An Evaluation of Code Generation of Dataflow Languages on Manycore Architectures

Süleyman Savas, Essayas Gebrewahid, Zain Ul-Abdin, Tomas Nordström and Mingkun Yang

Centre for Research on Embedded Systems Halmstad University, Halmstad, Sweden

{suleyman.savas, essayas.gebrewahid, zain-ul-abdin, tomas.nordstrom}@hh.se, minyan09@student.hh.se

***Abstract*—Today computer architectures are shifting from single core to manycores due to several reasons such as performance demands, power and heat limitations. However, shifting to manycores results in additional complexities, especially with regard to efficient development of applications. Hence there is a need to raise the abstraction level of development techniques for the manycores while exposing the inherent parallelism in the applications. One promising class of programming languages is dataflow languages and in this paper we evaluate and optimize the code generation for one such language, CAL. We have also developed a communication library to support the inter-core communication. The code generation can target multiple architectures, but the results presented in this paper is focused on Adapteva's many core architecture Epiphany. We use the two-dimensional inverse discrete cosine transform (2D-IDCT) as our benchmark and compare our code generation from CAL with a hand-written implementation developed in C. Several optimizations in the code generation as well as in the communication library are described, and we have observed that the most critical optimization is reducing the number of external memory accesses. Combining all optimizations we have been able to reduce the difference in execution time between auto-generated and hand-written implementations from a factor of 4.3x down to a factor of only 1.3x.**

***Index Terms*—Manycore, Dataflow Languages, code generation, Actor Machine, 2D-IDCT, Epiphany, evaluation**

I. Introduction

In the last decade we have seen a transition from single core processors to multicore and manycore architectures, we furthermore see a transition to a distributed memory model where each core contains part of the common memory. These transitions have emphasized the need for programming languages that better express inherent parallelism available in the targeted applications. As a result, new languages based on parallel and concurrent programming paradigms have emerged [1], and the dataflow programming model [2], [3], [4] seems to be a particularly good candidate for the big class of streaming applications where data is processed as a continuous stream. There is however a lack of compilation frameworks, supporting dataflow languages, that generate code for the latest generation of manycore architectures.

In this paper we will evaluate and optimize a compilation framework targeting Adapteva's manycore architecture Epiphany [5]. This architecture is an example of the next generation of manycore architectures that have been developed in order to address the power and thermal dissipation limitations of traditional single core processors. In the Epiphany architecture the memory is distributed among all cores and a shared address space allows all cores to directly access data located in other cores. However, in contrast to traditional multicores there is no cache memory and no direct hardware support for memory coherence.

The dataflow language used in this work is the CAL actor language (CAL) [4], [6], which is a modern dataflow language that is adopted by the MPEG Reconfigurable Video Coding (RVC) [7] working group as part of their standardization efforts. In [8] the CAL compilation framework evaluated and optimized in this paper is described. Our code generator uses a simple machine model, called *actor machine* [9], to model actors in a dataflow application. We have also introduced an *action execution intermediate representation* [8] to support program portability and we currently support generating sequential C, parallel (manycore) C, and aJava/aStruct languages. These intermediate representations are further introduced in section IV. For the parallel C backend the code generator utilizes an in-house developed communication library [10] to support the communication between actors.

In order to evaluate various optimization methods for our CAL code generator we will use a CAL implementation of the Two-Dimensional Inverse Discrete Cosine Transform (2D-IDCT) as a case study. We will compare the performance of our automatically generated code (from CAL) with a hand-written implementation programmed in C. From this comparison a number of optimization opportunities has been found and we will be able to evaluate the benefit of the various optimizations implemented.

II. Related Works

CAL compilers have already been targeting a variety of platforms, including single-core processors, multicore processors, and programmable hardware. The Cal2C compiler [11] targets the single-core processors by generating sequential C-code. The Open-RVC CAL Compiler (ORCC) [12] generates multi-threaded C-code that can execute on a multicore processor using dedicated run-time system libraries. Similarly the d2c [13] compiler produces C-code that makes use of POSIX threads to execute on multicore processors.

In this paper, we present and evaluate a code generator for manycore architectures that use *actor machine* [9] and *action execution* intermediate representations (IRs) together with a backend for the target architecture and a custom communication library. Executing a CAL actor includes choosing an action that satisfy the required conditions and firing the action. In other CAL code generators [11] [12], all required conditions of the actions are tested at once, thus common conditions among actions will be tested more than once. However in our work, the schedule generated by the *Actor machine* stores the values of the conditions and avoids a second test on the conditions that are common.

