
    int IN_rdInd = IN->rdInd % IN->SIZE;
    int OUT_wrInd = OUT->wrInd % OUT->SIZE;
    int ind_src, ind_dst;
    for(int row = 0; row<=3; row++) {
        for(int col = 0; col<=3; col++) {
            ind_src = IN_rdInd + (4*col+row);
            ind_dst = OUT_wrInd + (row*4+col);
            OUT->buff[ind_dst] = IN->buff[ind_src];
        }
    }
    IN->rdInd += 16;
    OUT->wrInd += 16;
}
```

Listing 3. Dependence-free action

3.4. Multi-level Dynamic Scheduling

As defined by Lee and Parks [1], the execution of a DPN-based actor is modeled by the repeated evaluation of the firing rules that are, in case of a success, followed by the firing of the associated action. This process is usually defined as the action scheduling. The action scheduler can be implemented by a simple function that evaluates the firing rules in order [11] such as presented in Listing 4. In theory, the scheduler evaluates only two conditions to determine the fireability of an action: the input pattern, the amount of tokens required in the input channel (hasTokens), and the guard, the potential condition on the values of tokens and/or state variables (isSchedulable). In practice, the scheduler has also to evaluate the output pattern so as to ensure that enough rooms are available in the output channels to allow the firing of the action without blocking (hasRooms). While the validation of the output pattern is not required by the DPN model, it is necessary when several actors are executed concurrently on the same processor. Indeed, waiting for the availability of an output channel, using blocking writes for instance, inevitably leads to a deadlock if the target of the channel, the consumer, is mapped to the same processor. Additionally, the scheduler checks if a sufficient number of tokens are aligned in the FIFO channels to be able to execute the optimized version of

```c
Listing 4. DPN-based actor
```
the action \((\text{areAligned})\).

Apart from this internal scheduling, the execution of a DPN program in a concurrent environment requires actor scheduling to order and time the actor execution. In previous works [13, 14], we have introduced run-time actor mapping/scheduling strategies dedicated to DPN-based actors. Our scheduling strategies execute the current actor until it cannot fire anymore to exploit spatial and temporal locality. We assume that an actor should not be fired indefinitely without external contribution (other actors that consume/produce the tokens). So, the actor currently scheduled will be blocked at some point, with no chance to be fired anymore, and will exit from the action scheduler to let the actor scheduler decide the next actor to schedule.

```c
void Transpose4x4_0_scheduler() {
    while (1) {
        if (hasTokens(fifo_Src, 16) && isSchedulable_transp) {
            if (hasRooms(fifo_Dst, 16)) {
                goto finished;
            } else {
                transp(); // Fire the action
            } else {
                transp_aligned(); // Fire the action
            }
        } else { // Check the next action...
            goto finished;
        }
    }
    finished:
    return; // Return to actor scheduler
}
```

Listing 4. Action scheduler

To conclude, the execution of DPN-based programs involves both actor scheduling and action scheduling. While they are two distinct levels of scheduling, they are intimately related since the success of the action scheduling within an actor is directly dependent on the production/consumption performed by its predecessors/successors.

4. RESULTS

This section studies the implementation of dynamic dataflow programs on both desktop and embedded multi-core platforms. On the one hand, the desktop implementation is generated by use of the C back-end of Orcc [4]. The generated C code is compiled with GCC and executed on top of Ubuntu GNU/Linux. Concerning the platform that has been used during these experiments, we use an Intel Core i7 with 2 cores clocked at 3.2GHz. On the other hand, the embedded implementation targets multi-core platforms composed of homogeneous Very Long Instruction Word-style processors, based on the Transport-Trigger Architecture (TTA) [15], running at 100MHz and interconnected by point-to-point shared memories. In this configuration, the tested software implementations are generated by use of the TTA back-end of Orcc [16], then the generated code is compiled and simulated thanks to the TTA-based Co-design Environment (TCE) [17].

Table 1 summarizes the decoding frame-rates obtained from different implementations of DPN-based video decoders. All the results have been obtained with the same application descriptions (standardized) and video sequences \((\text{foreman QCIF})\). The results clearly show that our implementation significantly improves the performance thanks to our advanced software synthesis techniques.

<table>
<thead>
<tr>
<th>Description</th>
<th>Desktop [18]</th>
<th>Embedded [16]</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPEG-4 SP</td>
<td>12</td>
<td>90</td>
</tr>
<tr>
<td>MPEG-4 AVC</td>
<td>N/A</td>
<td>60</td>
</tr>
</tbody>
</table>

Table 1. Improvement of more than 200% of the decoding frame-rates (QCIF) over previously proposed implementations [18, 11, 16]

Table 2 presents the decoding frame-rates obtained from our implementation on desktop platforms with high definition video sequence. The results show that our implementation can already achieve real-time decoding frame-rates even on a small number of processor cores.

<table>
<thead>
<tr>
<th>Description</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPEG-4 SP</td>
<td>30</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>HEVC</td>
<td>12</td>
<td>22</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Real-time HD decoding frame-rates (720P) on desktop multi-core platforms

5. CONCLUSION

We have proposed advanced software synthesis techniques to enhance the implementation of dynamic dataflow programs on both desktop and embedded processors. We have particularly focused on communication and computation issues. Our approach is validated by presenting real-time decoding frame-rates of HD video sequences.

6. REFERENCES


