Interfacing and scheduling legacy code within the Canals framework

Andreas Dahlin, Fareed Jokhio, Johan Lilius, Jérôme Gorin, Mickaël Raulet

INTERFACING AND SCHEDULING LEGACY CODE WITHIN THE CANALS FRAMEWORK

Andreas Dahlin, Fareed Jokhio, Johan Lilius

Turku Centre for Computer Science
Department of Information Technologies
Åbo Akademi University, Turku, Finland
{andalin, fjokhio, jolilius}@abo.fi

Jérôme Gorin, Mickaël Raulet

IETR/Image Group Laboratory
INSA Rennes, Rennes, France
{jgorin, mraulet}@insa-rennes.fr

ABSTRACT

The need for understanding how to distribute computations across multiple cores, have obviously increased in the multi-core era. Scheduling the functional blocks of an application for concurrent execution requires not only a good understanding of data dependencies, but also a structured way to describe the intended scheduling. In this paper we describe how the Canals language and its scheduling framework can be used for the purpose of scheduling and executing legacy code. Additionally a set of translation guidelines for translating RVC-CAL applications into Canals are presented. The proposed approaches are applied to an existing MPEG-4 Simple Profile decoder for evaluation purposes. The inverse discrete cosine transform (IDCT) is accelerated by the means of OpenCL.

1. INTRODUCTION

The ever increasing complexity in software requires more computational power. Since increasing the clock frequency of a single core is not a way forward any more[1], especially for embedded systems where cooling and battery life often are critical aspects, going multi-core has become the dominant approach. The transition to multi-core started in the desktop segment and is now also clearly visible in the embedded systems domain. A vast amount of existing software has been written with sequential execution on a single core in mind. Unfortunately platform-dependent software optimized for sequential execution cannot easily make use of the additional computational power provided by additional cores. Synchronization of software executing simultaneously on several cores requires a more structured approach to scheduling than what is present in most presently available software implementations. Completely rewriting all software would be the best solution, but since impossible in practice there clearly is a need to deal with legacy code. As an example, video coding software is fairly complex because modern video standards, including MPEG-4[2], allow a bitstream to be encoded in a variety of ways. It is a tedious task to implement a full decoder/encoder and since a number of video decoder/encoder implementations already exist it would be beneficial to use those as a starting point and only rewrite selected parts in a language that provides proper support for scheduling applications on multi-core platforms. The parts of most interest are those in which most time is spent, the bottlenecks. Bottlenecks that could be eliminated by performing computations in parallel are of special interest. Such bottlenecks can be found, for instance, by profiling and analyzing how the data flows through the application.

Our analysis[3] of a particular RVC-CAL implementation of a MPEG-4 Simple Profile decoder shows that one third of the computation time is spent on overheads associated with scheduling, e.g. the IDCT component spends 73% performing real computations and 27% is consumed by overheads. Clearly there is a need for better scheduling approaches than the simple round-robin scheduler used in this decoder. To be able to elaborate with different scheduling strategies it is beneficial to separate scheduling decisions from the computational code. For this purpose the Canals Scheduling Framework is suitable. Benefits of using Canals for legacy applications are improved performance and increased ability to adapt to changes in the execution environment. The contributions of this paper are:

- Presentation of a way to interface and schedule legacy code using the Canals framework.
- A code translation approach for critical parts of the system where interfacing legacy code is not sufficient. Translation enables full Canals support for scheduling, mapping and code generation.
- OpenCL interfacing/code generation by Canals for hardware acceleration purposes.

The paper is structured as follows. Background information is provided in section 2. Section 3 focus on scheduling and interfacing legacy code from Canals, while we in section 4 present guidelines for translating applications written in RVC-CAL to the Canals language. A case study, conducted on a MPEG-4 Simple Profile Decoder demonstrates how the approaches described in the two previous sections can be applied in practice, is presented in section 5. Finally, in section 6, we conclude the paper.

2. BACKGROUND

2.1. Canals

In this section we provide brief background on the Canals language [4]. Canals aims to facilitate code generation for heterogeneous platforms, primarily in the data flow driven application domain. Thus, one of the goals is to be able to explicitly express data flows making analysis and efficient code generation possible. Canals does also provide fine-grained scheduling and execution of a program. A Canals program describes the intended behaviour of the program in a platform independent manner. All elements exist in parallel and are capable of performing computations concurrently from a resource point of view. Only data dependencies restrict the parallelism. The completely concurrent behaviour can be restricted in the compilation process by supplying a mapping and an architecture description to the compiler.