In order to increase the portability of our compilation tool we have used an imperative IR called *action execution intermediate representation*(AEIR) which distinguishes computation and communication parts of the CAL applications. Hence it becomes easier to port the application to another architecture just by replacing the communication part with architecture specific communication operations. In addition, our compilation tool generates separate code for each actor instance so that it could be executed on individual processing cores. Therefore we do not require any run-time system support for concurrent execution of actors, in contrast to ORCC and d2c.

III. Background

A. Epiphany Architecture

Adapteva's manycore architecture Epiphany [5] is a two-dimensional array of processing cores connected by a mesh network-on-chip. Each core has a floating-point RISC CPU, a direct memory access (DMA) engine, memory banks and a network interface for communication between processing cores. An overview of the Epiphany architecture can be seen in Figure 1.

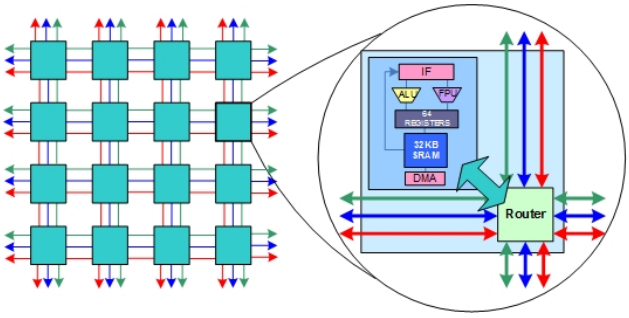

Fig. 1. Epiphany architecture overview.

In the Epiphany architecture each core is a superscalar, floating-point, RISC CPU that can execute two floating point operations and a 64-bit memory load operation on every clock cycle. The cores are organized in a 2D mesh topology with only nearest-neighbor connections. Each core contains a network interface, a multi-channel DMA engine, a multicore address decoder, and a network-monitor. The on-chip node-to-node communication latencies are 1.5 clock cycles per routing hop, with zero startup overhead. The network consists of three parallel networks which are used individually for writing on-chip, writing off-chip, and all read 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 architecture uses a distributed memory model with a single, flat address space. Each core has its own aliased, local memory range which has a size of 32kB. The local memory of each core is accessible globally from any other core by using the globally addressable IDs. However, even though all the internal memory of each core is mapped to the global address space, the cost (latency) of accessing them is not uniform as it depends on the number of hops and contention in the mesh network.

B. Dataflow Programming

Dataflow programming is a programming paradigm that models a program as a directed graph of the data flowing between operations or actors that operate on the dataflow. Dataflow programming emphasizes the movement of data and models programs as a series of connected actors. Each actor only communicates through explicitly defined input and output connectors and functions like a black box. As soon as all of its inputs become valid, an actor runs asynchronous from all other actors. Thus, dataflow languages are inherently parallel and can work well in large, decentralized systems. Dataflow programming is especially suited for streaming applications, where data is processed as a continuous stream, as, e.g., video, radar, or base station signal processing.

A number of dataflow languages and techniques were studied [2], [3], and one of the interesting modern dataflow languages is CAL Actor Language which has recently been used in the standardization of MPEG Reconfigurable Video Coding (RVC).

C. CAL Actor Language

A CAL dataflow program consists of stateful operators, called actors, that transform input streams of data objects (tokens) into output streams. The actors are connected by FIFO channels and they consist of code blocks called actions. These actions transform the input data into output data, usually with the state of the actor changed.

An application consists of a network of actors. When executed, an actor consumes the tokens on the input ports and produces new tokens on the output ports. Following example shows an actor which has one input port, two output ports and two actions.

```
actor Split() int I1 ==> P, N:
  action I1:[a] ==> P:[a]
    guard a >= 0 end
  action I1:[a] ==> N:[a]
    guard a < 0 end
```

```
end
```

The first line declares the actor name, followed by a list of parameters (which is empty, in this case) and the declaration of the input (I1) and output ports (P and N). The second line defines an action [14] and the third line defines a guard statement. In many cases there are additional conditions which need to be satisfied for an action to fire. These conditions can be specified using the guard statements, as seen in the Split actor. The schedule keyword in CAL permits to determine the transitions between the different states. Each state transition consists of three parts: the original state, a list of action tags, and the following state. For instance,

```
init (readT) --> waitA;
```

`init` is the original state, `readT` is an action tag and `waitA` is the following state. In this example, if `readT` action is executed within the `init` state, the actor will go to the next state which is `waitA`.

