Overview of the tool-flow for the Montium Processing Tile

Gerard J.M. Smit, Michèl A.J. Rosien, Yuanqing Guo, Paul, M. Heysters
University of Twente dept. EEMCS
PO Box 217, Enschede, the Netherlands
smit@cs.utwente.nl

Abstract.

This paper presents an overview of a tool chain to support a transformational design methodology. The tool can be used to compile code written in a high level source language, like C, to a coarse grain reconfigurable architecture. The source code is first translated into a Control Data Flow Graph (CDFG). A Control Dataflow Graph contains not only the dataflow operations (e.g. arithmetic or logical operations on data) but also control flow operations (e.g. operators for loop and if then else constructs). The CDFG is minimized using a set of behavior preserving transformations such as dependency analysis, common sub-expression elimination, etc. After applying graph clustering, scheduling and allocation transformations on this minimized graph, it can be mapped onto the target architecture.

Keywords: tools for reconfigurable, energy-efficient, coarse grain reconfigurable, transformational design.

I. INTRODUCTION

In the CHAMELEON[1] project we are designing a heterogeneous reconfigurable System-On-a-Chip (SOC). This SOC contains a general-purpose processor (ARM core), a bit-level reconfigurable part (embedded FPGA) and several word-level reconfigurable parts (MONTIUM tiles; see Section III). We believe that in future 3G/4G terminals heterogeneous reconfigurable architectures are needed. The main reason is that the efficiency (in terms of performance or energy) of the system can be improved significantly by mapping application tasks (or kernels) onto the most suitable processing entity. The design of the above-mentioned architecture is useless without a proper tool chain supported by a solid design methodology. Usually applications can be structured as a set of processes or threads that communicate (or synchronise) via channels. These processes have to be executed on the most suitable execution tile (e.g. general purpose CPU, Montium, FPGA, etc.) and the on-chip network has to support the communication patterns of the application.

[1] This research is supported by the PROGram for Research on Embedded Systems & Software (PROGRESS) of the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the technology foundation STW.

In this paper we will concentrate on the tools, supporting a transformational design method, for 'compiling' one such process to a coarse-grain reconfigurable processing tile.

With transformational design we mean that the specification can be transformed through a series of verifiable steps to a final system implementation model that meets the system requirements and constraints imposed by the designers [3]. These steps can be either semantics preserving or non-semantics preserving. The latter introduces design decisions imposed by the designer. In our tool chain we start with semantics preserving transformations, but as we approach the target architecture also non-semantics preserving transformations will be needed. The main idea is that the transformation result of each step is sufficiently close to the previous expression that the effort of verifying the transformation is not excessive. The benefits of this approach are:

- Each step in the refinement brings the design one step closer to the implementation.
- The final implementation is either a true implementation of the initial specification which is guaranteed by the use of a set of semantics preserving transformation rules, or can be proven to be correct for certain assumptions w.r.t. non-semantics preserving transformations, which are introduced by design decisions.
- A transformational approach made up of a sequence of small steps is very effective and more efficient than theorem proving which is usually very difficult because of the gap between implementation and specification is much larger than the gaps between each transformation step.

The difficulty in applying a pure transformational approach to a realistic application is the large amount of transformations that have to be performed. To cope with this problem, we first decompose a large system in smaller parts that can be handled by a transformational system. This is not a real drawback as decomposition has to be done anyway because only the computational intensive parts will be executed in reconfigurable hardware. Examples of parts that can be transformed in reasonable time with our tools are: FIR filters, FFT and DCT algorithms (see section V). The incorporation of this approach into the design methodology will offer opportunities to improve the design process.

The objective of this paper is to show that a practical transformational design method with the support of highly automated mechanized transformations can be used to transform processes, written in a high level language such as C, to configurations for a reconfigurable platform. First, a small introduction of our coarse grain reconfigurable architecture is presented in section III. Next we show how a process, written in a high level language such as C, will be translated into a Control Dataflow Graph (CDFG, see section IV). The resulting graph will be simplified using various behavior preserving transformations and finally will be clustered, scheduled and allocated onto a MONTIUM tile.

