Realizing Efficient Execution of Dataflow Actors on Manycores

Essayas Gebrewahid*, Mingkun Yang*, Gustav Cedersjö+, Zain Ul-Abdin*,
Veronica Gaspes*, Jörn W. Janneck+, and Bertil Svensson*

*Centre for Research on Embedded Systems, Halmstad University, Sweden
{*essayas.gebrewahid, veronica.gaspes, bertil.svensson, zain-ul-abdin*}*@hh.se*
minyan09@student.hh.se
+Department of Computer Science, Lund University, Sweden
{*gustav, jwj*}*@cs.lth.se*

***Abstract*—Embedded DSP computing is currently shifting towards manycore architectures in order to cope with the ever growing computational demands. Actor based dataflow languages are being considered as a programming model. In this paper we present a code generator for CAL, one such dataflow language.**

We propose to use a compilation tool with two intermediate representations. We start from a machine model of the actors that provides an ordering for testing of conditions and firing of actions. We then generate an Action Execution Intermediate Representation that is closer to a sequential imperative language like C and Java. We describe our two intermediate representations and show the feasibility and portability of our approach by compiling a CAL implementation of the Two-Dimensional Inverse Discrete Cosine Transform on a general purpose processor, on the Epiphany manycore architecture and on the Ambric massively parallel processor array.

***Index Terms*—dataflow languages; compilation framework; code generation; manycore; CAL;**

I. Introduction

High performance embedded computing is now shifting towards manycore architectures in order to cope with the ever growing computational demands of DSP applications together with limited energy and power budgets. One of the most interesting engineering problems that arise in this context is targeting these architectures with application development frameworks and compilation tools. To overcome this challenge a number of parallel programming models have been proposed. This is due to the fact that traditional programming languages, like C, C++, or Java, lack high-level abstractions that can reflect the inherently parallel nature of the applications and the underlying parallel architecture. As stated by Fuller and Millett [1], "the era of sequential computing must give way to a new era in which parallelism holds the forefront".

In this new era of parallelism, actor-oriented dataflow programming has gained acceptance. A number of such languages have been proposed for developing streaming signal processing applications at a higher level of abstraction; examples include Erlang [2], SALSA [3] and E language [4]. Actor oriented models encapsulate concurrent computation in components called *actors* [5]. Each actor is an independent entity that interacts with other concurrent actors via channels. This makes an actor model naturally suitable for parallelism.

CAL is one such language that provides actor oriented abstractions independent of the underlying hardware [6]. RVC-CAL, a subset of CAL, has been adopted by MPEG and ISO as a standard to specify video coding [7]. CAL actors take computation step by firing *actions* that satisfy all the required *conditions*. These conditions depend on the input tokens and the actor's internal state. CAL provides dependencies, priorities, finite state machines and token rates to support various models of computations (MoCs) such as Synchronous Dataflow (SDF) [8], Cyclo-Static Dataflow (CSDF) [9], Kahn Process Networks (KPN) [10], and Dataflow Process Networks (DPN) [11].

Earlier work has suggested the use of *Actor Machines* (AM) [12] as an execution model for actors i.e. to schedule the testing of conditions and the execution of actions. In this paper we demonstrate that AM in fact is a good source for generating code for manycores. We do so by introducing *Action Execution Intermediate Representation* (AEIR) to bridge the gap between the abstract AM and an imperative implementation of the scheduler and the actions. We believe that AM can be used for high-level optimization such as, composing and splitting actors and for dataflow optimizations, and AEIR can be used for low-level optimizations, such as loop optimizations, dead code elimination and inlining functions.

The paper describes the transformation of AM to AEIR, and the translations and different techniques used to generate sequential and parallel code. The portability of CAL applications is demonstrated by using three different platforms. The paper also demonstrates the feasibility of our approach through a case study. We program in CAL and provide two interpretations of the application, as DPN and KPN. Using the DPN interpretation we generate sequential C code for general purpose (GP) processors and parallel C code for the Epiphany manycore architecture[13]. From the KPN interpretation we generate `aJava` and `aStruct` code for the Ambric processor array [14].

II. Background

A. The Programming Model

In general, a dataflow system is a collection of several components, called actors, connected by channels. In this work, we have used CAL Actor Language to define the actors [6] and Network Language (NL) to express the communication among the actors [15].

CAL is a domain-specific language that provides high-level abstraction for dataflow programming. CAL actors may have private variables to control the state of the actor, named input/output ports to communicate with other actors and actions to perform specific task. An actor does not have access to the state of another actor. Thus, interaction among actors happens only via input/output ports. Each actor is characterized by a step-by-step executions of actions. Each execution step may update the private state, consume tokens and/or send tokens. During execution, an actor can take different *actions* depending on (1) the availability of tokens on the input port, (2) the actual values of the tokens and (3) the internal state of the actor. These conditions are called the *firing conditions* of an action. In addition, a CAL actor may also have an *action scheduler* to order firing of actions and/or *action priorities* to select an action with highest priority if there is more than one eligible action. Listing 1 shows an actor with two actions that produce tokens on different output port depending on the value of the token read from the input port.

```
1  actor Split ( ) A ==>  P, N:
2    A1 : action A: [ v ] ==> P: [ v ]
3       guard v >= 0
4       end
5    A2 : action A: [ v ] ==> N: [ v ]
6       guard v < 0
7       end
8  end
```

Listing 1. Split: A simple CAL actor.