Canals is based on the concept of nodes (*kernels* and *networks*) and links (*channels*), which connect nodes together. Computations are performed in the nodes and the links are representing intermediate buffers in which output data from nodes is stored before the data is consumed by the next node. We have included expressive data type descriptions in Canals, since they are essential for understanding the precise behaviour of a data flow driven application.

Kernels are the fundamental computational element in Canals. All major computations in Canals are carried out inside kernels. Computations are performed on data available on the incoming data port and the results are written to the outgoing data port. All declared variables are local: they are only accessible from the kernel in which they are declared. Variable values are stored between invocations of a kernel, implying that kernels have a state. Computations (data processing) are performed in the *work block* using the sequential *Canals Kernel Language*. Kernels are also the mechanism through which communication between the Canals program and the external environment is handled. Reading input data from a file, a network stream or a pipe as well as writing output data are all examples of this kind of external communication with the environment.

The channel is an abstraction of an inter-kernel memory buffer, used for storing data produced and consumed by two connected kernels. A channel must specify the type of data the channel can hold, while other optional channel restrictions, for instance channel capacity, are specified in the channel definition body. Canals has one predefined channel type; *generic_channel*, which is an unbounded FIFO queue that can hold elements of any defined data type. The task of distributing data and selecting appropriate data paths is essential in data flow based approaches. *Scatter* and *gather* are the Canals elements responsible of this. A scatter is responsible for distributing data from one input channel to several output channels. The policy for distributing the data and the amount of data distributed to each channel is specified as attributes in the scatter body. Gathering parallel data flows is possible using the gather element. Definition of a gather is similar to the definition of a scatter. Scatter and gather elements can act as switches between data paths rather than distributing data on all paths. Kernel, scatter, gather and channel are all basic elements in the language. In order to be able to group a number of these elements into a larger functional module we need a container, in Canals denoted *network*. All defined elements, including networks, can be added to a network. It is also in the network the elements are connected together to build a larger functional unit. Elements in a network are connected together with the *connect* statement.

Canals is a language that does not implement a single model of computation (MoC) or a set of predefined MoCs, but instead the computational model is possible to express in the language itself. Furthermore, each network can execute using its own model of computation, rather than relying on one central MoC for the entire program. In Canals, scheduling is concerned with the task of planning the execution of a Canals network, considering data flow as well as resource use. To be able to reason about scheduling considering both of these, they are handled by separate elements in Canals. The *scheduler* is responsible for planning the execution of kernels, in such a way that data available as input to the network is routed correctly through the network and eventually becomes available as output from the network. The scheduler can accomplish correct routing by inspecting data, obviously at run-time, arriving to the network and make routing decisions based on the contents of the data. The list of kernels that must be executed in order to actually move data according to the calculated route is denoted a *schedule*. Triggering of kernels is the task of the *dispatcher*. The dispatcher should strive to execute the schedule in an optimal order regarding available processing resources.

2.2. CAL

The CAL Actor Language (CAL) [5] is a Domain-Specific Language which is especially designed to provide a concise and high-level description of actors. RVC-CAL is a subset of CAL normalized in MPEG RVC [6] as the reference programming language for describing coding tools in MPEG-C [7]. RVC-CAL, compared with the CAL, restricts the data types, and operators that cannot be easily implemented onto the platforms. Figure 1 shows an example of an actor describe in RVC-CAL that computes the absolute value of token received on the input port I to the output port O.

An actor contains one or several *actions*. An action is the only entry point of an actor that may read tokens from input ports, compute data, change state of the actor and write tokens to output ports. The body of an action is executed as an imperative function with local variables and imperative statements. When an actor fires, a single action is selected among others according to the number and the values of tokens available on

```
actor Abs () int I ==> uint O :
        pos: action I: [u] ==> O:[u]
             end
        neg: action I :[u] ==> O:[-u]
                guard u < 0
             end
        priority neg > pos; end
end
```

Fig. 1. CAL Actor for computation of absolute value.

ports and the current state of the actor. The guard conditions specify additional firing conditions, where the action firing depends on the values of input tokens or the current state of the actor. Action selection may be further constrained using a *Finite State Machine* and *priority* inequalities to impose a partial order among action tags.

A composition of RVC-CAL actors forms an RVC-CAL network. In an RVC-CAL network, actors communicate via unbounded channels by reading and writing tokens from and to FIFOs. At a network level, each actor works concurrently, executing their own sequential operations. RVC-CAL actors are only driven by token availability. An actor can fire simultaneously regardless of the environment, allowing the application to be easily distributed over different processing elements. This feature is particularly useful in the context of multi-core platforms. An important point of the RVC-CAL representation is that an actor is not specified in a specific execution model. RVC-CAL is expressive enough to specify a wide range of programs that follow a variety of dataflow models, trading between expressiveness and analyzability.