II. RELATED WORK

Pioneering work in transformation systems was done by Burstall and Darlington [3]. Their system transformed applicative recursive programs to imperative ones and their ideas have heavily influenced today's transformation systems. In our approach we do not start from an applicative language but start with the imperative language such as C. The reason to choose the 'low-level' high-level language C is that the majority of DSP algorithms is specified in C or Matlab. The system designers often start with an executable C or Matlab reference specification, and the final implementation are be verified against this reference specification.

In [2] a formal systems design methodology is presented. Unfortunately, like more work on transformational design, this work only presents a formal framework, without tools to mechanize the transformations. These tools are necessary because of the large amount of transformations that need to be performed. In [4] a compiler framework is given for mapping applications to a coarse-grained reconfigurable architecture. In their approach they use a language SA-C an expression-oriented single assignment language. Their language does not support pointers and recursion.

Today most compilers for reconfigurable computers often rely on a library of highly optimized and hand-mapped functions. (e.g. FIR, DCT or FFT operations). Examples of compilers that are specially designed for a reconfigurable architecture can for instance be found in [5], [6].

In our approach, however, we start with a specification in C, even the frequently used kernel operations are specified in C. We do not rewrite this specification to an implementation oriented C with special library calls, but stick to the original C specification. This means that we can avoid the tedious hand-mapping work, and that we can use a C specification for the whole design. This also avoids the error-prone rewriting of the C specification to an implementation language with special implementation dependent library calls. On the other hand it also means that our approach has to find an efficient implementation. In this process also dead code has to be eliminated and inefficient implementations have to be pruned. Similar work has been done by IMEC on a 'concept' called the 'software washing machine' [9].

The work presented in this paper is a logical continuation of the work presented in [7]. In this paper we present a tool that supports the transformational design methodology. This transformational design methodology has the following properties:

- It supports design space exploration and allows final implementations in software and hardware.
- It is based on a formal model in which structure, behavior, memory and time can be expressed.
- The model supports a 'framework' for describing specification, design and transformations. Model and language support the specification in terms of a process model [8] with processes described in C and the implementation level in terms of primitive operations like adders and registers.
- The model is compositional, i.e. when a part of the design description is replaced by a different part with the same external behavior, the external behavior of the total design remains unchanged.
- The design process is based on local, small behavior-preserving transformations. More complex transformations can be built from small ones.
- The model provides means to reduce the complexity: i.e. hierarchy of functions (like procedures in a programming language), hierarchy of data (records and arrays), and repetition (loops and recursion).

III. THE RECONFIGURABLE MONTIUM ARCHITECTURE

The tools we present in this paper were primarily designed for the MONTIUM architecture [1], but we believe that considerable parts of the tools can be used for other reconfigurable architectures as well. The MONTIUM targets the 16-bit digital signal processing (DSP) algorithm domain. Figure 1 depicts a single MONTIUM processing tile. At first glance the MONTIUM architecture bears a resemblance to a VLIW processor. However, the control structure of the MONTIUM is very different.

The MONTIUM Tile Processor is tailored for a specific algorithm domain. The algorithms that we used to refine the datapath of the MONTIUM tile include: FIR filters, FFT, correlation, turbo decoding, DCT, matrix multiplication and linear interpolation. Figure 1 reveals that the hardware organization within a tile is very regular. The five identical ALUs (ALU1…ALU5) in a tile can exploit spatial concurrency to enhance performance. This parallelism demands a very high memory bandwidth, which is obtained by having 10 local memories (M01…M10) in parallel. The small local memories are also motivated by the locality of reference principle to obtain energy-efficiency.

The ALUs are rather obese; they can perform a quite complex operation in one clock cycle e.g. *(max(A,B)*D + C)*.

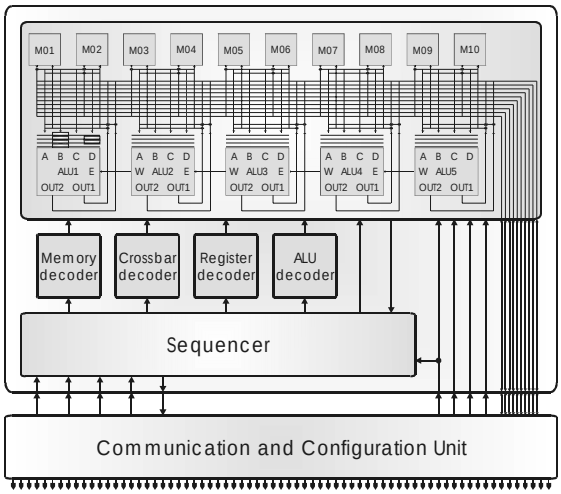

Figure 1: The MONTIUM processing tile.

