A C++-embedded Domain-Specific Language for Programming the MORA Soft Processor Array

W. Vanderbauwhede
Dept. of Computing Science
University of Glasgow
Glasgow, UK
wim@dcs.gla.ac.uk

M. Margala, S. R. Chalamalasetti, S. Purohit
Dept. of Electrical and Computer Engineering
University of Massachusetts at Lowell
Lowell, MA, USA
martin_margala@uml.edu,
sairahul_chalamalasetti@student.uml.edu,
sohan_purohit@uml.edu

***Abstract*—MORA is a novel platform for high-level FPGA programming of streaming vector and matrix operations, aimed at multimedia applications. It consists of soft array of pipelined low-complexity SIMD processors-in-memory (PIM). We present a Domain-Specific Language (DSL) for high-level programming of the MORA soft processor array. The DSL is embedded in C++, providing designers with a familiar language framework and the ability to compile designs using a standard compiler for functional testing before generating the FPGA bitstream using the MORA toolchain. The paper discusses the MORA-C++ DSL and the compilation route into the assembly for the MORA machine and provides examples to illustrate the programming model and performance.**

***Index Terms*—Reconfigurable Processor, Soft Processor Array, Multimedia Processing, Domain-Specific Language**

I. Introduction

Media processing architectures and algorithms have come to play a major role in modern consumer electronics, with applications ranging from basic communication devices to high level processing machines. Therefore architectures and algorithms that provide adaptability and flexibility at a very low cost have become increasingly popular for implementing contemporary multimedia applications. Reconfigurable or adaptable architectures are widely being seen as viable alternatives to extravagantly powerful General Purpose Processors (GPP) as well as tailor made but costly Application Specific Circuits (ASICS). Over the last few years, FPGA devices have grown in size and complexity. As a result, many applications that were earlier restricted to ASIC implementations can now be deployed on reconfigurable platforms. Reconfigurable devices such as FPGAs offer the potential of very short design cycles and reduced time to market.

However with the ever increasing size and complexity of modern multimedia processing algorithms, mapping them onto FPGAs using Hardware Description Languages(HDLs) like VHDL, Verilog provided by many FPGA vendors has become increasingly difficult. To overcome this problem several groups in academia as well as industry have engaged in developing high level language support for FPGA programming. The most common approaches fall into three main categories: *C-to-gates*, *system builders* and *soft processors*.

The *C-to-gates* design flow uses special C dialects and additional keywords or pragmas to a subset of ANSIC C language specifications to extract and control parallelism out of algorithms. Promising examples in this category are Handel-C [1], Impulse-C [2], Streams-C [3], and Trident [4]. Despite the advantage of a smaller learning curve for programmers to understand these languages, a significant disadvantage of this C-based coding style is that it is customized to suit Von Neumann processor architectures which cannot fully extract parallelism out of FPGAs.

By *system builders* we mean solutions that will generate complex IP cores from a high-level description, often using a wizard. Examples are Xilinx's CoreGen and Altera's Mega wizard. These tools greatly enhance productivity but are limited to creating designs using parameterized predefined IP cores.

Finally, *soft processors* have increasingly been seen as strong players in this category. Each FPGA vendor provides their own soft cores such as Microblaze and Picoblaze from Xilinx and Nios from Altera. However, the traditional architectures with shared memory access and mutual memory access are far from ideal to exploit the inherent parallelism inherent in FPGAs for media processing applications. To address this problem, different processor architectures are needed. One such architecture, commercialized by Mitrionics, is the "Mitrion Virtual Processor" (MVP), a massively parallel processor that can be customized for the specific programs that run on it [5]. Other alternatives are processor arrays such as [6], which is based on the OpenFire processor.

II. Contribution of the Paper

In this paper we present a high-level programming solution for the MORA (Multimedia Oriented Reconfigurable Architecture) soft processor array, as a solution at a lower granularity than conventional processor arrays but higher than the MVP. The MORA architecture is composed of an array of small pipelined SIMD processor-in-memory (PIM) cells [7]. A key feature of our approach is that each processor is customized at compile time for program specific instruction execution. The interconnects, memories and processing unit modules are all customized at compile time, resulting in an instance of the array optimised for running a particular program.

The main contribution of the paper is a MORA-C++, a Domain-Specific Language (DSL) for programming the MORA soft processor array at high level. MORA-C++ is embedded in C++ as an API an a set of rules. Consequently, any MORA-C++ program is valid C++ and can also be compiled to run on the host platform. The main advantage of embedding a DSL into a mainstream language like C++ is that the developers do not need to learn an entirely new language (as is the case for e.g. the Mitrion platform); on the other hand, thanks to the PIM array abstraction, there is no need for the programmer to have in-depth knowledge of the FPGA architecture (as is the case for e.g. Handel-C and Catapult-C).

To understand the compilation process and programming model, we also present the MORA assembly language, and advanced assembly language that can be targeted by high-level language compilers and a complete tool chain to automatically generate the FPGA configuration from the assembly source code. Our initial results show that the concept has great potential for high-level programming of multimedia processing applications on FPGAs.