CAL has a network language (NL) in order to configure the structure of the application. The structure is built by instantiating the actors, defining channels and connecting the actors via these channels. Following example demonstrates a network with one input port and one output port. In the network, one processor is instantiated and it is connected to the input and output ports of the network.

```
network idct X0 ==> Y0
entities
    scale = Scale()
structure
    X0 --> scale.X0
    scale.Y0 --> Y0
end
```

IV. Code Generation

The code generator starts from high-level CAL implementation and generates a native implementation. Figure 2 shows the CAL compilation framework. In CAL, actors execute by firing of actions that satisfy all the required conditions like availability of tokens, value of the tokens and internal state of the actor. In code generation, the code generator has to consider the ordering of these conditions and the number of the tests performed before an action is fired. To schedule the testing of firing conditions of an action we have used a simple actor model called *actor machine* (AM) [9]. In addition, we also have used an *Action Execution Intermediate Representation* (AEIR) that bring us closer to imperative languages [8]. As shown in Figure 2, in our compilation process each CAL actor is translated to an AM that is then translated to AEIR. Finally, the AEIR is used by three backends that generate target specific code. Currently we have three backends: a uniprocessor backend that generates sequential C code for a single general purpose processor, an Epiphany backend that generates parallel C code for Epiphany, and Ambric Backend that generates aJava and aStruct for Ambric massively-parallel processor array [15].

A. Intermediate Representation

1) Actor Machine: AM is a controller that provide an explicit order to test all the firing conditions of all actions in an actor. AM is made up of a set of states that memorize the values of all conditions in an actor. Each state has a set of AM instructions that leads to a destination states once applied on the state. These AM instructions can be

- a *test* to perform a test on a guard or input port for availability of token,
- an *exec* for an execution of an action that satisfy all the firing conditions, or
- a *wait* to change information about absence of tokens to unknown, so that a test on an input port can be performed after a while.

Cedersjö and Janneck [16] have presented and examined several ways of transforming an actor to an actor machine . The code generator used in this paper translates CAL actors to actor machines which memorize all the conditions and have at most one AM instrucion per state.

2) Action Execution Intermediate Representation: AEIR is a data structure that is applicable to generate code for imperative languages like C, C++ and java. It has constructs for expressions, statements, function declarations and function calls. Translation of AM to AEIR deals with two main tasks. The first task is the translation of CAL constructs to imperative constructs. This includes CAL actions, variable declarations, functions, statements and expressions. The second task is the implementation of the AM that is the translation of the AM to a sequential action scheduler. The states and the instructions of the AM are translated to programing language constructs: the states are translated to unique labels or if-statement on state variable, *test* nodes to if-statements, *exec* to function call and *wait* to a call for a target specific operation that pause the execution of a process. The action scheduler is kept as a separate imperative function in the AEIR. In a prior work [8], we have presented the use of AEIR to generate sequential and parallel C code. Section V-B presents three different parallel C codes generated for Epiphany architecture.

B. Mapping CAL actors

There is no well-defined semantics for NL to guide an implementation. In our work we have used Kahn Process Networks (KPN) [17] for Ambric and extended Dataflow Process Networks (DPN) [18] to generate sequential C code for a general purpose CPU and parallel C code for Epiphany.

Ambrics proprietary tool comes with communication API and mapping and scheduling protocols that support KPN. Thus, for Ambric we have adapted the code generation in accordance with the existing support for KPN. 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 Ambrics compiler.

For DPN we have implemented a communication API that use bounded buffers to connect output ports to input ports:

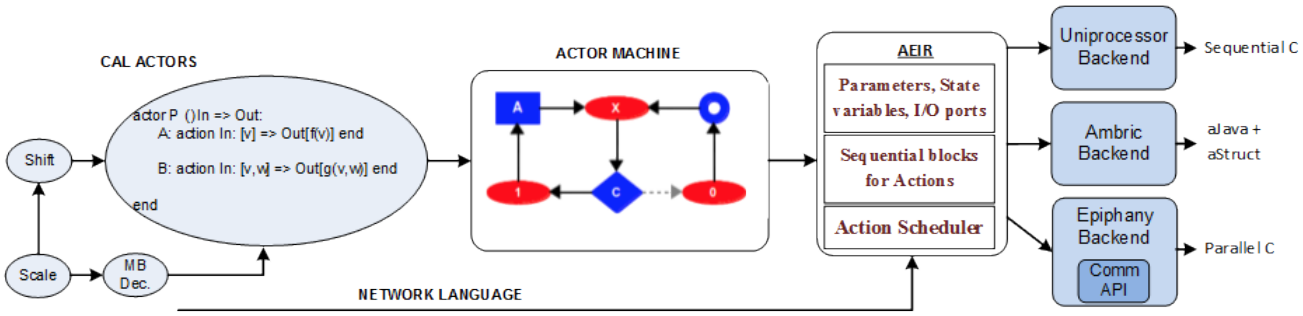

Fig. 2. CAL compilation framework.

these are blocking for writing when the buffer is full, but allows peeking without blocking. If there are multiple actors on single core, writing on full buffer blocks the running of the current actor and gives control to the other actors. The flattened network is used to generate a round-robin scheduler for the sequential C code and a host code for Epiphany that map the actors on separate processors.

