Profiling, Compilation, and HDL Generation within the hArtes Project

Koen Bertels, Georgi Kuzmanov, Elena Moscu Panainte, Georgi Gaydadjiev,
Yana Yankova, Vlad Mihai Sima, Kamana Sigdel, Roel Meeuws, Stamatis Vassiliadis
Computer Engineering Laboratory, EEMCS
Delft University of Technology, The Netherlands
http://ce.et.tudelft.nl
Email: {K.L.M.Bertels, G.K.Kuzmanov, E.Moscu-Panainte, G.N.Gaydadjiev,
Y.D.Yankova, V.M.Sima, K.Sigdel, R.J.Meeuws, S.Vassiliadis}@tudelft.nl

Abstract

The hArtes project addresses optimal and rapid design of embedded systems from high-level descriptions, targeting a combination of embedded processors, digital signal processing, and reconfigurable hardware. In this paper, we present three tools from the hArtes toolchain, namely profiling, compilation, and HDL generation tools, that facilitate the HW/SW partitioning, co-design, co-verification, and co-execution of demanding embedded applications. The described tools are provided by the DelftWorkBench framework.

1. Introduction

The hArtes project addresses research and development issues of embedded systems. It investigates hardware/software integration and builds a general-purpose toolchain, which accepts applications written in a multiplicity of high-level algorithm descriptions and it produces semi automatically a "best fit" mapping of such applications into a heterogeneous reconfigurable system. The tool chain is intended to provide a fast development trajectory from application coding to the design of a reconfigurable embedded computing system.

The hArtes toolchain is composed of the following three toolboxes: 1) *algorithm exploration and translation* ToolBox; 2) *design space exploration (DSE)* ToolBox; and 3) *system synthesis (SysSyn)* ToolBox. The input of this tool-chain is a high level application algorithm, described in one of several supported formats and languages, such as, Matlab or handcrafted C. The internal representation of the application algorithms is C code, annotated with pragmas by the tools in the toolchain. The objectives of each hArtes ToolBox can be summarized as follows:

- The *Algorithm exploration and translation ToolBox* provides tools with two basic functionalists. They assist the designers to instrument and possibly improve the input algorithm at the highest level of abstraction. Also, they translate the input algorithms, described in different formats and languages (e.g., Simulink or graphical entry), into a unified internal description in the C language.
- The *Design space exploration ToolBox* provides an optimal hardware/software partitioning of the input algorithm for each reconfigurable heterogeneous system considered. A set of profilers and cost estimators employ specific metrics to evaluate a particular mapping of a candidate application on the particular reconfigurable heterogeneous system with respect to performance, hardware complexity, power consumption, etc. The input of the DSE ToolBox is the C description of the application algorithm, annotated with specification directives from the algorithm exploration and translation ToolBox, and models of the reconfigurable heterogeneous system platform. The DSE output is an optimized partitioning of the application algorithm for the considered reconfigurable heterogeneous system.
- The *System synthesis ToolBox* processes the optimized partitioning of the application algorithm provided by the DSE ToolBox as its input. The output comprises all generated files, required to map the application algorithm on the components of the considered reconfigurable heterogeneous system with respect to its partitioning, i.e., program executables, configuration bitstreams, memory images, etc.

In this paper, we address three specific tools from the

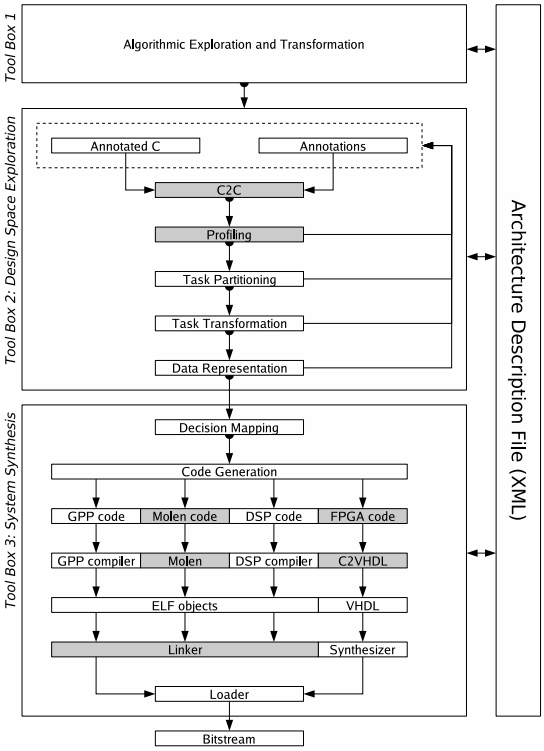

Figure 1. hArtes Design Flow

hArtes toolchain, namely the profiling toolset from the DSE toolbox, and the compilation and HDL generation tools from the SysSyn Toolbox. All these three tools have been developed within the DelftWorkBench [2] - a semi-automatic tool platform for integrated hardware-software co-design targeting heterogeneous computing systems containing reconfigurable components, which provide the required support for the Molen Programming Paradigm [18]. The hardware support of this paradigm is provided by the Molen machine organization [20]. In the hArtes context, the DelftWorkBench tools are further extended to support digital signal processors and application specific hardware, besides the reconfigurable co-processors support. The positions within the overall design flow of the tree tools discussed are presented in Figure 1.