The interconnection between actors is expressed using NL. NL has three sections a *variable declaration section* to define variables that are used as attributes for actors and sub-networks, an *entity section* to declare actors or sub-networks, and a *structure section* to express channels that sketches the dataflow network. In NL a programmer can add additional information, like FIFO size, via annotations. Some manycore architectures provide hardware channels, but usually a channel is implemented as a stream of packets or a buffered FIFO.

B. Target Platform

1) Epiphany: Epiphany [13] is a power-efficient manycore architecture constructed by 2D array of nodes that are connected by a low-latency mesh network-on-chip. Each node consists of a floating-point RISC processor that supports ANSI C, 32KB local memory which is part of a distributed shared memory system, network interface and DMA engine. The network consists of three parallel networks which are used individually for writing on-chip, writing off-chip, and all reading requests, respectively. Due to the differences between the networks, writes are approximately 16 times faster than reads for on-chip transactions. The transactions are done by using dimension-order routing (X-Y routing), which means that the data first travels along the row and then along the column. The DMA engine is able to generate a double-word transaction on every clock cycle and has its own dedicated 64-bit port to the local memory. The Epiphany memory system uses a flat address space visible by every mesh node, and supports up to 4096 mesh nodes. Even though all the internal memory of each core is mapped to the global address space, the cost of accessing individual address space is not uniform, as it depends on the number of hops and contention in the mesh network.

2) Ambric: Ambric [14] is an array of globally asynchronous locally synchronous brics. Each bric is composed of two pairs of Compute Unit and RAM Unit. The Compute Unit consists of four 32-bit RISC processors, two SR (Streaming RISC) processors and two SRD (Streaming RISC processors with DSP extensions). The RAM Unit consists of four independent banks of RAM with 512 words.

Ambric is designed to support KPN [10] with bounded buffer execution. Ambric is programmed using structured object programming model. Each process performs a sequential execution an object programmed in an assembly language or in `aJava` which is a subset of Java language. The structure of each object and the communication among objects is defined using `aStruct` language. Each object communicates via a unidirectional, point-to-point, asynchronous and single word width channel. `aJava` does not support some features of standard Java e.g, *continue*, *switch case* statement and *foreach* statement.

III. Compilation Methodology for CAL

To address portability, productivity and to provide a common compilation tool for various manycore architectures, we propose to start with CAL Actor Language, and use a compilation tool with two IRs. In the compilation process each CAL actor is translated to an AM that is then translated to AEIR. Finally, the AEIR is used to generate code for the target platform. We expect that these IRs, offer support for various dataflow languages that can model DPN and manycore architectures that are suitable for DSP applications. Fig 1 sketches our framework that uses AM and AEIR to generate efficient sequential execution of actions.

A. Actor Machines

An *actor machine* (AM) is an abstract machine model for dataflow actors, which is explicit about how to test the firing conditions of the actions [12]. An AM is a controller with a set of states made from all firing conditions of actions together with an action scheduler and action priorities. Each state in the controller has knowledge about the conditions of the actor, and a set of instructions that can be performed on the state in order to proceed to the next state. An instruction can be a *test* on one of the actor's conditions (guard or availability of token on input port), an *exec* for the execution of an action, or a *wait* to remove knowledge about the test on availability of tokens.