The datapath has a width of 16-bits and the ALUs support both signed integer and signed fixed-point arithmetic. The ALU input registers provide an even more local level of storage. Locality of reference is one of the guiding principles to obtain energy-efficiency in the MONTIUM. A relatively simple sequencer controls the entire PPA.

Each local memory is 16-bit wide and has a depth of 512 positions, which adds up to a storage capacity of 8Kbit per local memory. A memory has only a single address port that is used for either reading or writing. A reconfigurable Address Generation Unit (AGU) accompanies each memory. It is also possible to use a memory as a lookup table for complicated functions that cannot be calculated using an ALU, such as a sine function or division with a constant.

IV. AUTOMATED MAPPING OF ALGORITHMS ONTO THE MONTIUM

Architectures such as the MONTIUM with a reconfigurable instruction set pose a challenge for developing a compiler. One of the reasons is that the instruction set is reconfigurable. Yet, the availability of high-level design entry tools is essential for the (economic) viability of any reconfigurable architecture. It is essential that new applications can be implemented quickly and at low cost. The MONTIUM has a straightforward control mechanism that has moderate overhead and enables the development of a compiler. This paper presents a C compiler, based on a transformational framework that is currently being developed for the MONTIUM architecture.

The design flow

Figure 2 shows the entire C to MONTIUM design flow. First the system checks whether a kernel (C code) is already in the library, if so the MONTIUM configurations can be generated directly. Otherwise, a high-level C program is translated into an intermediate CDFG Template language (With the correct templates other high-level languages, such as Java, Pascal or Matlab, can be translated to this format). A Hierarchical CDFG can then be build from this intermediate language. Subsequently this graph is cleaned: e.g. dead code elimination and common sub-expression elimination transformations are performed. At this point the source language specific part ends and the architecture dependent part starts.

Next clusters are generated from the CDFG. These clusters are scheduled and allocated to the 5 physical ALUs of a MONTIUM tile. After that the MONTIUM configurations can be generated. These configurations can also be generated manually by the Montium editor.

This editor was used to generate the first Montium programs. Currently this editor is used to inspect the automatically generated MONTIUM code. Finally, the configurations can be tested on the MONTIUM cycle true simulator, or can be run on an actual MONTIUM tile. In the next sections the different sub-tools of the chain will be discussed in more detail.

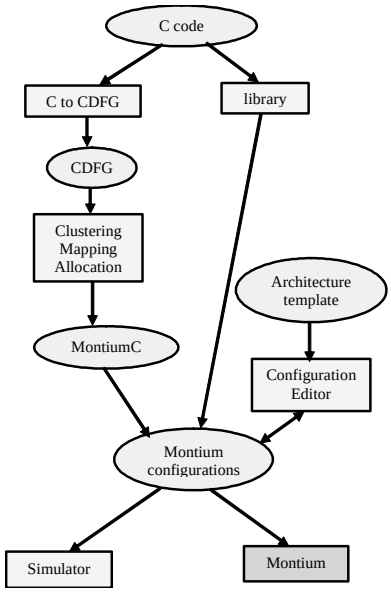

Figure 2: MONTIUM tool flow

Step 1: Translating the source code to a CDFG

To be able to map a C program to a CDFG it is necessary to map the linear, random access memory model of C to a model that can be used in a CDFG. For this reason, a mathematical abstraction of a memory model, called the *state space*, is introduced in [7]. The state space is a set of tuples: {(*ad*,*da*),(*ad*,*da*),…}. A tuple consists of an *ad* field,

which represents the address, and a *da* field, which represents the data at that address. This data can be anything, including a tuple of this type again. In this way structs, pointers and arrays can be specified.

Interaction with the state space is, in principal, done with two primitive graphs called **Store** and **Fetch**.

The store graph is used to add a tuple to the state space. It has three inputs: The incoming state space (*ssin*), an address (*ad*) and the data (*da*) to be written on this address. It has a single output, the modified state space
$ssout = ssin \cup \{(ad,da)\}$.

The fetch graph is used to read data from an address. It has two inputs: The incoming state space (*ssin*) and the address from which to read (*ad*). The output is the data at this address (*da*). A fetch operation does not modify the state space.