During the profiling phase, relevant execution and data storage information is collected. In addition, preliminary estimations of the hardware and the software costs of the application's kernels are performed. In the next step, a set of optimizations and parallelizing transformations are applied on the application. The output of the toolbox is annotated C code. In the next tool box, the final partitioning of this code is derived. The compiler processes the code to be executed on the GPP, placing the necessary configuration and hardware execution instructions. In addition, intra- and inter-procedural optimizations addressing the configuration and data transfer latency are applied. Simultaneously, the HDL generation unit provides the VHDL designs of the selected kernels. As final step, the outputs of the compiler and the HDL generation unit are linked and the execution is carried on the MOLEN platform. The communication between the tools and the feedback loops within the chain are secured through an architecture description file.

The organization of the paper is the following: Section 2 presents the related work. Section 3 presents the architecture description file used by the different tools in the toolchain. Section 4 discusses the profiling framework within the hArtes project. Section 5 and section 6 present the MOLEN backend compiler and the DWARV HDL generation tool, respectively. Section 7 concludes the paper.

2. Related Work

Profiling and Estimation Model: The goal of the profiling and estimation model is to analyze the program statically or/and dynamically in order to determine relevant information for design exploration, hardware/software partitioning, and optimization. Multiple profiling techniques are developed to analyze the input application behavior for different criteria, such as performance, memory bandwidth, power consumption, etc. These techniques are applied either at compilation time (using static analysis) or at run-time (dynamic profiling). In [22], static pointer analysis is performed and in [10] memory size of dependencies between loops is estimated. For runtime profiling, the most common methods are code instrumentation [15], sampling [9], [21], or hardware profiling [14]. Static analysis is less accurate than dynamic profiling as it is based on estimation, while dynamic profiling is slow and requires program intervention. In our approach we combine both static and dynamic approaches to develop an efficient profiling tool in terms of accuracy and speed for hardware/software partitioning.

There are also several approaches for preliminary hardware implementation cost estimation. In [17], we find a constant time incremental estimation approach targeted at iterative hardware/software partitioning, where at each partitioning step the estimates are updated. More similar to our approach, the authors of [11] estimate area by using linear regression models per DFG node. Different DFG nodes are characterized by different linear models. Nevertheless, those works use synthesis-like schemes to predict area and delay, while we use a linear model based on software metrics.

Compiler: The main objectives of a compiler tar-

geting reconfigurable architectures is to guide the hardware/software partitioning and to generate code optimized for the specific features of such a hybrid architecture. There are several compilation approaches for reconfigurable architectures. One approach is to use standard compilers for general purpose processors (GPP) included in the target Field-programable Custom Computing Machines (FCCM) and impose the programmers to manually modify the assembly code in order to take into account the reconfigurable hardware. However this is a time consuming and error-prone process which requires deep understanding of both hardware and software features of the target architecture and application.

Another common approach (Garp [16], Napa [8]) is to use compiler front-ends with high level optimizations and to generate back C annotated code which is processed by standard C compilers for the target GPP. The annotations are transformed into calls to a dedicated library that handle the GPP-reconfigurable hardware interface. In consequence, the code quality of the generated code is decreasing due to the translation back to C thus loosing specific low level optimizations.

In our approach, we do not require manual interface between the software and the hardware. Moreover, we avoid the library calls overhead and perform low-level specific optimizations.

Automated HDL Generation: One of the major goals of the automated HDL generation is to support fast prototyping and provide early performance estimation. Projects like Handel-C [3], oriented towards the hardware designers, extend the C syntax with constructs, exposing all hardware details to the designer. Another goal of the HDL generation in the context of the reconfigurable computing systems is to allow the software designers to develop applications for such platforms without having hardware design knowledge. Projects like DEFACTO [7], SPARK [6] and ROCCC [5] consider unmodified C code as input. They emphasize parallelizing transformations and some also address memory access optimizations. Nevertheless, the majority of those projects are highly application domain oriented (ROCCC can handle only perfectly nested loops from the image processing domain) or target only one type of optimizations (SPARK focus on scheduling and increase of the instruction level parallelism). In the recent years, several commercial tools that generate implementations from HLL input also appeared (e.g. Catapult-C [1], Impulse-C[4]). These tools, however, require direct input for certain low-level optimizations as well as for the actual mapping process. In other words, these tools save code-writing time for the hardware designers rather than facilitate the software designers. Our work is similar to the previous ones in the sense that it combines the advantages of them all, but also differs in several aspects. Our tool is oriented towards the software designers like ROCCC and SPARK. But we differ from ROCCC as we target broader application domain and we are looking into different level optimizations (unlike SPARK). Moreover, we do not impose severe limitations on the accepted C subset and respect the physical limitations of the available IO bandwidth and access times. Under these limitations, the available operation parallelism is fully exploited. The generated designs have the MOLEN CCU interface, which allows actual execution on a real hardware prototype platform.