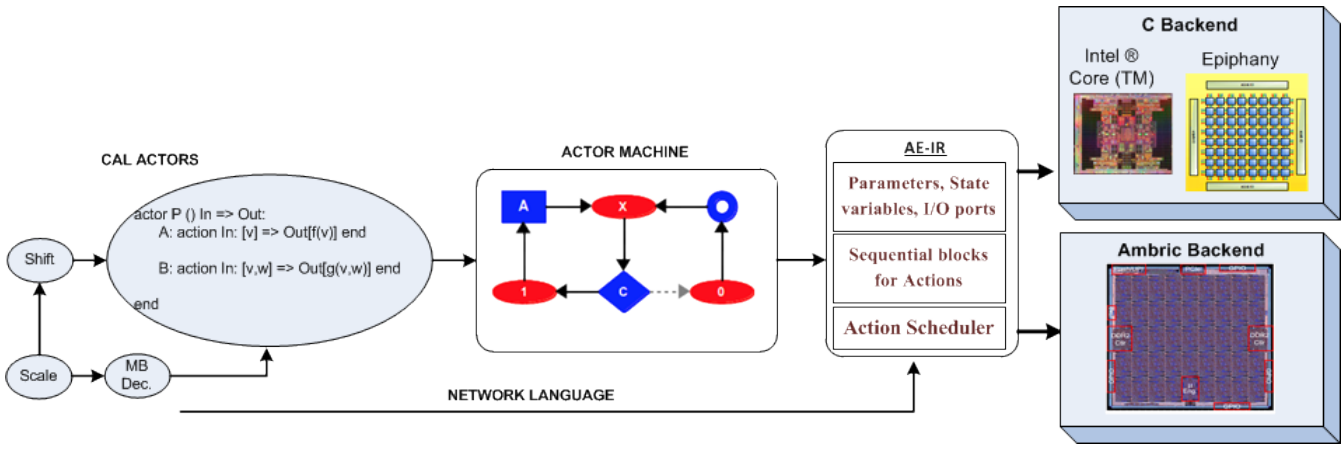

Fig. 1. CAL-AM compilation framework.

The goal of the AM is to speed up the process of selecting an appropriate action. There are several ways of translating an actor to an AM, like Round Robin and Memorize Tests [16]. Round Robin tests all the conditions of an action at once. With this translator conditions that are common among actions can be tested repeatedly, even if they have the same value. For example, in the split actor of Listing 1, since both actions depend on the availability of a token on port A, this condition has to be tested twice before action A2 is fired. Memorize Tests avoids this problem by memorizing earlier test results. With Memorize Tests the AM states represent information about conditions of the action: **X** if the condition has not been tested, **1** if the test is true, and **0** if the test is false. Fig. 2 shows the AM for the split actor from Listing 1, where

- states are represented by ellipses containing values of the three conditons, i.e., a token in port A, guard of action A1 and guard of action A2
- AM instructions are represented by diamond (*test*), rectangle (*exec*) and annular (*wait*)

The compiler translates CAL actors to AMs with the Memorize Tests translator. The Memorize Tests starts from the initial state where all the conditions are unknown. Next, for eas instrucion, it adds a possible instruction and a destination state. Finally, it repeats the process for the added states. On the current state, if an action is eligible to be fired based on the knowledge of the state, the translator adds an *exec* instruction. The conditions in the destination state of the *exec* instruction are set to unknown. If there is no eligible action and if there are

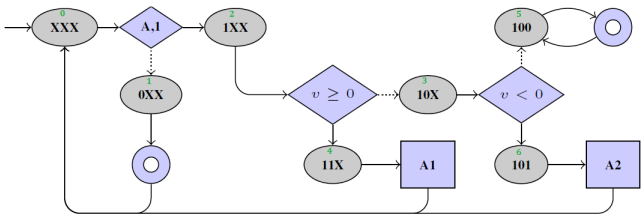

Fig. 2. Actor machine for Split actor.

actions that need additional testing, the translator adds a *test* instruction. A *test* instruction has two destinations that reflect the information gained by the test. If no instruction is added and if the current state is the destination of a failed input test, then a wait instruction will be added. The destination of wait sets the information about absence of tokens to unknown.

In addition to the set of controller states, an AM also has

- a list of I/O ports to interconnect with other AMs,
- an internal state that can be updated during the execution of an action,
- a list of parameters to store values passed to the actor from NL, and
- functions and procedures to bundle reusable code segments.

B. Action Execution Intermediate Representation

An actor machine consists of AM states that have knowledge about conditions and a set of AM instructions. These have to be transformed to different programming language constructs, such as function calls to execute the actions, *if* statements to test the conditions and flow control structures to traverse from the current AM state to the destination state. These constructs have different implementations in different programming languages and platforms. Thus, we have chosen to introduce an IR that brings us closer to a sequential action scheduler without having to choose a target language. We call this IR the Action Execution IR (AEIR) a low-level representation for actor oriented languages. AEIR keeps the implicit parallelism which makes it suitable to recognize both sequential and parallel optimization possibilities. AEIR includes constructs for expressions, statements, function declarations and function calls.

Currently, we plan to exploit the parallelism between actors. To allow different implementations for the communication among actors, AEIR includes communication primitives such as operations for testing input/output ports and sending/receiving tokens. Given that different hardware platforms have very different communication mechanisms, it is important to keep these primitives available until the very last phases of

code generation. As can be seen in Section IV-C some of the primitives are translated differently for the three different implementations.

After performing some optimizations on the AM, such as translating the AM to a *single instruction actor machine*, i.e. one that has at most one instruction in each state [16], the AM is translated to AEIR. Transitions of the AM, actions of the CAL actor, are converted to functions, where

- action variables are translated to local variables
- input/output ports are converted to parameters
- input pattern to local variable declarations initialized by *ConsumeToken* statements
- output patterns to *SendToken* statements situated at the end of the function execution, and
- the body of the action becomes the body of the function.

However, the guards of the action are not incorporated in this function, because they are evaluated before the action is fired, in accordance with the schedule provided by the AM.