Two additions to the before mentioned operations have been defined to be able to support scope levels/local variables more easily, a **Delete** operation (Del) and a **Define** operation (Def). The delete operation signals when a specific tuple is no longer valid. The define operation doesn't actually do anything but it signals when the lifetime of a specific tuple begins. The **Define** and **Delete** operations are very helpful when performing a dependency analysis or other simplifications.

Once the CDFG is built, a number of behavior preserving transformations can be performed. Currently more than 30 transformations are supported for example:

- Hierarchy removal
- Common sub expression elimination
- Dead code elimination
- Constant propagation
- Dependency analysis.

Some transformations are already standard available in compilers like constant propagation and common sub-expression elimination etc. However, the same transformation framework can also be used for more advanced transformations such as implementation dependent clustering transformations, loop unrolling etc.

Operators

All unary and binary C operators (like +, -, *, /, <<, >> etc.) can be directly implemented to a node in the CDFG.

Iteration & Jumps

Iteration is modeled by recursion. A **while** statement, for example, calls itself recursively to go to the next iteration instead of actually 'jumping' back to the start (see Figure 4 for the while template).

In Figure 3 $0 denotes placeholder for the graph for the while condition, $1 the while body and $R denotes the recursive call of the while statement. Depending on the result of $0 the MUX takes another iteration or exits the while loop. Normal jump statements such as **goto** and **return** can also be modeled by recursion but lead to very complex unreadable graphs.

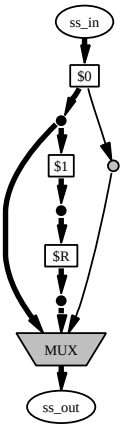

Figure 3: Template of a while statement

The graphs that illustrate this are too complex to show here. Figure 5 shows a screenshot of the C to CDFG tool.

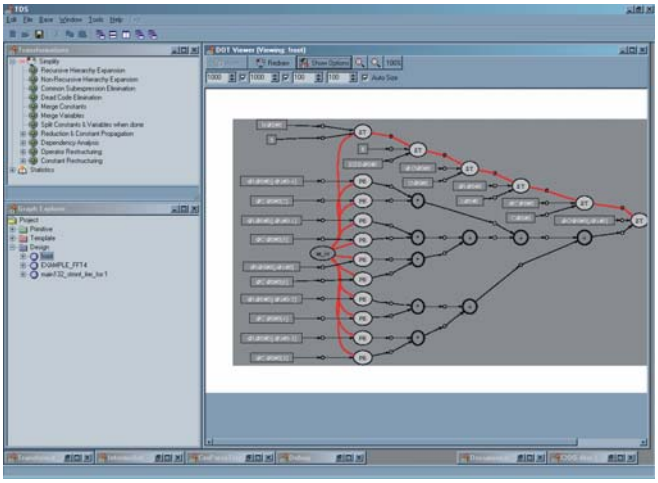

Figure 4: Screen shot of the C to CDFG Tool

When the CDFG has been simplified as much as possible, a so-called clean graph is obtained. In this step inefficiencies of the source code (e.g. dead-code or unevaluated constants) can be removed. This is similar to the approach presented in [9]. Up to this point the tool-flow is largely architecture independent, although some steps are architecture driven e.g. controlling the amount of loop unrolling.

Step 2: Clustering and allocation

The next steps in the flow are highly architecture dependent. First the CDFG is clustered. These clusters will eventually constitute the 'instructions' of the reconfigurable processor. Examples of clusters are: a butterfly operation for a FFT and a MAC operation for a FIR filter. Clustering is critical step as these clusters (='instructions') are application dependent

and should match the capabilities of the processor as close as possible. In our Montium processor this means that we choose clusters such that they can be executed on one ALU in one clock cycle. More information on our clustering algorithm can be found in [10]. Next the clustered graph is scheduled taking the number of ALUs (in our case five) into account. Finally, the resources such as registers, memories and crossbar are allocated.

In this phase also some Montium specific transformations are applied, for example, conversion from array index calculations to Montium AGU (Adress Generation Unit) instructions, transformation of the control part of the CDFG to sequencer instructions.

Once the entire graph has been clustered, scheduled, allocated and converted to the Montium architecture, the result is outputted to MontiumC, a cycle true 'human readable' description of the configurations. This description, in an ANSI C++ compatible format, can be compiled with a standard C++ compiler. When the compiled code is run, the MONTIUM configurations are generated. After compilation, the resulting code can simulated or run on the actual Montium processor.