It should be noted that MORA-C++ and the MORA assembly language are not limited to deployment on the FPGA-based soft MORA: the ASIC version of MORA shares the same processor architecture and will therefore run the same programs as the soft MORA. However, the soft MORA is an ideal development platform for exploring new designs and features to be implemented on the ASIC MORA.

III. MORA Architecture

The MORA processor array is aimed towards the implementation of streaming algorithms for media processing applications. It consists of an array of small processing cores called Reconfigurable Cells (RC) [8], [9]. For the FPGA-based soft MORA, the RCs and their connections are instantiated at compile time. In the ASIC version the RCs and the interconnection network are run-time reconfigurable.

Each RC (Figure 1) has a PIM architecture [7] with a small (typically 256 bytes) local memory, two external input ports and two output ports.

In order to decrease the memory access delays between the data RAM and the processing core as well as to avoid memory contention issues between multiple cells (as would be the case in a shared-memory architecture), each RC has a local data memory. The Processing Element of the RC performs fixed-point arithmetic, logical, shifting and comparison operations. The Control Unit inside each RC manages internal synchronization within the processor as well as external communication with other RCs. To achieve better resource utilisation and performance, data memories are implemented on the block RAMs (BRAMS) available in the FPGA. The control unit also handles asynchronous handshaking mechanism to control data flow within the array.

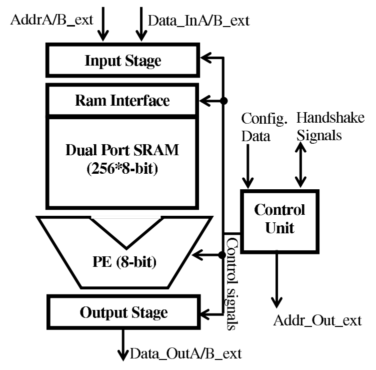

Figure 1. MORA Reconfigurable Cell (RC)

A. Asynchronous Handshake

To minimize the impact of communication networks on the power consumption of the array, each RC is equipped with a simple and resource efficient communication technique. As every RC can in principle operate at a different clock speed, an asynchronous handshake mechanism was implemented. As MORA is a streaming architecture, a two-way communication mechanism is required, one to communicate with the upstream RCs and another to communicate with the downstream RCs. Altogether, a total of four communication I/O signals are used by each RC to communicate with the other RCs efficiently in streaming fashion. They are described as follows:

- rc_rdy_up is an output signal signifying that the RC is idle and ready to accept data from upstream RCs.
- rc_rdy_down is an input signal signifying the downstream RCs are idle and ready to accept new data.
- data_rdy_down is an output signal asserted when all the data transfers to the downstream RCs are completed.
- data_rdy_up is an input signal to RC corresponding to the data_rdy_down signal from upstream RCs.

Each RC can accept inputs either from two output ports of a single RC or from two individual ports of different RCs. The output of each RC can be routed to at most of four different RCs. In order to support multiple RC connections to a single cell, a two bit vector for data_rdy_up (data_rdy_up[1:0]) and four bit vector for rc_rdy_down (rc_rdy_down[3:0]) is used.

B. Execution Model

The RC has two operating modes: processing and loading. When the RC is operating in processing mode, it can either write the processed data back into internal memory or write to a downstream RC. Each RC has two execution modes while processing input data. One is a sequential way of execution used for normal instructions (ADD, SUB, etc..) with write-back option. The second is pipelined execution for accumulation and instructions with write-out option. Instructions with sequential execution take three clock cycles to complete, with each clock cycle corresponding to reading, executing and writing data to the RAM. A prefetching technique is used for reading instructions from the instruction memory, this involves reading a new instruction word while performing the last operation of the previous instruction. This approach saves one clock cycle for every new instruction.

For pipelined operation the controller utilizes the pipelining stage between the RAM and the PE. This style of implementation allows the accumulation, write-out operations to complete in n+2 clock cycles. The latency of 2 clock cycles results from reading and execution of the first set of operands. The single-cycle execution for instructions with write-out option makes the RC very efficient for streaming algorithms.

IV. The MORA assembly language

The MORA assembly language has been published in detail in [10]. We summarize briefly its most important characteristics.

The aim of the MORA assembly language is to serve as a compilation target for high-level languages such as MORA-C++ whilst at the same time providing a means of programming the MORA processor array at a low level.

The language consists of three components: a *coordination* component which allows to express the interconnection of the RCs in a hierarchical fashion, an *expression* component which corresponds to the conventional assembly languages for microprocessors and DSPs and a *generation* component which allows compile-time generation of coordination and expression instances.

A. Expression Language

The MORA expression language is an imperative language with a very regular syntax, similar to other assembly languages: every line contains an instruction which consists of an operator followed by list of operands. The main differences with other assembly languages are:

• Typed operators: the type indicates the word size on which the operation is performed, e.g. bit, byte, short.