Furthermore, a function is introduced as an *action_scheduler* to implement the AM. This function has parameters driven from the Input/Output pattern of the AM. Its body is made up of statements translated from the nodes of the AM. Additionally, the scheduler needs a scheme to traverse from state to state.

AM *test* nodes are translated to if-statements. While translating tests on predicate conditions, if the expression uses variables other than the state variables, then the required variables are initialized before the predicate is tested.

AM *exec* nodes are translated to function calls to the function that implements the action to be executed.

The translation of a *wait* node depends on the target platform and on the number of actors mapped on a single core. When there is more than one actor on a single core, then *wait* is translated to a return statement to give control back to the local actor scheduler. Otherwise, if there is one actor per core, *wait* can be translated as block, deactivate, pause or any other operation depending on the target architecture. Therefore, to allow different implementations of a *wait* node, AEIR introduces a primitive called *wait()*.

To traverse through the AM states, initially we have used two simple schemes. One of them labels each AM state and uses a *goto* statement to jump to the label of the destination state. The other scheme first assigns a unique number to each state, and then connects all the states using a *switch-case* or *if-else* statement that checks the state variable which stores the unique number. These two schemes can result in unreadable and unstructured code, especially for large AMs. Usually AMs have a large number of states, for example, the split AM in Fig. 2–which memorizes only three conditions– has seven states. In our case study the average number of states is 77. In addition, some architectures native tools do not allow unstructured programming, which restricts both *goto* and *switch-case* statements.

To meet these restrictions and to generate a readable code, we used a different scheme that composes states with *test* instructions until it comes to a state with either a *wait* or an *exec* instruction. This new scheme can be seen as a set of continuous test sequence broken by *wait* or *exec* instructions. When a sequence is broken, a new test sequence is generated, starting from the destination of the instruction that broke the test sequence. Each test sequence begins with a statement that can be translated either to a unique label or to a test on a state variable. When the test sequence is broken by a *wait* instruction, the destination state is stored in the state variable before a call to the *wait()* primitive. If an *exec* instruction breaks the sequence, a statement is added which can be translated either to an assignment that stores the destination state or to a *goto* statement that jumps to the destination state. The choice of using *goto*, labels or state variables depends on the constructs used to connect the test sequences. In 2D-IDCT, the average number of test sequences is only six. This means, the number of test sequences is small enough to generate a readable code even if we use unstructured programming constructs. Therefore, all test sequences can be connected by using *goto*, *switch-case* and/or *if-else* statements, depending on the target language.

Translation of the other constructs of the AM is straightforward. Internal states are converted to global variables. Functions and procedures are converted to imperative functions.

IV. Code Generation

Compilation now proceeds by generating a sequential action scheduler from each AEIR and an actor scheduler from the network of CAL actors. The path from here depends on the target platform.

A. AEIR Transformations

The code generation from AEIR has two parts: code generation for the *action_scheduler* and for the other constructs in AEIR. Code generation of the constructs is straightforward. The actor's list of parameters and input/output ports are used to generate C header files for Epiphany and `aStruct` files for Ambric. These are used to capture the structure and the interface of the actor. CAL functions and actions are translated to target language functions. In the C code generator, since each actor is translated into a separate C file, the global variables and functions are declared as static. This will make the internal states and the functions accessible from anywhere in the current source file, but unreachable from other source files. In case of Ambric, each actor is translated to an `aJava` class, and global variables and functions are translated as member variables and functions.

Most of the statements and expressions of CAL are standard and find corresponding constructs in AEIR. However, some CAL constructs are not supported by both C and `aJava` code, so to handle this we have used *imperative pass* that translate unsupported CAL constructs to imperative constructs. For example CAL generators are handed by the imperative pass. For a simple generator, as in

```
mem := [k*2 : for k in Integers(a, b)]
```

the members of the list are stored directly in the original array:

```
int k;
for(k = a; k <= b ; k ++)
   mem[k-a] = k*2;
```

Compound generators are translated to temporary arrays in order to find a way of sequencing the statements that compute each element. When this is done, the resulting elements are placed in the final array. The issue of variable initialization is also handled by the imperative pass. In CAL, variables can be initialized by constant values, by other variables, or by any expression. On the contrary, in C, global variables can be initialized only with compile time constants. To handle this, all non-constant initializations are collected and pushed to a function dedicated to initialization. Our translator does not yet support the `Set` data type, lambda expressions, partial application and generic types.

B. Building the network

NL defines instances of actors and creates channels that connect outputs and inputs. In translating a CAL program, the translation of actors as described above has to be complemented with a translation of the NL description. This proceeds by creating instances, flattening the network and then connecting the instances of actors using channels. The flattened network is used to generate

- a round-robin scheduler for the sequential C code,
- a top-level design file for Ambric to bind the `aJava` objects that correspond to instances of CAL actor.