1) The Dataflow Communication Library: All the communication that is done between the actors is done through FIFO buffers, thus making this functionality a key component for the compilation of the applications developed in CAL onto a manycore architecture. We suggest to implement this functionality as a dataflow communication library, with only five basic functions. In addition to the traditional functions of `write` and `read` from the FIFO buffer we have added functions such as `connect` which logically connects two actors, `disconnect` which logically disconnects the actors, and finally a function `end_of_transmission` that flushes the buffer and indicates that there are no further tokens to be sent.

When implementing these buffers on the Epiphany architecture, two special features of this architecture need to be considered. 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 to the memory transfer.

We have investigated three ways to implement the FIFO buffering. The first implementation is a 'one-end-buffer' which places the buffer inside the input port in the destination core. (Putting the buffer on the source core would result in reading from a remote core instead of writing to a remote core and thus a tenfold slowdown.) The communication overhead resides completely in the sender.

If we want to use DMA, we need to have a buffer on both the sender and receiver side. In the second implementation ('two-end-buffer') each core performs read and write transactions on its local memory and then uses DMA to transfer the data. This transfer is performed when both sides are ready, which requires that the sender's buffer is full and the receiver's buffer is empty. Even though we are using DMA for data transfer, the processor will be busy waiting for the DMA to finish. This is obviously not very efficient, and this method should be seen as a transition to our third method.

To allow the DMA to work in parallel with the processing core, we have implemented a 'double-two-end-buffer' method, which introduces two "ping-pong" buffers on each side of the communication channel. This allows the cores to work on one local buffer while the data from the other buffer is transferred to the other core by means of DMA.

If the token production rate on the sending actor is equivalent to the token consumption rate on the receiving actor, it is expected that the 'double-two-end-buffer' method should be the most efficient. On the other hand, if there is a big imbalance in the production/consumption rate, all three buffering methods will suffer from blocking, after the buffer gets full.

We have implemented the broadcast capability in all three implementations of the communication API. This mechanism is used when the output channel of an actor needs to be connected to multiple input channels of the other actors. Hence, the same output data can be written to multiple input ports and actors. In all implementations, the synchronization between sender and receiver is achieved by using flags belonging to the buffers. These flags indicate the availability of the data or the empty space in the buffers. These flags are kept in the local memories of the cores due to being polled continuously in a busy waiting loop, during the read and write operations.

The `write` and `read` function calls are designed to be *asynchronous* (non-blocking calls) by default, however, they will be blocking if the buffer is full or empty respectively. The functions `end_of_transmission`, `connect`, and `disconnect` calls will always be blocking. More details of the communication library are described in a different work [19].

V. Implementation

As our case study, we use the two-dimensional inverse discrete cosine transform (2D-IDCT), which is a component of MPEG video decoders. The CAL implementation of the 2D-IDCT is used as input to the code generator and as a reference implementation for the hand-written code. This implementation consists of 15 actors and these actors are mapped one-on-one to the Epiphany architecture using 15 out of 16 available cores. This implementation of the 2D-IDCT uses two one-dimensional inverse discrete cosine transforms after each