• Typed operands: operands are tuples indicating not only the address space but also the data type, i.e. word, row, column, or matrix.

• Virtual registers and address banks: MORA has direct memory access and no registers (an alternative view is that every memory location is a register). Operations take the RAM addresses as operands; however, "virtual" registers indicate where the result of an operation should be directed (RAM bank A/B, output L/R/both)

We illustrate these characteristics with an example. The instruction for a multiply-accumulate of the first row of 8×8-matrix in bank A with the first column of 8×8 matrix in bank B reads in full:

```
MULTACC:C 8 Y A:0:W A:0:R B:0:C
```

The '8' indicates that 8 operations need to be performed on a range of addresses. The 'Y' is a virtual register indicating that the resulting 2 bytes must be written to the output ports. The ':C' indicates the type of operand of the operation, in this case 'char' (1 byte). The groups A:0:W etc are the address tuples. They encode the address and the type of the data stored at the address, in this case a word (1 byte) stored at address 0 of bank A. The tuple B:0:C encodes a 8×1 column of bytes starting at address 0 of bank B.

B. Coordination Language

MORA's coordination language is a compositional, hierarchical netlist-based language similar to hardware design languages such as Verilog and VHDL. It consists of primitive definitions, module definitions and instantiations.

Primitives describe a MORA RC. They have two input ports and two output ports. Modules are groupings of instantiations, very similar to non-RTL Verilog. Modules can have variable numbers of input and output ports. Instantiations define the connections between different modules or RCs, again very similar to other netlist-based languages.

C. Generation Language

The generation language is an imperative mini-language. The language acts similar to the macro mechanism in C, i.e. by string substitution, but is much more expressive.

The generation language allows instructions to be generated in loops or using conditionals. The generation language can also be used to generate module definitions through a very powerful mechanism called module templates. Instantiation of a module template results in generation of a particular module (specialization) based on the template parameters. This is similar to the way template classes are used in C++.

V. The MORA-C++ Domain-Specific Language

The MORA-C++ DSL allows the developer to program the MORA platform using a C++ API and a subset of the full C++ language. The API is used to describe the connections between RCs or groups of RCs (modules). The C++ subset is used to describe the functionality of the RC or module. The rationale for this approach (i.e. giving the programmer full control over the RC functionality and interconnections) is based on performance: to write a high-performance program for a given architecture requires in-depth knowledge of the architecture. For example, to write high-performance C++ code one needs a deep understanding of stack and heap memory models, cache, bus access and I/O performance. Conversely, to write a high-performance program for MORA, one needs to understand the MORA architecture and make the best possible use of it. Therefore we do not attempt to hide the architecture from the developer but we expose it through a high-level API. On the other hand, because of the processor array abstraction, there is no need for the programmer to have in-depth knowledge of the FPGA architecture.

A. Key Features

The MORA-C++ DSL relies heavily on the type system to determine the compilation route for given expressions. Because MORA is targeted at vector and matrix operations, these are fundamental types in the DSL. Operators are overloaded to support powerful matrix and vector expressions. The type system is also used to infer RCs to split and merge signals, so that there is no need for the developer to explicitly instantiate them. Finally, MORA-C++ uses automatic static memory allocation. Apart from the powerful syntactic constructs, this is probably the most significant feature of the DSL. Static memory allocation is of course standard in C/C++, but essentially the compiler assumes that the memory is infinite. In MORA, the local memory of each RC is very small, requiring the compiler to check if there is sufficient memory available for a given program.

B. MORA-C++ by Example

In this section we illustrate the features of MORA-C++ using an implementation of a DWT and a DCT algorithm as example.

1) Discrete Wavelet Transform: As an example application to illustrate the features of the MORA assembly we present the implementation of the Discrete Wavelet Transform (DWT) algorithm. An 8-point LeGall wavelet transform is implemented using a pipeline of 4 RCs, each RC computes following equations :

$$\begin{aligned} y_i &= x_i - (x_{i-1} + x_{i+1})/2 \\ y_{i-1} &= x_{i-1} + (y_i + y_{i-2})/4 \end{aligned}$$

The MORA-C++ code for the pipeline stages is implemented as a single function template:

```
template <int N,typename TL,typename TR>
  UCharPair dwt_stage(TL x,TR y_in) {
  UChar y_l;UChar y_r;
  if (N==6) {
      y_r = x[1] - x[0]/2;
  } else {
      y_r = x[1] - (x[0]+x[2])/2;
  }
  if (N==0) {
      y_l = x[0] + y_r/4;
  } else {
      y_l = x[0] + (y_r+y_in)/4;
  }
  UCharPair out(y_l,y_r); return out;
}
```