When creating instances, actual values are passed to parameterized actors. Every connection in the structure section of the flattened network is used to generate a bounded buffer and communication operations from the AEIR. The communication operations are *TestInputPort* to check the availability of a token on the input port, *TestOutputPort* to check the availability of place on the output port, *ReadToken* to read a token, *ConsumeToken* to consume a token, *SendToken* to send a token and *EndTransmission* to flush the buffer when the sender stops sending tokens.

There is no well defined semantics for NL to guide an implementation. In our work we have extended DPN to generate C code for GP-CPU and Epiphany, and KPN for Ambric. For DPN we have implemented a communication API [17] that uses bounded buffers to connect output ports to input ports: these are blocking for writing when the buffer is full, but allows peeking at input ports without blocking. If there are multiple actors on a single core, writing in a full buffer blocks the running of the current actor and gives control to the other actors.

C. Code Generation from AEIR

Code generation for the *action_scheduler* deals with the construct used to connect the test sequences mentioned in Section III-B, and with the interpretation of AEIR primitives, i.e *wait()* and the communication operations. Based on the construct used to connect the test sequences and the inlining of actions, we have two types of code generation: inlined and non-inlined versions. In the inlined version, each test sequence is labeled and a combination of *goto* and *switch-case* statement is used to traverse through the test sequences. Listing 2 shows the use of *goto* statements to jump to a test sequence and a *switch-case* on a state variable to move to the current state of the actor. Furthermore, the actions are inlined automatically. The non-inlined version assigns a unique number to each test sequence and uses *if-else* statements to connect the sequences. Listing 3 shows the non-inlined code generation for the *Split* actor.

```
int Scheduler_SplitTest() {
   int v_ac0;
   int v_ac1;
   switch (next_state) {
   case 0:
      goto SplitTest_State0;      break;
   case 7:
      goto SplitTest_State7;      break;
   }
SplitTest_State0:
   if (TestInputPort(&A, 1)) {
      v_ac0 = ReadToken(&A, 1);
      if ((v_ac0 >= 0)) {
         // body of action_0
         goto SplitTest_State0;
      }
      else{
         v_ac1 = ReadToken(&A, 1);
         if ((v_ac1 < 0)   {
            // body of action_1
            goto SplitTest_State0;
         }
         else {
            next_state = 7;
            wait();
         }
      }
   }
   else {
      next_state = 0;
      wait();
   }
SplitTest_State7:
   next_state = 7;
   wait();
}
```

Listing 2. Inlined code generation for the Split actor.

```
int Scheduler_SplitTest() {
 int v_ac0;
 int v_ac1;
 while(1){
   if ((SplitTest_State == 0)){
      if (TestInputPort(&A, 1)) {
         v_ac0 = ReadToken(&A, 1);
         if ((v_ac0 >= 0)) {
            action_0();
            SplitTest_State = 0;
         }
         else {
            v_ac1 = ReadToken(&A, 1);
            if ((v_ac1 < 0)) {
               action_1();
               SplitTest_State = 0;
            }
            else {
               SplitTest_State = 7;
               wait();
            }
```

```
            }
        }
        else {
            SplitTest_State = 0;
            wait();
        }
    }
    else if ((SplitTest_State == 7)) {
        SplitTest_State = 7;
        wait();
    }
  }
}
```

Listing 3. Non-inlined code generation for the Split actor.

To demonstrate the portability of the compilation tool, we have used our code generator to produce a completely sequential version of a CAL program in C for a single GP processor, parallel C code for the Epiphany [13] manycore architecture and `aJava` and `aStruct` code for Ambric [14]. In the case of Epiphany and Ambric, each actor runs on a separate processor.

1) Sequential C: Here, the CAL application is considered as a DPN where the vertices are coded in CAL. In the DPN model of computation a set of *firing rules* are evaluated to map input tokens into output tokens. CAL extends DPN by adding a state to DPN vertices and *guards* to the *firing rules*. The *firing rules* are scheduled by the AM, which is implemented as an *action_scheduler*. The *action_scheduler* checks the availability of enough tokens, evaluates the predicate and fires an action. The description of the network of actors is used to organize in-memory queues that connect appropriate outputs to inputs. In addition, the network is also used to generate a sequential scheduler for the complete system. This can be done with different degrees of sophistication. We have simply implemented a round-robin strategy among the actors. The whole application is mapped on a single GP-CPU.

We have implemented two sequential passes to generators either inlined (Listing 2) or non-inlined (Listing 3) versions. In both versions, since there are multiple actors running on a single core, the *wait()* is translated to an API that gives control to the other actors. We have used a state variable to store the next test sequence that has to be tested when the actor is active, i.e., the current state of the actor. Similarly, writes on full buffer blocks the execution of the current actor and gives control to the other actors.