2.3. OpenCL

The Open Computing Language[8], more commonly known as OpenCL, is a royalty-free specification for general purpose parallel programming tasks in heterogeneous systems. The specification is a platform-independent interface, but implementations are targeted towards a specific platform and vendor. OpenCL is based on the concepts of a platform model, an execution model and a memory model. The platform model, an abstraction of the underlying hardware, consists of a host and one or more devices. In practice this often today means that the CPU is the host and the GPU is a device. The memory model specifies the memory hierarchy of the device. Code executing on a device is called kernels. Kernels are written in a restricted version of C99, and can be either binary precompiled or source compiled by the host at runtime. Runtime (located on host) delegates tasks to the devices for execution. OpenCL supports data parallel and task parallel programming models. Command queues coordinate the execution of OpenCL kernels in a variety ways including in-order and out-of-order execution. All data passed between kernels should be encapsulated as memory objects, thus enforcing the OpenCL programmer to consider the data flow aspects of his program. The benefits from accelerating embarrassingly parallel sections, which operate on large amounts of data, in an application using OpenCL are significant, while parallel sections with inter-dependencies and small data sets gain less.

3. INTERFACING LEGACY CODE FROM CANALS

In this section an approach for scheduling legacy code in Canals is presented. As discussed in the introduction, a large quantity of properly working software exists. The software might require partial redesign to benefit from multi-core processors, a design and programming effort that still is reasonable compared to rewriting the entire application. If critical parts and blocks of legacy code can be scheduled, mapped and executed within the same framework, the application can benefit from multi-core with a relatively low effort.

Interfacing legacy code in this case means that the main function of the application is generated by the Canals compiler. A native Canals application can, a bit simplified, be seen as a collection of kernels performing computations and exchanging computational results over channels. Wrapping source or binary code into what we denote *external kernels*, gives the Canals Scheduling Framework access to them as if they were normal kernels. External kernels can share data over *external channels* or normal Canals channels depending on the interface specification. The use of external channels is an easy starting point when data transfers are of no or little interest, since external channels essentially hide data transfers from Canals. From this follows that scheduling of memory transfers is not possible if external channels are used. An external kernel is defined similarly to a normal kernel. The body of an external kernel can contain the following backend dependant attributes for specifying the interface between the kernel and Canals:

type External kernel type can be source or binary.

files Files necessary for proper access to the kernel.

inithandle A handle to initialization funtionality that should be invoked before first kernel execution.

workhandle A handle to the normal execution routine.

enabledhandle A handle to functionality that can decide if the kernel is eligible for execution.

inputhandle, outputhandle Specifying data access.

getamounthandle, putamounthandle Provides a handle to the current consunmption/production rate of the kernel.

supportsCanalsIO Specify if the kernel use the Canals channel API (get, put, look).

An example of a defintion compatible with the C++ backend:

```
external kernel bit -> bit get * put * IDCT2d {
    type = binary;
    files = "common_constants.h, idct.o";
    workhandle = "extern(C):void idct2d(); idct2d();";
    enabledhandle = "extern(C): bool idct2d_is_enabled();
                     idct2d_is_enabled();";
}
```

Now that the Canals compiler is aware of the legacy code block through the external kernel mechanism, but we still need to gather scheduling information on these kernels. Additionally we must select a dispatching strategy.

3.1. Scheduling and Dispatching

Scheduling legacy code means that we must extract information about the components and their interconnection, and make that information available to the Canals scheduler. A Canals scheduler contains at least one kernel, in which the schedule computations take place. The scheduling code is written in the Canals kernel language according to the intended scheduling strategy. To be able to make scheduling decisions the scheduler must be able to access information about the network it is scheduling, also during run-time. For this purpose any kernel inside a scheduler can navigate the scheduled network through the **Run-Time Network Navigation API**. The API implements functionality for retrieving the first element, last element, next element and previous element. Additionally element information queries regarding e.g. consumption/production rates, element type and number of output ports from a scatter element can be made. The same API is used for scheduling native as well as external kernels.

The API is supported by one network topology matrix per network. The matrix is a lower-triangular matrix containing all static information about the network we schedule and information on how to retrieve correct information for variable values, such as dynamic production rate or channel size. This compact representation of network topology and element information can be placed in a memory close to the processing unit on which the scheduler is mapped, thus ensuring low communication overhead for most scheduler queries. Multiple inputs/outputs are possible for external kernels. In the Canals network they are however only connected by one ingoing and one outgoing external channel, which represent the input/output port most relevant for scheduling purposes. The purpose of the connection is to provide information needed for building the topology table. The order of elements in the topology table is the same as in the connection order.