Using this template, the complete DWT algorithm becomes

```
Pair<Row8,Nil> dwt (Row8 inA) {
    vector<UChar> v012 =inA.slice(0,2);
    vector<UChar> v234 =inA.slice(2,4);
    vector<UChar> v456 =inA.slice(4,6);
    vector<UChar> v67 =inA.slice(6,7);
    Row3 x012(v012);
    Row3 x234(v234);
    Row3 x456(v456);
    Row2 x67(v67);
    Row8 ny;
    UCharPair res01=
dwt_stage<0,Row3,Nil>(x012,_);
    ny[1]=res01.left; ny[0]=res01.right;
    UCharPair res23=
dwt_stage<2,Row3,UChar>(x234,ny[1]);
    ny[3]=res23.left; ny[2]=res23.right;
    UCharPair res45=
dwt_stage<4,Row3,UChar>(x456,ny[3]);
    ny[5]=res45.left; ny[4]=res45.right;
    UCharPair res67=
dwt_stage<6,Row2,UChar>(x67,ny[5]);
    ny[6]=res67.left; ny[7]=res67.right;
    Pair< Row8, Nil > res(ny,_);
    return res;
}
```

The example illustrates several features of MORA-C++:

a) Data Types: The type `UCharPair` is a typedef for `Pair<UChar,UChar>`; a Pair is the fundamental template class used for returning data from an RC (for higher-level modules with more than two output ports, there is a Tuple class). The Pair has accessors `left` and `right` for accessing its elements.

MORA-C++ defines a number of signed scalar types, Char, Short, Int, Long and unsigned versions UChar etc. These map to 1, 2, 4, 8 bytes respectively. A special scalar type Nil is also defined and used to indicate unconnected ports. The constant variable '_' is of this type.

The types `RowN` are typedefs for `Row<UChar,N>`. Apart from the row vector, MORA-C++ also defines a Col vector and a Matrix type. All three template classes inherit from the STL `vector<>` template.

b) Split and merge: The example also illustrates the use of the slice method for accessing a subset of the data and the use of indexing for accessing scalar data. This is an important abstraction as it relieves the developer of having to create and connect RCs purely for splitting and merging data.

2) Discrete Cosine Transform: To illustrate another key feature of MORA-C++, operator overloading, we present the implementation of the 2-D Discrete Cosine Transform algorithm (DCT) on an 8×8 image block. In its simplest form, the DCT is a multiplication of a pixel matrix A with a fixed coefficient matrix C as follows:

$$M_{DCT} = C.A.C^T$$

The DWT is a pipelined algorithm with little or no parallelism, and as such only illustrates the pipelining feature of the MORA array. The DCT however provides scope for parallelism, by computing the matrix multiplication using a parallel divide-an-conquer approach (see Section VI-C).

The implementation of the DCT in MORA-C++ is extremely simple and straightforward:

```
typedef Matrix<UChar,8,8> Mat;
const UChar ca[8][8]={ ... };
const Mat c((const UChar**)ca);
const Mat ct = c.trans();

Pair<Mat,Nil> dct (Mat a) {
Mat m=c*a*ct;
    Pair<Mat,Nil> res(m,_);
    return res;
}
```

As the example shows, the multiplication operator (and other arithmetic and logic operators) are overloaded to provide matrix operations. The other classes Row and Col also provide overloaded operations, making integer matrix and vector arithmetic in MORA-C++ very simple.

VI. Compilation

As a MORA-C++ program is valid C++, it can simply be compiled using a compiler such as gcc. This is extremely useful as it allows for rapid testing and iterations. The API implementing the DSL attempts to catch as many architecture-specific issues at possible, so that a MORA-C++ program that

Algorithm 1 Memory allocation algorithm

```
1) Allocate space for used function arguments
2) Allocate space for constant data
3) Convert expressions into SSA
4) Identify intermediate expressions
5) Allocate space for intermediate expressions
```

Algorithm 2 Optimal grouping of slices

```
1) For every slice, group with all other slices in the set.
   Let $s_i(b_i, e_i)$ and $s_j(b_j, e_j)$ be the grouped slices. Every
   grouping receives a weight
   $w_ij = max(e_i, e_j) - min(b_i, b_j) + 1$ , i.e. the size of the
   combined slice.
2) Remove the group with the lowest weight from the set,
   assign to the first RC
3) Repeat the procedure until the set contains 0 or 1 slices
   ($N_S$ times if $N_S$ is even, $N_S - 1$ times if $N_S$ is odd)
4) Using the combined slices, repeat the procedure for the
   next level of the tree.
5) Finally, if $S_N$ is odd, prune the tree, i.e. remove any
   intermediate RCs that return the same slice as they
   receive.
```

works correctly at this stage will usually need no modifications for deployment on the FPGA. Some issues can however not be caught by the C++ compiler, for example it is not possible to determine the exact amount of MORA assembly instructions for any given MORA-C++ program (obviously, as gcc will produce assembly for the host platform, not for the MORA array). If the number of instructions exceeds the maximum instruction size of the RC, this error can only be caught by the actual compilation to MORA assembly.