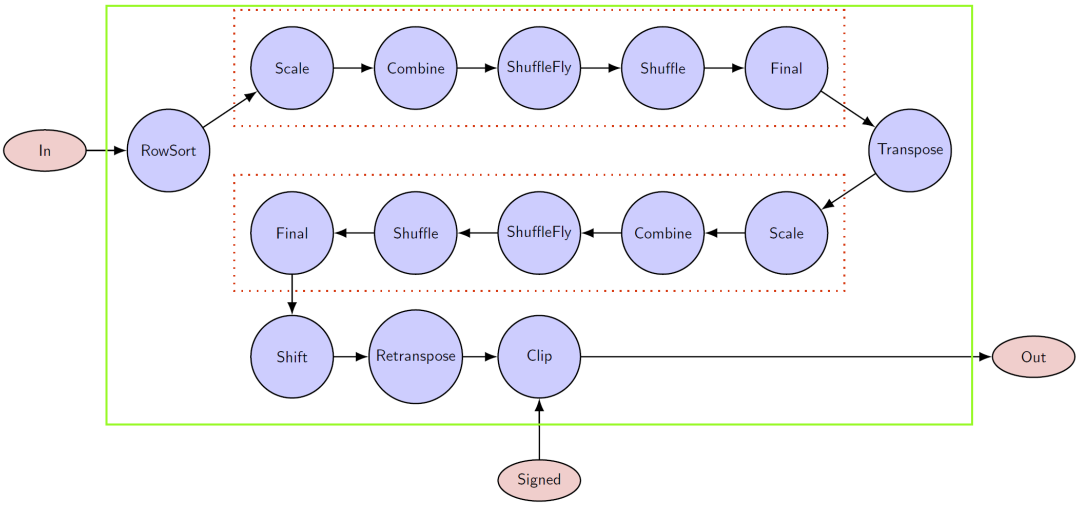

Fig. 3. The dataflow diagram of 2D-IDCT algorithm. The connections between actors, in blue, is done with 4 communication channels, while the external input, in red (In, Signed and Out), is done over a single channel.

other, with all actors connected in a pipeline fashion. One dimensional IDCTs are highlighted with the dotted rectangles in Figure 3.

Both hand-written and automatically generated implementations map the actors onto the cores by using the serpentine layout which can be seen in Figure 4. This layout takes into account the physical layout of the cores on the chip and the routing method of the architecture (X-Y routing in this case), so that consecutive actors are mapped into neighboring cores.

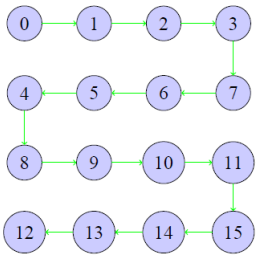

Fig. 4. Serpentine layout to map 2D-IDCT to cores on the chip. RowSort is mapped onto core 0, Clip is mapped onto the core13 whereas core 12 is not used.

A. Hand-written Implementation

The hand-written implementation of 2D-IDCT is based on the reference CAL code. The implementation includes an embedded communication mechanism. This communication mechanism is similar to the 'one-end-buffer' implementation of the communication library presented in the previous section. However, the buffer size is 1 which means an actor can send only one data token at a time to a target actor. Due to the write operation being cheaper than the read operation on the NoC of the Epiphany architecture, the write function of the communication mechanism writes the data to the memory of the remote core and the read function of this mechanism reads the data from the local memory of the core. The read and the write operations are both blocking. The communicating cores use the flags of the ports in order to inform each other about the availability of data elements or empty space in the buffer. Two actors had input availability control for all input channels in the guard states. Therefore, these actors have been modified further to be able to combine multiple communication channels into 1 channel. There is no broadcasting functionality. Majority of the actors have 1/1 input/output ratio however there are two actors which have 1/2 input/output ratio and two actors which have 2/1 input/output ratio. This information is used in order to run each actor in a loop until they produce the expected number of outputs.

B. Automatically Generated Implementations

The results from the initial and three different optimizations of our compilation framework can be found in Table I. The initial implementation, which is called as 'Uninlined' in Table I, is intended to be structurally similar to the original CAL actors. Actions are translated to two functions: *action_body* for the body of the action and a separate function, *action_guard*, to evaluate the conditions of the guard of the action. For proper initialization the variable declaration of the actions are pushed for both functions. In the action scheduler AMs *test* instruction are translated to function call to the *action_guard* and *test_input* functions, and *exec* instructions are translated to a call for *action_body*.

The second implementation, called as 'Inlined G' in Table I, optimizes the initial one by using a pass that inline all *action_guard* functions. Beside function inlining, the pass also analyses the scope of the variables and remove unused variable initializations.

The third implementation, 'Inlined G & B' in Table I, inlines both *action_guard* and *action_body* function calls. Like the second pass, third pass also use variable scope information

to eliminate unnecessary variable initializations. Thus, instead of copying the entire action variable initializations, only those variables which are used by the guard or the body of the action are initialized.

All generated implementations use the custom communication library. The parameters of the communication library such as buffer sizes, communication mechanism version and mapping layout are controlled by the code generator however for this evaluation they are kept fixed at the best configuration found after exploring the design space.

C. Differences Between Hand-written and Generated Implementations

To be able to understand the reasons behind the performance differences and evaluate the code generation, we can note some important differences between the automatically generated and the hand-written implementations. These differences are:

- State machines, in the generated code, often include high number of branches and states.
- Actions are generated as separate functions whereas in the hand-written code, they are combined in one function. Function calls add overhead (due to dealing with the stack) and decrease the performance.
- Different communication mechanisms are used. The mechanism, that is used in the generated code, is meant to be generic, e.g broadcasting is supported which is not necessary for the particular application used as the case study. Hence the generated code includes communication mechanism overhead when compared to the hand-written code.
- Several further optimizations are implemented in the hand-written code, such as reduction of channel numbers. In the CAL implementation, actors are connected to each other with 4 channels. These channels are combined into one channel in the hand-written code in order to decrease network setup overhead and simplify the usage of channels.