It is also possible to build the topology matrix based on other information than the information which can be extracted from the Canals network description. For rapid prototyping, the topology table can even be written manually in the target language (C++). In case a Canals network only contains elements written in RVC-CAL, the topology table can be derived from information provided by guards and consumption/production information available in each actor. A Canals scheduler can be generated based on information from guards, scheduling FSM and priorities. Run-time computations in the scheduler can be further reduced by analyzing the topology table and the scheduler at compile-time. In Canals the role of the dispatcher is to execute the schedules produced by the scheduler, through the Hardware Abstraction Layer (HAL). The dispatcher should strive towards optimal utilization of computational resources by rearranging tasks from one or several schedules. Sorting of tasks must not break data dependencies imposed by the scheduler. There is a default dispatcher that always runs a schedule to completion before starting the dispatching of the next schedule. The same rules apply for dispatching external kernels.

3.2. Interfacing OpenCL Kernels from Canals

It is possible to interface OpenCL kernels from Canals through the external kernel construct. The OpenCL framework (OpenCL API + execution model) can act as an interface between Canals and CAL. A CAL actor could, for instance, be compiled into an OpenCL kernel by the CAL compiler, and scheduled by Canals. The above mentioned interfacing and scheduling mechanism for external kernels applies for OpenCL kernels as well. Both the required OpenCL host and device code must be embedded into the function given to Canals as *work handle*. All initialization required should take place in the function specified as the *init handle*.

However, it should be mentioned that some properties of OpenCL and its run-time system mismatch with this simple interfacing approach, resulting in degraded performance and limited flexibility. The number of OpenCL kernel instances intended for concurrent execution is limited by available input data and hardware limitations (e.g. number of threads on the GPU). This control information must be completely handled by the external code, or alternatively be sent as control flow.

4. TRANSLATING RVC-CAL INTO CANALS

The interfacing mechanism described in the previous section enables scheduling of legacy functionality expressed in any supported language. In this section, another solution for scheduling code expressed in other languages is discussed, namely the possibility to translate existing source code into Canals. On the one hand, translation into Canals makes it possible to use all features of our scheduling framework, but on the other hand the required translation effort can often be larger than the effort for interfacing legacy code. The possible degree of automation is highly depending on the semantics of the source language and the software design philosophy applied. Since Canals is a streaming language, software that has been designed with data flow in mind is most suitable for translation into Canals. In this work we have studied how, and to what degree, translation from RVC-CAL (see section 2.2) can be carried out. General guidelines (see table 1) for such a translation are presented together with a concrete example on translation of the two-dimensional inverse cosine transform into Canals. RVC-CAL and Canals are quite similar in some aspects; both languages are based on the concept of computational nodes and links that connect these nodes. Each computational node consumes and produces

data. Differences between the languages relevant from a translation viewpoint are: number of input/output links from each node, data locality and scheduling. In Canals, two nodes (kernels) can only be connected by a single link while CAL actors can be connected by an arbitrary number of links. An approach for resolving this incompatibility is to group data from several links together into a larger Canals data structure. In RVC-CAL scheduling can be divided into *intra-actor* scheduling, scheduling execution of actions, and *inter-actor* scheduling which decides on the execution order among actors. Inter-actor scheduling is not decided on explicitly by the programmer but rather by the compiler implementation and/or run-time system. In practice this means that the most intuitive translation of an actor into a Canals kernel is not possible for all valid actors, since Canals schedulers are only associated with networks and not kernels. Therefore a CAL actor should in the general case be translated into a Canals network and an action translates into a kernel. In RVC-CAL, all variables are declared in the actor and shared by all actions. In Canals variables are private to the kernel, implying that two kernels cannot exchange data through a shared variable, which means that the shared variables must be modelled as data flow. The construction in figure 2a shows how a combination of scatter (switch) and gather (select) elements guarantees shared access to data among actions. Scheduler S decides on the action execution order based on information translated from guards, the finite state machine and priorities in an actor. Further, S directs data to the action to be executed next by altering the switch position for both scatter and gather elements through the means of control flow. D_{IN} holds the data available on all actor input ports, while D_{OUT} represents the data produced to all output ports. D_{SHARED} contains all variables shared between actions. Actor variables that are used only within one action can easily be identified and translated as a kernel variable. All needed data is streamed to a kernel through the data definition D, which is a composition of D_{IN} and D_{SHARED}. K forwards data from the gather element if data is available, otherwise data initialized to default value is generated. In the case multiple actors are connected together, forming a RVC-CAL network, two strategies can be applied. Either all network levels can be flattened into a single RVC-CAL network, the other option is to translate each RVC-CAL network into a Canals network. Flattening a RVC-CAL network is straightforward since the network descriptions are only a syntactic construction without any semantics. Once the RVC-CAL network is translated into Canals it is possible for the programmer to write a scheduler that decides on the execution order for actors.