To be able to emit MORA assembly from MORA-C++, the compiler needs to perform several actions. The most important ones are:

- memory allocation
- inferring split and merge trees
- inferring template modules

A. *Memory Allocation*

The MORA-C++ compiler considers the MORA RC's memory as two logical banks, typically one for the left port inputs and one for the right port; the total memory is a fixed value MEMSZ. Allocation is performed based on the type, i.e. the dimensions of the matrix. Thus, allocation is a 1-D bin packing problem with two bins; however, because of the limited size of the memory, a simple heuristic algorithm can be used. The outline of the overall allocation algorithm is shown in Algorithm 1. The third step of the algorithm is to convert expressions into Static Single Assignment (SSA) [11]. This is an intermediate representation commonly used in compilers for the purpose of memory/register allocation. Essentially, it consists of assigning every expression to a unique variable (hence the name "single assignment"). Intermediate expressions are those expressions that are not part of the returned tuple.

B. *Inferring Split and Merge Trees*

In most cases, the input data for a program will have to be distributed over a number of RCs for computation. For example the DWT algorithm requires an 8-byte vector to be split into three 3-byte vectors and one 2-byte vector. Conversely, to collect the final data for output, usually results from several RCs have to be merged. In MORA-C++ splitting of a vector into subvectors is achieved via the slice method, merging of subvectors into a single vector via the splice method. To split or merge single elements indexing is used. The compiler has to infer a corresponding "split tree" and "merge tree", a tree of RCs that performs the required operations.

1) Split algorithm: Because of the definition of *slice*, any intermediate slices can be removed: Let v be a vector of N elements 0..N-1 (of some type T):

```
Row<T,N> v;
s1=v.slice(b1,e1);
s2=s1.slice(b2,e2);
```

Obviously b1,b2≥0; e1,e2<N; also, b2≥b1 and e2≤e1 or the slice call will throw an exception. With these restrictions on the bounds of the slice, the following identities hold:

```
s1.slice(b2,e2)≡
v.slice(b1,e1).slice(b2,e2)≡
v.slice(b2,e2)
```

The compiler has to infer the tree of RCs required to slice the divide data into the given slices. Let the total number of slices be N_S. Because every RC has 2 outputs, the tree is a binary tree. The process consists of following steps:

- Determine the minimum required number of RCs, N_{RC}

  $$N_{RC} = \{\ N_S/2\ \ , N_S \text{ is even} (N_s + 1)/2\ , N_S \text{ is odd},$$

- Compute the number of levels the tree (closest power of 2)
  $N_{lev} = \lceil log_2(N_{RC}) \rceil$
- Optimal grouping of the slices
  In many cases, some of the slices will overlap to some degree. The RCs have instructions to move a contiguous range of data in an efficient way (1 cycle per word + 1 cycle overhead); moving a non-contiguous set of data requires one instruction per subset, increasing the overhead. Consequently, it pays to move the smallest contiguous range required for the two slices of the leaf RCs. To determine the optimal grouping, we use a recursive algorithm as shown in Algorithm 2:

2) Merge algorithm: The complement of the split algorithm follows entirely the same pattern: a merge tree can be viewed as an upside-down split tree; the main difference is that ranges to be merged should be non-overlapping.

C. *Compilation of Matrix and Vector Arithmetic using Module Templates*

As discussed in Section IV, the MORA assembly language provides the ability to generate code at compile time and the ability to group instantiations of RCs into hierarchical modules. Module templates combine both features: when a module template is instantiated, it generates a module based on the template parameters. This feature of the assembly language was designed with the express purpose of supporting code generation from overloaded matrix and vector operations. Consider for example a matrix multiplication:

```
Matrix<UChar, NR1,NC1> m1;
Matrix<UChar, NC1,NC2> m2;
Matrix<UChar, NR1,NC2> m12;
m12=m1*m2;
```

The multiplication is computed by splitting the matrices into two submatrices (NR1/2)×NC1 and NC1×(NC2/2). The computation of m1.m2 results in four sub-matrices of size (NR1/2) ×(NC2/2) being computed in parallel and then combined into m12. Of course it is possible to split either one of the matrices into more submatrices, but this leads to larger numbers of RCs being used. In Section VIII we present both the smaller and the faster implementation of the DCT.

In terms of implementation, the multiplication is implemented as a template module which takes the dimensions of both matrices as parameters. Furthermore, if the multiplication is part of a compound arithmetic expression, the intermediate connections will use the full bandwith available, rather than inserting merge and split trees. The current algorithm for deciding if a template module can be used and if split/merge trees should be inferred is simple:

- Only compound arithmetic expressions on matrices or vectors will be implemented as a template module. This means that expressions with control constructs are not implemented this way, nor are expressions that result in changing the data type by slicing, splicing or joining.
- For every such compound expression, a split tree will be inferred for the leaf terms and a merge tree for the result.