2) Code Generation for Epiphany: For Epiphany we have also used an extended DPN like the sequential C, but here each actor is mapped to a separate mesh node and run concurrently while communicating with buffered FIFOs. When implementing these buffers on the Epiphany, two special features of the architecture are considered. The first one is the speed difference between read and write transactions. As mentioned earlier, writes are faster. The second one is the potential use of DMA to speed up memory transfer and allowing the processor to do processing in parallel with the memory transfer.

These features are used as a guide when building the communication library incrementally. The first implementation puts the FIFOs in the destination core memory so that reading tokens happens on the local buffer. The source core is blocked when the buffer is full. However, reading an empty FIFO is non-blocking. The second implementation tries to use DMA for transferring tokens between cores. While DMA is performed, both source and destination cores are blocked for synchronization. It is also possible to improve the performance by introducing another pair of buffers so that both cores could work on one pair of buffers while the DMA is working on the other pair; this is what we have done in the third implementation of the communication API.

3) Code Generation for Ambric: Ambric code generation generate `aJava` object and `aStruct` code for each actor and top level design file for the application. The `aJava` source code is compiled into executable code by using Ambric's compiler. Ambric's proprietary tool comes with communication API and mapping and scheduling protocols that support only KPN. Thus, for Ambric we have adapted the code generation in accordance with the existing support for KPN.

The flattened network is used to generate `aStruct` code for the top-level design which builds the full structure of the application using channels and instance declarations.

Due to the limitation of `aJava` and constraints of KPN, the Ambric code generation has different interpretation for some of AEIR primitives. Unlike DPN, we cannot check availability of tokens or read a token without consumption. Like KPN, in Ambric reading input is a blocking statement, as long as there is no input to consume, the object blocks and waits for an input token. Reading an input port consumes a token. In CAL, reading an input port to evaluate a predicate in a guard does not consume a token; tokens are consumed when an action is fired. Therefore, for Ambric testing on input port is removed from the *firing conditions* of an action. Because of this, actors whose behavior is based on the arrival time of tokens, cannot be implemented. Thus, the actions are fired sequentially according to the finite state machine, priorities and the evaluation of the guard predicates. This will give us a *deterministic sequential process* that satisfy Kahn and MacQueen semantics [10].

AEIR communication primitives are translated to the existing Ambric's communication API, i.e *ReadToken* and *ConsumeToken* are translated to `input_stream.read` and *SendToken* to `output_stream.write` operation. Because the *wait* primitive is also associated with availability of token and is already handled by Ambric's blocking reads, the translation disregards the *wait* primitive. In KPN, each token can be consumed exactly once, thus translation of *ReadToken* an *ConsumeToken* may use an additional variable to store the value of a token. If the predicates in the guard use a token from an input port, like Listing 1, the token will be stored in a global variable and the variable will be used when any action that reads from the same input port is fired. In addition, a flag is also used to indicate whether the variable is used by an action or not. If the value is already used then a new value is stored in the variable and the flag is set to *true*. The global variables and flags are used only if a guard predicate uses a token from an input port.

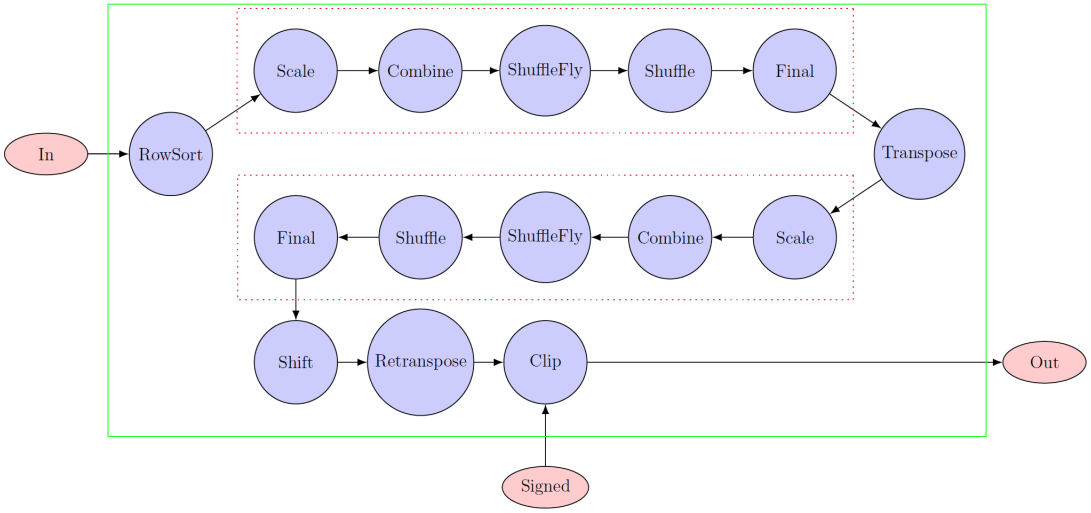

Fig. 3. Actors dataflow diagram for 2D-IDCT.