An example of a piece of MontiumC code is shown below.

```
CLOCK()
  .INC(MEM(PP1,LEFT),1)
  .MOV(MEM(PP1,LEFT),REG(PP1,A0))
  .MOV(MEM(PP1,RIGHT),REG(PP1,C0))
  .MOV(MEM(PP5,RIGHT),REG(PP5,C0))
  .MOV(OUT1(PP1),REG(PP1,D0))
  .MOV(OUT1(PP1),REG(PP2,D0))
  .USE(REG(PP4,D0))
  .USE(REG(PP5,A0))
  .USE(REG(PP5,C0))
  .USE(REG(PP5,D0))
  .CALC(Out1_A_Times_C_Plus_D)
  .NOP()
;
```

It shows for a particular clock cycle what data moves will be done (.MOV); what registers are used (.USE); what operation are done (.CALC); what address generation units do (.INC) and finally what the sequencer does (.NOP).

Step 3: Simulation

The tool chain also supports a graphical configuration editor. Figure 5 shows the Montium Configuration Editor which was initially used to manually create configurations for the Montium. It can also be used to transform code created by MontiumC to configurations which can be used by the Montium simulator.

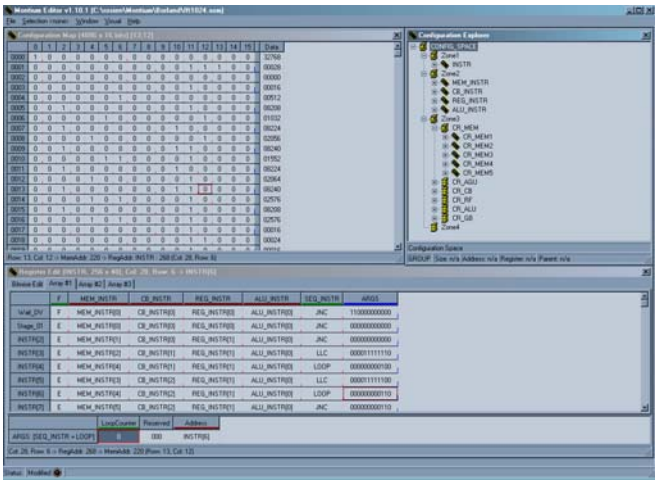

Figure 5: Screenshot Montium Configuration Editor

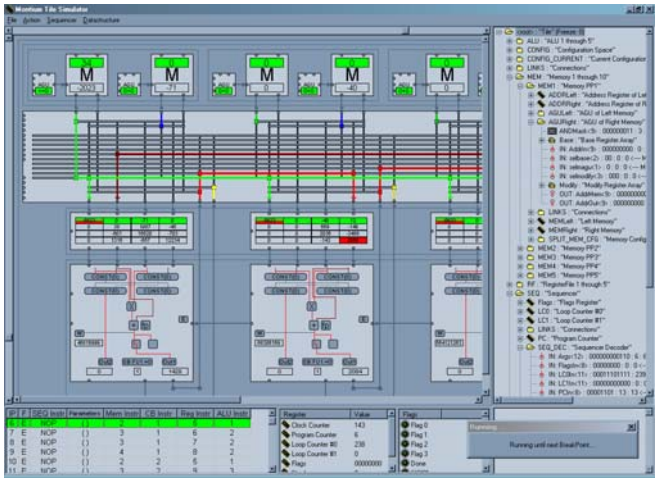

Figure 6: Screenshot Montium Tile Simulator

Figure 6 shows the Montium Tile Simulator which is able to fully simulate a Montium Tile. In the simulator after each clock cycle all memories, registers and busses can be inspected. Furthermore for each clock cycle the selected ALU operation is depicted. Figure 6 shows two views in the simulator a graphical view (on the left) and a hierarchical view (on the right). On the bottom the status of the sequencer is shown.