As the actual syntax of the MORA assembly template modules is out of scope for the paper, we present the module template using the equivalent MORA-C++ syntax. This also gives a good idea of how much complexity is handled by the compiler when inferring a template module from an overloaded matrix multiplication. For conciseness we have omitted the type fields of the template instances.

```
template <int NR1,int NC1,int NC2>
Pair<Matrix<NR1,NC2>,Nil> mmult4
(Matrix<NR1,NC1> m1, Matrix<NC1,NC2> m2) {
Matrix<NR1,NC2> m_res=m1*m2;
Pair<Matrix<NR1,NC2>,Nil> out(m_res);
return out;
}
template <int NR1, int NC1, int NC2>
Tuple<Matrix<NR1,NC2> > mmult
(Matrix<NR1,NC1> m1,Matrix<NC1,NC2> m2) {

  // split. In assembly, this is a split tree
  b11=m1.block(0,0,NR1/2-1,NC1);
  b12=m1.block(NR1/2,0,NR1/2,NC1);
  b21=m2.block(0,0,NC1,NC2/2-1);
  b22=m2.block(0,NC2/2,NC1,NC2);
  // compute partial results
  Pair<Matrix<NR1/2,NC2/2>,Nil> p11=
  mmult4<NR1/2,NC1,NC2/2>(b11,b21);
  Pair<Matrix<NR1/2,NC2/2>,Nil> p12=
  mmult<NR1/2,NC1,NC2/2>(b11,b22);
  Pair<Matrix<NR1/2,NC2/2>,Nil> p21=
  mmult4<NR1/2,NC1,NC2/2>(b12,b21);
  Pair<Matrix<NR1/2,NC2/2>,Nil> p22=
  mmult4<NR1/2,NC1,NC2/2>(b12,b22);
  // merge. In assembly, this is a merge tree
  Matrix<NR1,NC2/2> m_u =
  p11.left.merge<NR1/2,NC2/2>(p12.left);
  Matrix<NR1,NC2/2> m_l =
  p11.left.merge<NR1/2,NC2/2>(p12.left);
  Matrix<NR1,NC2> m =
  m_u.merge<NR1,NC2/2>(m_l);
  Tuple<Matrix<NR1,NC2> > out(m);
  return out;
}
```

The equivalent MORA assembly template is structurally identical to the MORA-C++ version. The above code also serves to illustrate the MORA-C++ merge function. This is a method call implemented using a polymorphic function template which works out how to merge the matrices based on the specified return type, i.e. the dimensions of the returned matrix.

VII. Soft Processor Array and Bitfile Generation

The MORA-C++ compiler is written in the functional language Haskell using the Parsec parser combinator library. It compiles the MORA-C++ program into the MORA assembly language. The MORA assembler (written in Perl) generates the VHDL code for the soft processor array. The MORA processor array is configured for a given program by configuring the instruction and data memories for each RC and connecting the RCs as required. The assembler generates the required memory configurations and the interconnect configurations.

A. *Memory Configuration*

To generate the data memory configurations we use the Xilinx CoreGen utility. The content of the memories can be provided using external files (.coe files) at synthesis time. As every RC can have a different memory configuration, a template-driven generator is used to create multiple instances of the CoreGen configuration file templates. It then generates and runs a script which calls coregen to build the actual memories.

B. *RC Generation*

The RC contains the memories for data and instructions, the PE and the control unit. Each of these has to specialized based on the program. The assembler performs an analysis to determine the size of the instruction memory and the required instructions for the PE and control unit. Based on this information the template-driven generator creates specialized instances for every enclosing module. Currently, the data paths are 8 bits, but we are working on an RC with a configurable data path width.

C. *Interconnect and Toplevel Generation*

The final step of the generator instantiates all the generated RCs and creates the required interconnections. This is the most complicated step in the process as the generator must infer the control nets as well as wiring up the data nets, and must also correctly strap unused nets to ground or leave them open.