The translation guideline above covers translation of a general actor or actor composition well. However, the translated Canals application introduces a large number of networks and thus run-time overheads. These overheads can be reduced by using optimizations in the Canals compiler. If semi-automated translation is a viable option, knowledge about the internal workings and data flows of an actor can be utilized to get a better translation into Canals. In figure 2b, we have an example of where an actor has been translated into a Canals kernel and the action selection is controlled by the scheduler of the containing Canals network. Such a translation is to recommend for actors where we know that the actor operates in certain modes, each mode represents a certain action firing sequence (quasi-static schedule). It is even possible to integrate the selection of mode into the kernel itself (figure 2c), increasing performance but at the same time reducing the possibilities for fine-grained scheduling. For actors where a static schedule can be calculated, it is always possible to translate them into a stand-alone kernel without use of control flow connections.

RVC-CAL	Canals
Network description	Network
Actor	Network
Actor variable	Data definition/flow
Action	Kernel
Procedure, Function	Inlined kernel code
Buffer	Channel
Guards, FSM, Priorities	Scheduler, Control flow

Table 1. RVC-CAL to Canals element mappings.

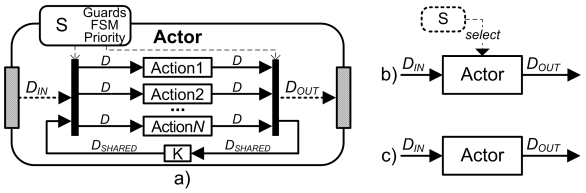

Fig. 2. Canals representations of a RVC-CAL actor

4.1. Translation of an IDCT-2D Actor

To illustrate how translation is done in practice, a CAL implementation of the inverse discrete cosine transform (IDCT) is translated into Canals. The two-dimensional IDCT, operating on an 8x8 block, is frequently used in video and image processing. The chosen IDCT (see figure 3) is implemented as a single CAL actor, with two actions. Additionally priorities, guards, functions, procedures, constants and variables are present in this design. There are two inputs and a single output. The dominating input is the coded coefficients and the output is the decoded coefficients, an additional input represents a single $SIGNED$ boolean value for the entire block. The $SIGNED$ flag is a control token that affects the clipping functionality of the $IDCT$, which in practice means that it decides which action is to be fired when the actor is executed. Both actions will consume 64 values from actor port IN, a single value from actor port $SIGNED$ and produce 64 values to port OUT. In figure 4, the Canals translation based on the previously given guidelines is given. Some kernel code

```
actor Algo_IDCT2D_ISOIEC_23002_1 ()
 int(size=13) IN, bool SIGNED ==> int(size=9) OUT :

 int A=1024;int B=1138;...int J=2528;
 List(type:int, size=64) scale = [A,B,...,E];

 intra: action IN:[ x ] repeat 64, SIGNED:[ s ]
           ==> OUT:[ block1 ] repeat 64
 var
   List(type:int,size=64) block1, block2
 do
   // multiplier-based scaling
   block1 := [scale[n] * x[n] : for int n in 0..63];
   block1[0] := block1[0] + lshift(1, 12);
   // scaled 1D IDCT for rows and columns
   idct1d(block1, block2);
   ...
   // clipping
   block1 :=[clip(block1[n], 0) :for int n in 0..63];
 end

 inter: action IN:[ x ] repeat 64, SIGNED:[ s ]
          ==> OUT:[ block1 ] repeat 64
 guard s
 var List(...) block1, List(...) block2
 do
   ...
   block1:=[clip(block1[n],-255):for int n in 0..63];
 end

 procedure idct1d ... end
 function clip(int x, int lim) --> int ... end
 priority inter > intra; end
end
```

Fig. 3. Code listing for an IDCT CAL actor.

have been omitted, since the translation of action code into Canals kernel code is fairly straightforward and not the purpose of this study. Since the actions in this case only share access to constant data and does not access any shared variable, the feedback loop construction is not required.