VI. Results and Discussion

We tested both of the implementations on the Epiphany III microchip running at 600 MHz. Among the different mechanisms of the communication library, we got the best results with the 'one-end-buffer' mechanism. Hence, we use the results of this mechanism for the discussions and the evaluation. In the test cases, we read the input data elements from the external memory and write the output elements back to the same external memory.

As can be seen in the last row of Table I the execution of the hand-written implementation takes 0.14 ms when the input size is 64 data elements (a block) and 93.6 ms when the input size is 64k data elements. These are the results which we aim to achieve with the automatically generated implementation.

Initially, the execution of the auto-generated implementation took around 1.2 ms for 64 data elements and around 405 ms for the 64k data elements. These results are given in the first row of Table I. The hand-written implementation out-performed the auto-generated implementation by 8.8x for 64 data elements and by 4.3x for 64k data elements. After further analysis, it is realized that the main bottleneck for both implementations is the external memory access. Most of the execution time is spent on reading the data from the external memory. In addition to the reading, writing to the external memory consumes some time as well. In the hand-written implementation, the external memory access for each input and output data is at a minimal level. However, in the auto-generated implementation, there were more external memory accesses per input and output elements due to both the communication library and the code generation.

In the auto-generated implementation, the variables, which were residing in the external memory, are moved to the local memory of the cores. This significantly decreased the execution time from 405 ms to 154 ms for 64k data elements, which resulted in a throughput increase by 63%. The performance of the auto-generated implementation got as close as 1.6x to the performance of the hand-written implementation. This optimization is referred as memory optimization in Table I.

TABLE I
EXECUTION TIMES AND CYCLE NUMBERS OF THE IMPLEMENTATIONS, USING SERPENTINE LAYOUT AND EXTERNAL MEMORY ACCESS (BUF SIZE: 256 FOR THE AUTO-GENERATED IMPS). (MEM OPT: MEMORY OPTIMIZATION G: ACTION_GUARD, B: ACTION_BODY, K: THOUSAND, M: MILLION)

	64 data elements	**64k data elements**
No mem opt + Uninlined	1.206ms (724k cyc)	405.6ms (243M cyc)
Mem opt + Uninlined	0.389ms (233k cyc)	153.9ms (92M cyc)
Mem opt + Inlined G	0.387ms (232k cyc)	130.6ms (78M cyc)
Mem opt + Inlined G&B	0.386ms (231k cyc)	125.2ms (75M cyc)
Manual	0.136ms (81k cyc)	93.6ms (56M cyc)

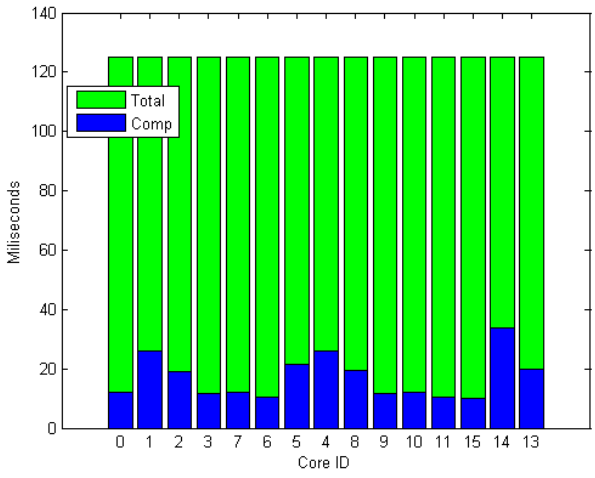

Fig. 5. Total execution times for the auto-generated implementation together with the times spent for the computation.

So far, the optimizations were done only on the communication library. Then, as the first optimization to the code generation, we inlined the functions which were the

correspondence of the guard conditions. The results, when this optimization is applied, can be seen in the third ('Mem opt + Inlined G') row of the Table I. As the second optimization on the code generator, we inlined the functions which corresponded to the action bodies. This optimization provided us a new auto-generated implementation and the results of this implementation corresponds to the fourth ('Mem opt + Inlined G & B') row of the Table I. By applying these optimizations to the code generation, we decreased the execution time from 154 ms to 125 ms for 64k data elements. This decrease in the execution time corresponds to 18% throughput increase. Using this result, the hand-written implementation shows 1.3x better performance. The difference is reduced further from 1.6x to 1.3x.