V. EXAMPLE

As an example, we will show how a C program (FIR5 filter) is converted to configurations. Figure 8 shows the C code we will use in the example.

```
int Input[SIZE], Output[SIZE];
int C0, C1, C2, C3, C4;
int T0 = 0, T1 = 0, T2 = 0, T3 = 0, T4 = 0;
void main() {
  for (int i = 0; i < SIZE; ++i) {
    T4 = T3 + Input[i] * C4;
    T3 = T2 + Input[i] * C3;
    T2 = T1 + Input[i] * C2;
    T1 = T0 + Input[i] * C1;
```

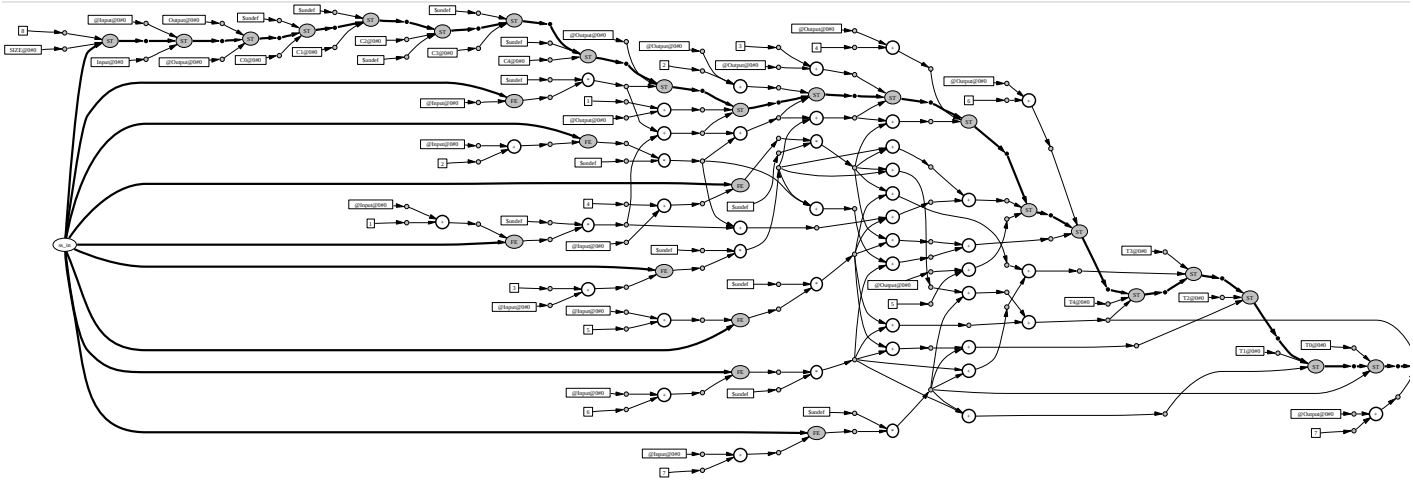

Figure 7: Completely unrolled (N=8)

```
    T0 = Input[i] * C0;
    Output[i] = T4;
  }
}
```

Figure 8: FIR5 Filter C Code

To show that we also support pointers Figure 10 shows the same code but now with pointers. This code results in exactly the same CDFG.

```
int Input[SIZE], Output[SIZE];
int C0, C1, C2, C3, C4;
int T0 = 0, T1 = 0, T2 = 0, T3 = 0, T4 = 0;
void main() {
  int* op = &Output[0];
  for (int* ip = &Input[0];
       ip <= &Input[SIZE-1]; ++ip)
  {
    T4 = T3 + *ip * C4;
    T3 = T2 + *ip * C3;
    T2 = T1 + *ip * C2;
    T1 = T0 + *ip * C1;
    T0 = *ip * C0;
    *op = T4;
    ++op;
  }
}
```

Figure 10: FIR5 filter C code with pointers

The resulting CDFG could be fully expanded and simplified to obtain a flat CDFG without loops. Figure 7 shows a fully unrolled graph for SIZE=8. Since a fully expanded graph for SIZE > 8 is generally too large to display, we usually use one or a few unrolled iteration(s) of the **for** statement. The resulting flat CDFG of one iteration of the **for** statement is shown in Figure 9**.** The thick arrows denote the state-space.