If we apply knowledge about the functionality (domain and language expertise) this actor provides and its computations, a better translation can be made (see figure 5). Here the actor has been translated directly into a Canals kernel, since the action selection can be decided on completely by the kernel itself. Examining the original actions reveals that large portions of the actions could be merged together; only leaving a decision point before clipping is applied.

5. CASE STUDY - A MPEG-4 SP DECODER

In this section we demonstrate how the interfacing approach (see section 3) can be used to implement a improved scheduling strategy for an MPEG-4 Simple Profile Decoder[9] written in RVC-CAL. Further, hardware acceleration through OpenCL is enabled for a part of the decoder.

5.1. Analyzing and Profiling the Decoder

The idea in this case study is to optimize the performance of the inverse discrete cosine transform (IDCT) component, but before we start the process of interfacing and scheduling the MPEG-4 SP decoder from Canals, we must assure that we have thorough understanding of it. Therefore we analyze the RVC-CAL design carefully to get an understanding of the in-

```
network IN_SIGNED -> OUT Algo_IDCT2D_ISOIEC_23002 {
  constant int32 A=1024; ... constant int32 J=2528;
  constant int32[64] scale = [A, B, C, ..., E];
  set_scheduler S_Algo_IDCT2D_ISOIEC_23002;
  set_dispatcher default;

  add_scatter sc<ActionSwitch>;
  add_gather ga<ActionSelect>;
  add_kernel intra<ActionIntra>;
  add_kernel inter<ActionInter>;
  connect NETWORK_IN -> sc;
  connect sc.outport[1] -> intra -> ga.inport[1];
  connect sc.outport[2] -> inter -> ga.inport[2];
  connect ga -> NETWORK_OUT;
}

scheduler IN_SIGNED->Schedule S_Algo_IDCT2D_ISOIEC {
  set_scheduler default;
  add_kernel k<S_Algo_ComputeSchedule>;
  connect SCHEDULER_IN -> k -> SCHEDULER_OUT;
}

kernel IN_SIGNED -> Schedule S_Algo_ComputeSchedule {
   work look 1 put 1 { /* Scheduling code */ }
}

kernel IN_SIGNED -> OUT ActionIntra {
  variable IN_SIGNED input;
  variable OUT output;
  variable int13[64] x;
  variable int32[64] block1;
  variable int32[64] block2;
  variable int32 n;

  work get 1 put 1 {
    input = get();
    x = input.IN;
    for(n=0; n<64; n++){block1[n] = scale[n] * x[n];}
    // inlined procedure call idct1d(block1, block2)
    for(n=0; n<64; n++){
      /*inline clip(block1[n], 0)*/
    }
    output.OUT = block1;
    put(output);
  }
}

kernel IN_SIGNED -> OUT ActionInter { ... }
scatter IN_SIGNED->[IN_SIGNED,IN_SIGNED] ActionSwitch
gather [OUT, OUT]->OUT  ActionSelect...

datadef IN_SIGNED { int13[64] IN; bool SIGNED; }
datadef OUT { int9[64] OUT; }
datadef int32 (type="integer") { bit[32]; }
datadef int13 (type="integer") { bit[13]; }
datadef int9  (type="integer") { bit[9];  }
datadef bool  (type="boolean") { bit[1];  }
```

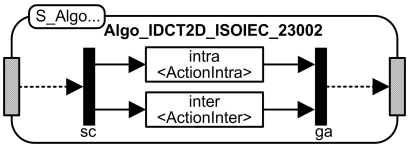

Fig. 4. IDCT actor expressed in Canals.

```
kernel DCTCodedBlock->Block Algo_IDCT2D_ISOIEC_23002{
  constant int32[64] scale=[1024,1138,1730,...,1264];
  variable DCTCodedBlock input;
  variable Block output;

  work get 1 put 1 {
    input = get();
    if (input.SIGNED) { // Code for inter action }
     else { // Code for intra action }
    put(output);
  }
}
datadef DCTCodedBlock { int13[64] IN; bool SIGNED; }
datadef Block { int9[64] OUT; }
```

Fig. 5. A manual translation of the IDCT actor into Canals.

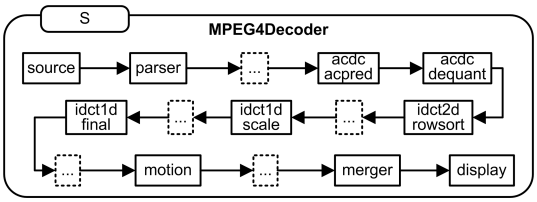

Fig. 6. Decoder based on actors (as external kernels).