After the optimizations on the communication library, apart from the input and output data elements, no data element is kept in the external memory. There are a few data elements which reside in the shared memory and used for synchronization between the host and core0, however, they can be ignored since they are used only once before the cores start running. Even if the only access to the external memory is the access to input and output data elements we find that this external access is dominating the execution time. Figure 5 shows the total execution times of individual cores together with the time spent for the computation. The difference between the total time and the actual computation time gives the time that is spent on the communication and waiting for I/O operations due to the slow external memory accesses. In Figure 5 we can see that for most of the cores around 90% of this time (the difference of the total and the computation times) is spent on reading the input data elements from the external memory. The external memory accesses are performed by only the first and last cores. For 64k elements, the first core performs 64k external memory reads whereas the last core performs 1k external memory reads and 64k external memory writes. In the hand-written implementation, the first core runs for approximately 56 million cycles and 51.9 million of these cycles are spent for reading the input from the external memory. The situation is more or less the same in the auto-generated implementation. Due to the slow read operations in the first core, the other cores wait for the input data and this waiting time is the main contributor of the difference between the total and the computation times. The cost of the write operations, in the last core, which is far less than the cost of the read operations, are mostly hidden by the cost of the read operations. The cost of the communication between the cores and the computation times are mostly hidden by the cost of the external memory accesses. In summary, in this application, the cores are mostly blocked while reading the input data elements and they are blocked for a shorter time while writing the output data elements.

An additional aspect for comparison between the hand-written and the auto-generated implementation is the code size. In Table II the machine code size of each actor/core is given for both hand-written and auto-generated implementations. (The auto-generated implementation, which includes memory optimization and inlined action_guards & action_bodies, is used.) These code sizes are obtained by running the 'size' command in linux command line. The .elf files, which are produced by the Epiphany linker, are used as input parameters to the 'size' command. The auto-generated implementation has 71% more machine code when compared to the hand-written implementation.

TABLE II
MACHINE CODE SIZE (IN BYTES) OF EACH CORE FOR HAND-WRITTEN AND AUTO-GENERATED IMPLEMENTATIONS

Cores / Actors	Hand-written	Auto-generated
host	8,636	8,077
core0 / RowSort	5,488	9,020
core1 / Scale	5,556	9,568
core2 / Combine	5,012	9,372
core3 / ShuffleFly	5,080	8,888
core4 / Shuffle	5,556	9,452
core5 / Final	4,952	9,060
core6 / Transpose	5,708	10,628
core7 / Scale	5,244	9,568
core8 / Combine	5,008	9,376
core9 / ShuffleFly	4,920	8,892
core10 / Shuffle	5,080	9,456
core11 / Final	4,952	9,064
core12 / Shift	4,952	9,044
core13 / Retranspose	7,160	10,564
core14 / Clip	5,044	10,236
Total	87,712	150,265

Another interesting aspect is the difference in development effort writing the 2D-IDCT application in C and writing it in CAL. In Table III the number of source lines of code (SLOC) for each actor for the hand-written (native - C) implementation and the reference CAL implementation is compared. We see that in total, 495 SLOC are needed to implement the 2D-IDCT application in CAL while more than four times as many (2229 SLOC) is needed for the C implementation. This clearly indicates the expressiveness of the CAL language and the usefulness of the CAL compilation tool as described in this paper.

VII. FUTURE WORK

For further evaluation of the code generation and the custom communication library, a more complex application such as the entire MPEG-4 decoder implementation can be both automatically generated and manually written. This work might need combining and splitting the actors. Hence the compilation framework may need to be extended. In addition to the evaluation of the code generation, the mapping approach proposed by Mirza et al. [20] can be tested with this application.

The performance of both hand-written and auto-generated implementations might be increased by further optimizations. We observed that the main bottleneck for both of the implementations is the read operations performed on the external memory. An optimization could be using the write operations

TABLE III
SOURCE LINE OF CODE OF EACH ACTOR FOR HAND-WRITTEN AND CAL IMPLEMENTATIONS. (THE NUMBER OF CODE LINES ARE THE SAME FOR THE DIFFERENT INSTANCES OF THE SAME ACTORS.)

Actors + libs	Hand-written	CAL
RowSort	190	57
Scale	142	26
Combine	148	48
ShuffleFly	116	22
Shuffle	143	41
Final	113	10
Transpose	214	71
Shift	109	9
Retranspose	203	54
Clip	175	45
Network + comm	333	112
Host	343	–
Total	2229	495

instead of read operations. Such as, instead of the Epiphany core(s) reading the input data from the external memory, the host can write the data to the Epiphany core's memory. The synchronization between the host and the Epiphany core can be obtained by using the blocking write operations. Hence no read operations would be used. Another optimization for the communication library would be customization of the functionality. E.g the operations such as broadcasting can be turned on and off for different applications in order not to add overhead when the operation is not needed. A radical optimization can be replacing the communication mechanism with a more efficient one which can make better use of the specific features of the architecture.