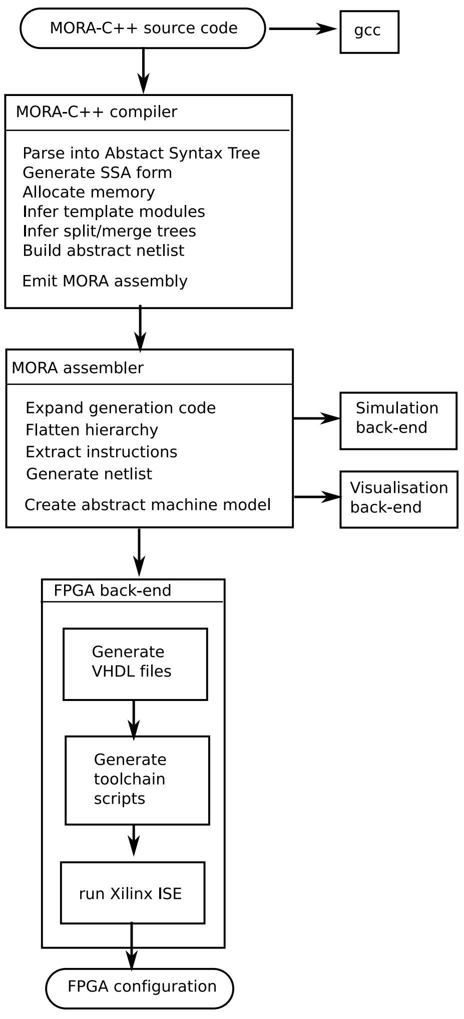

Figure 2. MORA toolchain

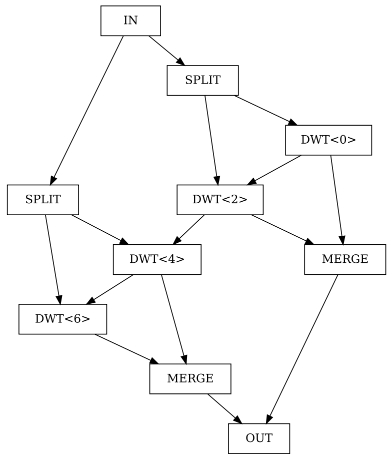

Figure 3. ADG diagram for the DWT algorithm

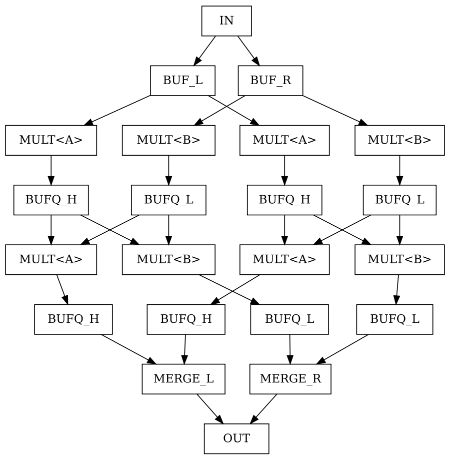

Figure 4. ADG diagram for the DCT algorithm

D. *FPGA Configuration Generation*

Using a template for a Xilinx ISE project in Tcl, the assembler finally builds the complete project including synthesis, place and route and bitfile generation. All the steps are completely automated, resulting in a truly high-level FPGA programming solution.

VIII. Results and Discussion

We implemented the two image processing algorithms describe above, Discrete Wavelet Transform (DWT) and Discrete Cosine Transform (DCT), as exemplars.

A. *Overview of Exemplars*

Fig. 3 (generated by the visualization backend) shows the connectivity (acyclic directed graph, ADG) of the RCs for the DWT algorithm. The implementation takes 55 cycles to compute the 24 arithmetic instructions of the DWT.

The ADG for the DCT algorithm as generated by the MORA assembler is shown in Fig. 4 and illustrates the 4 parallel data paths. Two different implementations, one for high throughput (DCT^F) and another for better resource utilization (DCT^S) were implemented for the DCT on the Virtex-4 LX200. The DCT^F and DCT^S compute the results in 110 and 200 clock cycles respectively running at frequencies around 100 MHz. The Virtex-4 LX200 can accommodate at least twenty five DCT^Ss or twelve DCT^Fs with little decrease in operating frequencies.

B. *Abstraction Level*

To demonstrate the level of abstraction of MORA-C++ for programming FPGAs, Table I shows the number of lines of source code (obtained using the cloc program[1]) for implementations of DWT and DCT. From a very small source code file (37 lines of on average 30 characters), the MORA assembler produces 124 VHDL files totaling 8815 lines of VHDL source code. Thanks to the powerful matrix operations, the DCT is even smaller: 18 lines of on average only 20 characters, compared to 16803 lines of VHDL code.

C. *Resource Utilisation and Performace*

Table II illustrates the area and performance results of the MORA array on a Xilinx Virtex 4 LX200 for both algorithms.

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

	Language	files	blank	comment	code
DWT	MORA-C++	1	6	1	37
	VHDL	124	1881	1924	8815
DCT	MORA-C++	1	6	1	18
	VHDL	214	3390	3748	16803

Table I
SOURCE CODE LINE COUNT FOR MORA-C++ AND VHDL IMPLEMENTATIONS OF DCT AND DWT

	Single RC	DWT	DCT	
			Smaller (S)	Faster (F)
Slice count	479	1836	3368	6867
BRAM count	1	10	22	44
Latency (cycles)	NA	55	200	110
Clock Freq(MHz)	NA	70	100	95

Table II
IMPLEMENTATION RESULTS ON VIRTEX-4 LX200

Although the project is still in an early stage and many optimizations have not yet been implemented, the performance of the MORA soft array is already comparable to other high-level FPGA programming tools. For example, Yankova et al [12] report a DCT implementation using their DWARV VHDL generator. The slice count is 3307, the clock frequency 100MHz. They do not report cycle counts but compare to a PowerPC implementation and report a speed-up of 9.74. For comparison we used the highly optimized DCT implementation by the Independent JPEG Group and compiled it using gcc. The average cycle count was 2550 cycles. Relative to this figure, our two DCT implementations DCT^S and DCT^F achieves a speed-up of 12.7 times and 23.2 times respectively. El-Araby et al [13] report on a comparison of different high-level programming tools and one of their test benches is a DWT implementation. They report percentage of slices utilized for a Virtex-II Pro XC2VP50. Their Impulse-C implementation utilized 17% or 4015 slices, their Mitrion-C version 6613 slices. Both run at 100MHz. The reported throughout is respectively, 1.35 and 375 MB/s. By comparison, our MORA implementation utilizes 1836 slices and has a throughput (at 70MHz) of 10 MB/s; the key difference is that our current DWT implementation has 8-bit I/O whereas the Cray XD1 board used by [13] has a 64-bit I/O interface. We could easily fit 8 parallel DWT pipelines on the XC2VP50.