V. Experimental Case Study

In this section, we present the implementation of the Two-Dimensional Inverse Discrete Cosine Transform (2D-IDCT). Our aim is to demonstrate the applicability of CAL for many-core architectures and to verify the portability of applications by using a framework with two IRs.

IDCT is a component of techniques used in video compression encoders, such as MPEG encoding, to transform an $N \times N$ image block from the spatial domain to the DCT domain. The 2D-IDCT algorithm has been implemented using a streaming approach in the CAL language and the dataflow diagram of the implementation is shown in Fig. 3. The implementation is based on a composition of two instances of 1D-IDCT, which are interconnected via the *Transpose* actor, thus consisting of a total of 15 actors communicating in a pipeline manner. We have used a fine-grained version in order to test the framework via a network of actors.

We used DPN as MoC to generate sequential C code, which runs on a 2.8 GHz general purpose processor (GP-CPU) and parallel C code mapped on the Epiphany chip with 600 MHz RISC cores. Both versions use the same code, with different implementations for AEIR primitives. For the sequential version we used two types of code generation: inlined (Listing 2) and non-inlined (Listing 3). The inlined code generator performs function inlining and also analyses the scope of a variable to remove unused initializations. The non-inlined version does not perform any optimizations. For Epiphany we have used only the non-inlined version because it has smaller code memory foot-print. Similarly, for Ambric we have used the non-inlined version and adapted the code generation in accordance with KPN.

TABLE I
Average runtime for 8x8 IDCT block.

Sequntial C on GP-CPU		Parallel Code on 15 cores	
Non-inlined	Inlined	Epiphany@600MHz	Ambric@300MHz
$27\mu s/block$	$18\mu s/block$	$19\mu s/block$	$25\mu s/block$

Both parallel implementations map each actor on a separate core, for a total of 15 cores. In Epiphany, consecutive actors are mapped manually on adjacent cores. In case of Ambric, we have used Ambric's proprietary mapping tool.

Performance has been measured by execution on real hardware. Since the clock speeds of the parallel architectures are much slower than the one of the sequential CPU (close to 5 times slower for Epiphany and 10 times slower for Ambric) the speedup that can be expected is quite limited. The fact that the pipeline over the 15 cores is not balanced of course also limits the possible speedup.

Preliminary performance results for 2D-IDCT with 64,000 samples are shown in Table I. For all versions the output is verified against the output of the OpenDF[1] simulator. For the sequential code, the result shows that the inlining of actions and the use of *goto* has improved the performance by 33%. The Epiphany's implementation has also improved the performance of the non-inlined sequential version by 30%. However, the performance of the optimized inlined version is still better than the parallel implementations. This is because the performance of the parallel codes is greatly affected by the communication overhead, which is very common in fine-grained parallelism.

Additionally, the parallel versions are slowed down by shared memory accesses. For example, the last node (*Clip*) accesses the shared memory to read input for *Signed* port and also to write the output on *Out* port, which makes it the bottleneck of the application. Since *Clip* spends most of the clock cycles in dealing with off-chip memory accesses, the output buffer of *Retranspose* will be full. This full buffer leads to backward pressure that affects the whole implementation, making all the actors wait till there is room in the output buffer.

However, without any code optimization, and considering the low clock frequency and the extra communication over-

[1] http://opendf.sourceforge.net/

head, both parallel implementations show a potential for performance portability. In addition, the parallel implementations are more power efficient.

VI. Related Work

In prior works, CAL programs have been used as sources when generating code in sequential imperative languages and also to synthesize parallel systems. Wernlin [18] have compiled CAL actors to Java classes that extend Ptolemy atomic actor and use Ptolemy framework to schedule actors and the firing conditions of an actor. The Open-RVC CAL Compiler (ORCC) [19] generates multi-threaded C-code that can execute on a multicore processor using dedicated run-time system libraries. Roquier et al. have presented the RVC framework and Cal2C compilation tool that targets single-core processors by generating sequential C-code [20]. In particular, [21], [22], describe the use of the Cal2C tool and SystemC framework to generate C code from a network of CAL actors. There are two main differences between our work and this earlier work:

1) In our work, actors are mapped to physical cores, but in [21], [22] actors are mapped to SystemC threads.
2) In our work, scheduling of action firing conditions and execution of actions is done by AM, but in [21], [22] a round robin scheduler is used.

In [23], a methodology is proposed for synthesizing multiprocessor systems starting from a CAL application program. In [24] CAL applications have been used to design customizable Transport Triggered Architecture processors on FPGA. In our work CAL applications are mapped on manycores.

VII. Summary and Future Work

This paper proposes a new compilation framework with two IRs for the CAL dataflow language. We have presented a brief overview of the methodologies used to translate CAL actors and NL into parallel and sequential imperative code. We have described our approach to map CAL application to one single, and to two manycore architectures. The approaches have been applied on 2D-IDCT to produce three final implementations: sequential C for GP-CPU, parallel C for Epiphany, and `aJava` and `aStruct` for Ambric. We expect that the generated code will be able to meet the performance requirement of many demanding DSP applications. If not, the generated code is effortlessly readable in order to be tuned by hand.