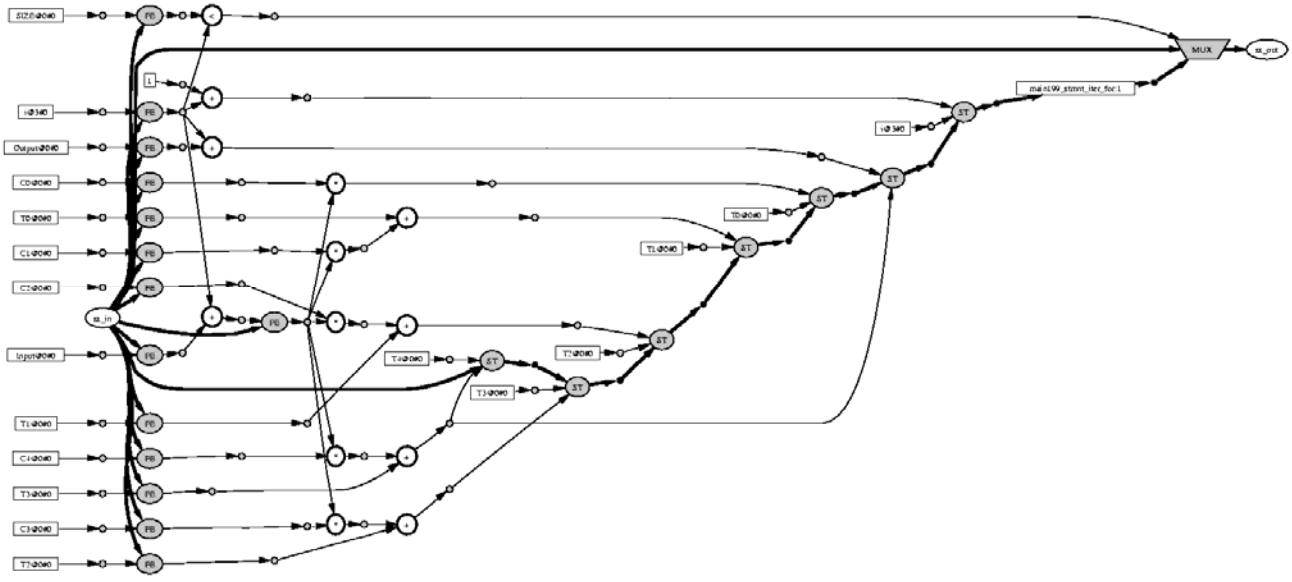

Figure 9: FIR5 one iteration of the for loop

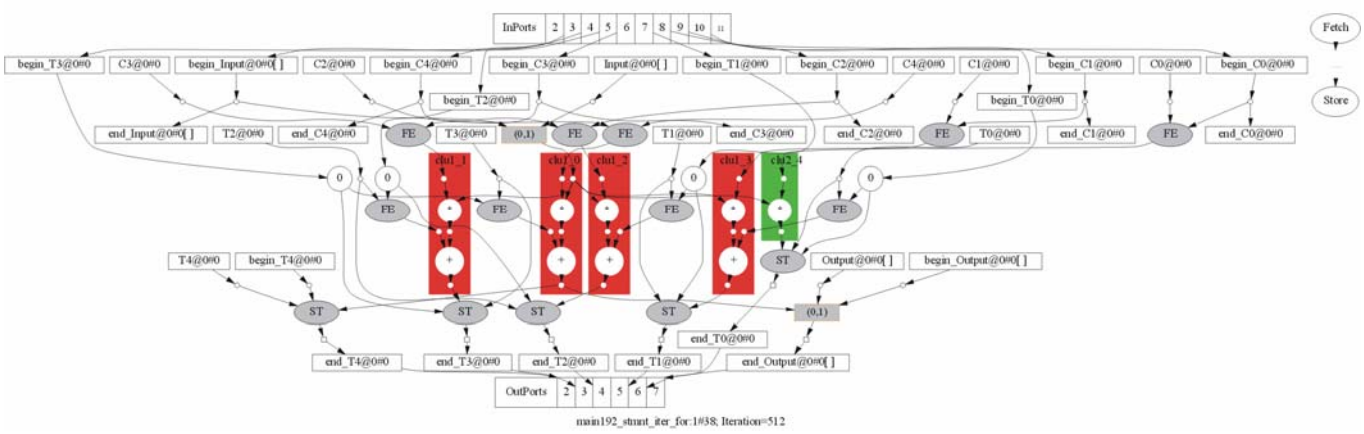

Figure 11: Clustered graph on 5 ALUs

The large square on the right side of the graph is the recursive call. This graph can then be clustered for the Montium architecture. Figure 11 shows the result. The five larger squares denote the five clusters that can be executed in one clock cycle on five ALUs. Because of the size of the resulting graphs only a small example is given. In the results section more statistical figures are given for larger graphs.