It should be noted that implementations using commercial tools such as Impulse CoDeveloper achieve similar clock frequencies as the MORA approach. However, as also noted in [12], these commercial tools, although accepting a C-like input language, do in fact require advanced hardware knowledge. MORA does not require in-depth hardware knowledge. The MORA-C++ API completely abstracts the implementation details of the MORA soft array architecture.

IX. CONCLUSION

In this paper we have presented a novel approach to high-level FPGA programming of multimedia applications. We have introduced the MORA soft processor array and MORA-C++, a C++-embedded Domain-Specific Language for high-level programming of the MORA platform. The advantages of MORA-C++ over other high-level FPGA programming approaches are that it is embedded in popular, powerful language and requires no knowledge of FPGAs. Our initial results demonstrate that the generated designs are very resource-efficient and provides high throughput for the multimedia exemplars on a Xilinx Virtex-4 FPGA. These results clearly indicate that the concept has great potential for high-level programming of multimedia processing applications on FPGAs. As future work, the RCs will be optimised to achieve higher operation frequency and higher throughput with a smaller footprint; the datapath width will be made configurable; and we will continue work on the compiler for MORA-C++.

REFERENCES

[1] C. Sullivan, A. Wilson, and S. Chappell, "Using C based logic synthesis to bridge the productivity gap," in *Proceedings of the 2004 conference on Asia South Pacific design automation: electronic design and solution fair*, pp. 349–354, IEEE Press Piscataway, NJ, USA, 2004.

[2] S. M. F. M. G. P. S. C. Antola, A., "A novel hardware/software codesign methodology based on dynamic reconfiguration with impulse c and codeveloper," pp. 221 –224, Feb. 2007.

[3] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language," in *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, p. 49, IEEE Computer Society Washington, DC, USA, 2000.

[4] J. Tripp, K. Peterson, C. Ahrens, J. Poznanovic, and M. Gokhale, "Trident: an FPGA compiler framework for floating-point algorithms," in *Proceedings of the 15th International Conference on Field Programmable Logic and Applications (FPL 2005)*, pp. 317–322.

[5] V. V. Kindratenko, R. J. Brunner, and A. D. Myers, "Mitrion-c application development on sgi altix 350/rc100," in *FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, (Washington, DC, USA), pp. 239–250, IEEE Computer Society, 2007.

[6] S. Craven, C. Patterson, and P. Athanas, "A Methodology for Generating Application-Specific Heterogeneous Processor Arrays," in *HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES*, vol. 39, p. 251, Citeseer, 2006.

[7] B. J. Jasionowski, M. K. Lay, and M. Margala, "A processor-in-memory architecture for multimedia compression," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 15, no. 4, pp. 478–483, 2007.

[8] S. R. Chalamalasetti, S. Purohit, M. Margala, and W. Vanderbauwhede, "Mora - an architecture and programming model for a resource efficient coarse grained reconfigurable processor," *Adaptive Hardware and Systems, NASA/ESA Conference on*, vol. 0, pp. 389–396, 2009.

[9] W. S. M. Chalamalasetti, S.;Vanderbauwhede, "A low cost reconfigurable soft processor for multimedia applications: Design synthesis and programming model," in *19th IEEE International Conference on Field Programmable Logic and Applications (FPL09)*, pp. 534–538, IEEE, 2009.

[10] W. Vanderbauwhede, M. Margala, S. R. Chalamalasetti, and S. Purohit, "Programming model and low-level language for a coarse-grained reconfigurable multimedia processor," in *ERSA*, pp. 195–201, 2009.

[11] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, p. 490, 1991.

[12] Y. Yankova, K. Bertels, G. Kuzmanov, G. Gaydadjiev, Y. L. 0004, and S. Vassiliadis, "Dwarv: Delftworkbench automated reconfigurable vhdl generator.," in *FPL* (K. Bertels, W. A. Najjar, A. J. van Genderen, and S. Vassiliadis, eds.), pp. 697–701, IEEE, 2007.

[13] E. El-Araby, M. Taher, M. Abouellail, T. El-Ghazawi, and G. Newby, "Comparative analysis of high level programming for reconfigurable computers: Methodology and empirical study," in *Proceedings of the 3rd Southern Conference on Programmable Logic (SPL)*, pp. 99–106, 2007.