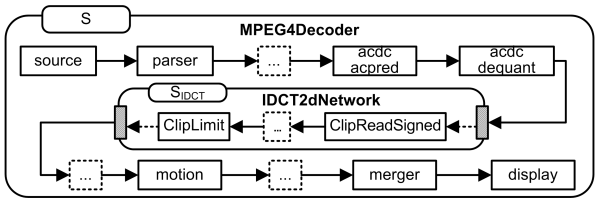

Fig. 7. Decoder based on actors and actions.

teraction between components. In this particular decoder the two-dimensional IDCT is described at a very detailed level with a large number of actors and actions. The next step is to verify that there actually is large overhead in scheduling the IDCT, overhead that can be eliminated by improved scheduling. Instrumentation and profiling the C-code generated from RVC-CAL by CAL2C for four video sequences give us that the default scheduling strategy is unsatisfactory. The decoder consists of 39 CAL actors that are scheduled and executed in a simple round-robin order. The scheduler function of an actor is called upon from a while loop in the main function. The scheduler function first checks if it is possible to execute any of the actions in the scheduler by checking the input and output requirements for each action and not violating rules imposed by the actor FSM. When no action can be executed anymore, the action scheduler will return and the main loop will call upon the scheduler function for the next actor in the list of actors. We are interested in the scheduling overhead, which is here defined as the time spent checking if an actor can execute (by checking the action guards) or not. For different actions the time consumed in guard conditions checking is different. If any guard condition is checked then there can be multiples checks for it. It can fail at the very first check means there is no need to check further conditions or it may fail at the very last check. The hardware used in this case study is a Desktop Computer equipped with an Intel Core 2 Duo at 2.66 GHz. The RAM was 2 GB and 32-bit windows 7 operating system. The number of checks was calculated for the MPEG-4 decoder with the visual studio profiler instrumentation tool. Detailed results from the analysis is available in [3].

5.1.1. Canals Scheduling Code Generated from CAL Actors

The first step in the case study is to verify that the decoder is running properly when executed through Canals, using the same scheduling as in the original code. This means that for each of the 39 actors a Canals external kernel is defined. In this case we use binary linking against the C object files generated by the CAL2C compiler and provide the scheduler function (which deals with both action scheduling and action execution) as the workhandle. Some of the actors requires a certain initialization routine to be run before their first execution; for this purpose the inithandle is used. All defined kernels are added to the top-level network and connected together by external channels, as can be seen in figure 6. The connection order is in this case not of great importance, but it will decide on the order elements will be placed in the network topology matrix. Scheduler S creates a schedule, by iterating over the element with the getNextElement() function, containing each kernel once. The system will only terminate if a iteration count argument is provided on command line.

Since we by analysis and profiling decided to try to improve the scheduling and performance of the IDCT component, the next step is to expose that part in a more fine-grained way to the Canals Scheduler by describing it at action level (see figure 7). In this design all actions from the 12 actors, which describe the IDCT-component in the original design, have been extracted and added as 36 external kernels to a Canals sub-network. It has its own scheduler S_{IDCT}, that decides on the action execution order within the network. The top-level scheduler has been altered so that instead of scheduling the 12 actors, it will only check if the Canals network is enabled (equivalent to the scheduler of the network being enabled) and if it is enabled add the network to the schedule. Enabledness for S_{IDCT} is handled through the enabledhandle attribute, which in this case specifies that the return value of a certain C function decides on its enabledness. In this particular implementation the function will check that 64 tokens of input data and an additional token for the $SIGNED$ value is available at the input queue and that there is space for 64 output tokens on the output queue. S_{IDCT} scheduling functionality could be derived from guards of each action, but in this case we have used another better approach. The work presented in [10] shows that a static action schedule (containing 755 action firings) can be calculated for the IDCT component. This static schedule is implemented in S_{IDCT} and put on the dispatch queue when S_{IDCT} is invoked. We now have a very fine-grained control over the scheduling of the IDCT component. Scheduling and dispatching each action one-by-one in this manner through the Canals HAL is excellent for evaluation purposes, but because of the dispatching overhead this results in reduced performance compared to the original version. Another optimization step must be conducted.