VI. RESULTS

To fully simplify a CDFG, a huge number of transformations have to be applied. Table 1 gives an indication of how many transformations have to be applied to fully simplify/expand a CDFG from a specific algorithm. In practice a systems programmer not always wants a complete flat CDFG of his algorithm. For instance the complete expansion of a 32-point FFT will result in a graph of 1224 nodes. Therefore, the user has the possibility to do partial loop unrolling; which means that he can specify how often a certain loop has to be unrolled. In practise the amount of unrolling is determined by the amount of parallelism the hardware architecture supports.

To give an indication of the speed: the 64-point complex FFT generates a graph of 2744 nodes. It takes 3847s to generate this on a Pentium4 1.7 GHz. The FFT16 took 62s.

Although we do support nearly complete C (we exclude function pointers), we normally use a subset of C (C without goto's and recursion) to speed up the transformations. The types of DSP algorithms we are targeting at are usually free of goto's and recursion.

Table 1: Number of Transformations

	FFT32	FIR64	2D_DCT8	FFT16	FFT64
Total # of nodes in resulting graph	1224	528	1367	544	2744
Hierarchy Removal	1592	186	1464	1479	1643
Common Sub expression Elimination	1066	128	84	449	2438
Dead Code Elimination	7337	430	787	3466	16230
Constant Propagation	2691	376	787	1487	5487
Dependency Analysis	5898	707	1530	2486	13494

VII. CONCLUSION

In this paper we presented a tool chain that supports a transformational design method to transform processes written in a high level language, such as C, to a CDFG (Control Data Flow Graph). To our knowledge this is the first system that can translate complete C (including pointers, functions and recursion but with the exception of function-pointers) to a CDFG. After clustering, scheduling and resource allocation this graph can be mapped onto a coarse-grain reconfigurable processing tile. Using the transformations a C program can be transformed into a flat Control Data Flow Graph. However, the user (system-designer) can also decide to partially unroll the loops and end up with a hierarchical graph. The transformations also can be used to optimize the system with respect to certain criteria e.g. energy efficiency or execution speed.

Future work

As we mentioned in the conclusion we can handle full C with the exception of function pointers. The next step will be to introduce function pointers in our model. This opens up the way to C++ including classes.

REFERENCES

[1] Heysters P.J.M., Smit G.J.M., Molenkamp E.: "Montium – Balancing between energy-efficinecy, flexibility and Performance", *Proceedings of Conference on Engineering of Reconfigurable Systems and Algorithms 2003*, pp 235-242, Las Vegas, Nevada, 2003

[2] W. Wu, I. Sander, A. Jantsch: "Transformational System Design Based on a Formal Computational Model and Aketletons", *Forum on Design Languages (FDL2000)* 4-8th September 2000.

[3] R.M. Burstall, J.Darlington: "A Transformation System for Developing Recursive Programs", *Journal of the ACM*, 24(1), pp 44-67, Jan 1977.

[4] Venkataramani G., Kurdahi F., Bohm W. "A compiler Framework for Mapping Applications to a coarse-grained Reconfigurable Computer Architecture", *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, Atlanta, Georgia, 2000

[5] C. Ebeling et al: "Mapping applications to the RaPiD Configurable Architecture" IEEE Symposium on FPGAs for Custom Computing Machines, Napa Vallay, CA, 1997

[6] S.C. Goldstein et al: "PipeRench: A reconfigurable Architecture and Compiler", *IEEE Computer*, vol 33, pp 70-77, 2000.

[7] Th. Krol, B.S. Visser, "High-level Synthesis based on Transformational Design", *Internal report*, University of Twente, Enschede

[8] G. Kahn: "The semantics of a simple language for parallel programming", *Information Processing* 74, pp 471-475, 1974

[9] Catthoor F.: "the Software Washing Machine", keynote Proceedings of the 15th international symposium on System Synthesis 2002, Kyoto, Japan, October 2002

[10] Y. Guo, G.J.M. Smit, P.J.M. Heysters, H. Broersma "A Graph Covering Algorithm for a Coarse Grain Reconfigurable System", Proceedings of LCTES 2003, pp. 199-208, San Diego, USA, June, 2003