In our test application, each actor is connected only to the neighbor actors. Hence the communication is not very complicated, and choosing the best fitting layout is not a complex task. However, with larger applications when the communication patterns get more complicated, choosing the mapping layout, quickly becomes a very complex task. For e.g individual actors might be connected to several other actors. Hence the mapping method would need to take network usage, communication frequency between actors etc. into account while searching for the best fitting layout. Mirza et al. [20] propose a solution to the mapping, path selection and router configuration problems. They refer that their approach is capable of exploring a range of solutions while letting the designer to adjust the importance of various design parameters.

As an extension to the compilation framework we used in our work, new backends can be implemented and integrated to the compilation tools in order to target other manycore architectures and the code generation can be tested on these architectures.

VIII. CONCLUSIONS

Manycore architectures are emerging to meet the performance demands of high-performance embedded applications and overcome the power dissipation constraints of the existing technologies. A similar trend is visible in programming languages in the form of dataflow languages that are naturally suitable for streaming applications. This paper deals with the evaluation of a compilation framework along with a custom communication library that takes CAL actor language (CAL) code as input and generates parallel C code targeting Epiphany manycore architecture. As a case study we have used a CAL implementation of a 2D-IDCT application and compare the automatically generated code from the proposed tool-chain with a hand-optimized native-C implementation.

The preliminary results reveal that the hand-written implementation has 4.3x better throughput performance with respect to the auto-generated implementation. After memory access optimizations on the communication library, the auto-generated implementation gained 63% throughput increase. With further optimizations on the code generation, the throughput of the auto-generated implementation was further improved by 18%. To sum up, we are able to decrease the difference in execution time between the hand-written and the auto-generated implementations from a factor of 4.3x to 1.3x. In terms of the development effort by considering source lines of code metric, we observe that the CAL based approach requires 4.5x less lines of source code when compared to the hand-written implementation.

To conclude we are able to achieve competitive results of execution time with respect to the hand-written implementation and the use of high-level language approach leads to reduced development effort. We also foresee that our compilation methodology will result in focusing on optimizing the tool-chain to produce efficient implementations rather than manual optimizations performed on each application.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] Z. Ul-Abdin and B. Svensson, "Occam-pi for programming of massively parallel reconfigurable architectures," *International Journal of Reconfigurable Computing*, vol. 2012, no. 504815, p. 17, 2012.

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

[3] B. Bhattacharya and S. S. Bhattacaryya, "Parameterized dataflow modeling for DSP systems," *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, 2001.

[4] J. Eker and J. W. Janneck, "CAL language report specification of the CAL actor language," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M03/48, 2003.

[5] Epiphany, "Epiphany architecture reference G3, rev 3.12.12.18," Adapteva, Tech. Rep., 2013.

[6] J. Eker and J. W. Janneck, "Dataflow programming in CAL – balancing expressiveness, analyzability, and implementability," in *Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR), 2012*. IEEE, 2012, pp. 1120–1124.

[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, Springer*, 2009.

[8] E. Gebrewahid, M. Yang, G. Cedersjö, Z. Ul-Abdin, J. W. Janneck, V. Gaspes, and B. Svensson, "Realizing efficient execution of dataflow actors on manycores," in *International Conference on Embedded and Ubiquitous Computing, IEEE*, 2014.

[9] 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.

[10] 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.

[11] 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.

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

[13] D2C, "CAL ARM compiler," http://sourceforge.net/projects/opendf/, Accessed: 3 Aug 2013, 2013. [Online]. Available: http://sourceforge.net/projects/opendf/

[14] J. W. Janneck, "Tokens? what tokens? - a gentle introduction to dataflow programming," Programming Solutions Group, Xilinx Inc., Tech. Rep., 2007.

[15] A. Jones and M. Butts, "Teraflops hardware: A new massively-parallel mimd computing fabric ic," in *Proceedings of IEEE Hot Chips Symposium*, 2006.

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

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

[19] M. Yang, "CAL code generator for epiphany architecture," Master thesis report 1401, Halmstad University, 2014.

[20] U. M. Mirza, F. Gruian, and K. Kuchcinski, "Design space exploration for streaming applications on multiprocessors with guaranteed service noc," in *NoCArc '13 Proceedings of the Sixth International Workshop on Network on Chip Architectures*, 2013.