Analysis of the $IDCT2dNetwork$ gives us that it is perfectly possible, because of the static schedule, to flatten the network into a single kernel (figure 8). The transformation to a kernel was manual, but this step can be automated. The transformation is also possible for quasi-static schedules. It can be observed from the instrumentation and profiling that

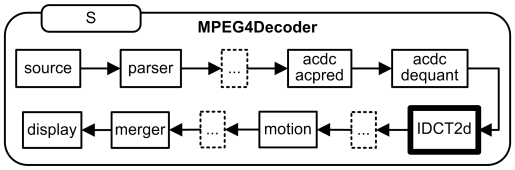

Fig. 8. Decoder with external kernel for IDCT.

for our four video sequnces there is a reduction in number of hits and misses after applying the quasi-static schedule, because we now only perform three checks per IDCT invocation. Besides the 25% speedup for the IDCT part as a result of reduction in overheads, a reduction in overheads for the other actors can be observed. The explanation for this is that the simple round-robin scheduler in the original design will try to execute actors the theoretically cannot be enabled before the IDCT has produced output. For the decoder as a whole, decoded frames per second improved by 13%.

5.1.2. Canals Scheduling a OpenCL Enabled Decoder

The IDCT operation on a block is obviously an operation that would benefit from data-parallel execution. For this purpose the external IDCT kernel in the design can be replaced with a external OpenCL IDCT kernel (figure 9).OpenCL host and device code is completely wrapped in the functions provided to Canals as init- and work-handle, including the decision on how many predefined number of parallel instances of the OpenCL kernel should be executed on the graphics processing unit (GPU). If only a small number of parallel instances are executed at once, the overhead caused by memory transfers from main to device memory will result in low performance. This is a common problem in OpenCL programming today, but future OpenCL enabled platforms, such as the Mali[11], aims to resolve the issue of memory transfers by using uniform memory for host and device.

As mentioned above, the buffer sizes and number of parallel instances are decided on in the external code block that the Canals scheduler cannot access nor modify. This is a drawback of the interfacing approach. The decoder illustrated in figure 9, uses a native Canals kernel for the IDCT. The scatter and gather elements indicate that there can be from 1 to N parallel instances, where N is restricted by mapping strategy and compiler backend, as well as by the scheduling strategy. The exact number is decided by the scheduler/dispatcher.

6. CONCLUSION

Existing software cannot directly benefit from the increased computational power available in multi-core systems. In this work we have presented two techniques for dealing with legacy code and adapting it gradually for better multi-core compatibility, by having more control over scheduling.

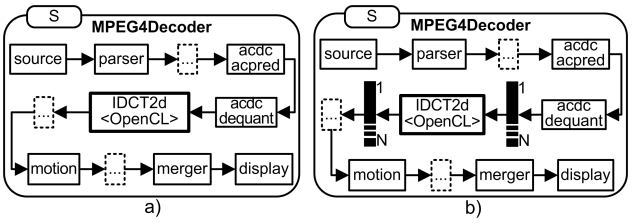

Fig. 9. OpenCL accelerated kernels in Canals.

Canals is suitable for the purpose of interfacing and scheduling general legacy code with a small effort. Complete control and more efficient code generation are provided by the translation approach for those parts of the application that are of certain interest and where a larger effort is motivated. We have also described how Canals can generate, interface and schedule OpenCL kernels. A case-study of a MPEG-4 Simple Profile decoder is used for validation purposes.

7. REFERENCES

[1] Herb Sutter and James Larus, "Software and the Concurrency Revolution," *Queue*, vol. 3, pp. 54–62, 2005.

[2] International Organization for Standardization, *ISO/IEC 14496-2:1999: Information technology — Coding of audio-visual objects — Part 2: Visual*.

[3] F. Jokhio et al., "Analysis of an RVC-CAL MPEG-4 Simple Profile Decoder," Tech. Rep. 1018, TUCS, 2011.

[4] A.Dahlin et al., "The Canals Language and its Compiler," 2009, SCOPES'09, pp. 43–52, ACM.

[5] J. Eker and J. Janneck, "CAL Language Report," Tech. Rep. ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, 2003.

[6] M. Mattavelli, I. Amer, and M. Raulet, "The Reconfigurable Video Coding Standard," *Signal Processing Magazine, IEEE*, vol. 27, no. 3, pp. 159 –167, 2010.

[7] ISO/IEC CD 23002-4:2008, "MPEG Video Technologies - Part4:Video Tool Library," .

[8] A. Munshi, "OpenCL - Introduction and Overview," January 2011, `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`.

[9] G.Roquier et al., "Automatic Software Synthesis of Dataflow Program: An MPEG-4 Simple Profile Decoder Case Study," in *SiPS'08*, pp. 281–286.

[10] J. Ersfolk et al., "Scheduling of Dynamic Dataflow Programs with Model Checking," in *SiPS'11*, 2011.

[11] A. Lokhmotov, "Mobile and Embedded Computing on Mali GPUs Presentation," 2010.