For the future, we will make further evaluation of the approach using the whole MPEG-4 SP decoder. We have also planned to develop mapping and scheduling solutions that explore the dataflow graphs of relatively complex applications and that consider constraints and architecture specific features of the underlying parallel architecture.

Acknowledgment

The authors would like to thank Adapteva and Nethra Imaging Inc. for giving access to their software development suites and hardware boards. This research is part of the CERES research program funded by the Knowledge Foundation and the HiPEC project funded by Swedish Foundation for Strategic Research (SSF).

References

[1] S. H. Fuller and L. I. Millett, "Computing performance: Game over or next level?" *IEEE Computer*, vol. 44, no. 1, pp. 31–38, 2011.

[2] J. Armstrong, "Erlang–software for a concurrent world," in *European Conference on Object-Oriented Programming, 2007*.

[3] C. A. Varela, G. Agha, W.-J. Wang, T. Desell, K. El Maghraoui, J. LaPorte, and A. Stephens, "The SALSA programming language 1.1. 2 release tutorial," *Dept. of Computer Science, RPI, Tech. Rep*, pp. 07–12, 2007.

[4] The E language. [Online]. Available: http://www.erights.org/elang

[5] G. A. Agha, "Actors: a model of concurrent computation in distributed systems," *The MIT Press, 1985*.

[6] J. Eker and J. W. Janneck, *CAL language report: Specification of the CAL actor language*. Electronics Research Laboratory, College of Engineering, University of California, 2003.

[7] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet, "Overview of the MPEG reconfigurable video coding framework," *Journal of Signal Processing Systems*, vol. 63, no. 2, pp. 251–263, 2011.

[8] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *In Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[9] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cycle-static dataflow," *Signal Processing, IEEE Transactions on*, vol. 44, no. 2, pp. 397–408, 1996.

[10] G. Kahn and D. MacQueen, "Coroutines and networks of parallel processes," *In Information Processing '77: Proceedings of the IFIP Congress*, 1977.

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

[12] J. Janneck, "A machine model for dataflow actors and its applications," in *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*. IEEE, 2011, pp. 756–760.

[13] Adapteva, Inc, *Epiphany architecture reference G3*, 2012, rev 3.12.12.18.

[14] M. Butts, A. M. Jones, and P. Wasson, "A structural object programming model, architecture, chip and tools for reconfigurable computing," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*. IEEE, 2007, pp. 55–64.

[15] J. W. Janneck, *NL—a network language*. ASTG, Processing Solutions Group, Xilinx Inc, 2006.

[16] G. Cedersjo and J. W. Janneck, "Toward efficient execution of dataflow actors," in *Signals, Systems and Computers (ASILOMAR), 2012 Conference Record of the Forty Sixth Asilomar Conference on*. IEEE, 2012, pp. 1465–1469.

[17] M. Yang, S. Savas, Z. Ul-Abdin, and T. Nordström, "A communication library for mapping dataflow applications on manycore architectures," in *Sixth Swedish Workshop on Mutlicore Computing*, 2013.

[18] L. Wernli, *Design and implementation of a code generator for the CAL actor language*. Electronics Research Laboratory, College of Engineering, University of California, 2002.

[19] ORCC, "Open RVC-CAL compiler," http://orcc.sourceforge.net/, Accessed: 3 Aug 2013, 2014. [Online]. Available: http://orcc.sourceforge.net/

[20] G. Roquier, M. Wipliez, M. Raulet, J.-F. Nezan, and O. Déforges, "Software synthesis of CAL actors for the MPEG reconfigurable video coding framework," in *15th IEEE International Conference on Image Processing, 2008. ICIP 2008.* IEEE, 2008, pp. 1408–1411.

[21] M. Wipliez, G. Roquier, and J.-F. Nezan, "Software code generation for the RVC-CAL language," *Journal of Signal Processing Systems*, vol. 63, no. 2, pp. 203–213, 2011.

[22] C. Lucarz, M. Mattavelli, M. Wipliez, G. Roquier, M. Raulet, J. W. Janneck, I. D. Miller, D. B. Parlour *et al.*, "Dataflow/actor-oriented language for the design of complex signal processing systems," in *Conference on Design and Architectures for Signal and Image Processing (DASIP 2008) Proceedings*, 2008, pp. 1–8.

[23] C. Lucarz, G. Roquier, and M. Mattavelli, "High level design space exploration of RVC codec specifications for multi-core heterogeneous platforms," in *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*. IEEE, 2010, pp. 191–198.

[24] J. Boutellier, O. Silven, and M. Raulet, "Automatic synthesis of TTA processor networks from RVC-CAL dataflow programs," in *Proceedings of Signal Processing Systems (SiPS), 2011 IEEE Workshop on*. IEEE, 2011, pp. 25–30.