The Molen polymorphic processor is established on the basis of the tightly coupled co-processor architectural paradigm [20][19]. Within the Molen concept, a general purpose core processor controls the execution and reconfiguration of reconfigurable co-processors (RP), tuning the latter to various application specific algorithms. An operation, executed by the RP, is divided into two distinct phases: *set* and *execute*. The set phase is responsible for reconfiguring the RP for the operation. In the execute phase, the actual execution of the operations is performed. No specific instructions are associated with specific operations to configure and execute on the RP. Instead, pointers to *reconfigurable microcode* ($\rho\mu$-code) are utilized. The $\rho\mu$-code emulates both the configuration and the execution of different RP implementations resulting in two types of microcode: 1) reconfiguration microcode that controls the RP configuration; and 2) execution microcode that controls the execution of the configured RP implementation. The main advantage of this approach is that a one-time polymorphic ISA (πISA) extension of only four (up to eight) instructions supports an arbitrary number of application specific functionalities running on the reconfigurable processor RP. The complete list of all πISA instructions is presented in [18].

Generally speaking, the Molen co-processors are not limited to be only reconfigurable implementations, they can actually be various types of augmenting hardware units. For example, in the context of hArtes, a digital signal processor (DSP) and reconfigurable hardware units are considered as Molen co-processors identically.

The Molen machine organization and the Molen programing paradigm are targeted by the DelftWorkBench and hArtes toolchains.

3. Architecture Description File

The XML Architecture Description File aims to provide a flexible specification of the target architecture and it is used for information exchange between the

tools involved in the hArtes project. More specifically, we use the XML format for the description of the target reconfigurable architecture as it is both human and machine readable, self documenting, platform independent and its strict syntax and parsing constraints allow using efficient parsing algorithms. In consequence, the XML format is suitable for the structured architecture description needed by the tools involved in the hArtes project.

File Organization: In the XML Architecture description File, the root element denoted as ORGANIZATION contains two elements HARDWARE and OPERATIONS. The HARDWARE element includes a list of FUNCTIONAL_COMPONENTS and a list of STORAGE_COMPONENTS. Each FUNCTIONAL_COMPONENTS is composed by

- NAME, such as GPP, FPGA or DSP,
- TYPE, for the identification of the FUNCTIONAL_COMPONENT, such as VIRTEXIIPROXC2VP40 or IBMPOWERPC405,
- SIZE, which is relevant only for reconfigurable hardware and represents the number of available CLB,
- FREQUENCY for the maximum frequency of the FUNCTIONAL_COMPONENT.

Additionally, in order to accommodate to the Molen Programming Paradigm, each FUNCTIONAL_COMPONENT has associated a set of XRs in the range limits specified by the START_XR and END_XR elemnets. Finally, MASTER element indicates if the FUNCTIONAL_COMPONENT is the unique master processor or one of the co-processors included in the target architectures.

```
Example:
 <HARDWARE>
  <FUNCTIONAL_COMPONENT>
     <NAME> GPP </NAME>
     <TYPE> PowerPC </TYPE>
     <MASTER> YES </MASTER>
     <START_XR> 1 </START_XR>
     <END_XR> 512 </END_XR>
     <FREQUENCY> 250MHz </FREQUENCY>
  </FUNCTIONAL_COMPONENT>
```

We notice that the SIZE element is not included in the presented example, because it is optional and not relevant for a GPP.

Similarly, each STORAGE_COMPONENT contains the following elements:

- NAME, e.g. MEM1
- SHARED, used to indicate if it is shared or local memory. Ex. yes=shared memory
- FUNCTIONAL_COMPONENT, the name of the functional component associated to this storage component. EX. FPGA (local memory of the FPGA)
- TYPE, e.g. SDRAM
- SIZE, the size of the memory. Ex. 128MB
- START_ADDRESS, END_ADDRESS - the range of memory addresses mapped to this memory, in order to meet the shared memory requirement for the target architecture.

```
Example:
  <STORAGE_COMPONENT>
     <NAME> MEM1 </NAME>
     <SHARED> YES </SHARED>
     <CONNECT> ALL </CONNECT>
     <TYPE> SDRAM </TYPE>
     <SIZE> 128MB </SIZE>
     <START_ADDRESS> 0 </START_ADDRESS>
     <END_ADDRESS> FFFF..FFFF </END_ADDRESS>
  </STORAGE_COMPONENT>
```

The OPERATIONS element is used for the description of the operations that are implemented on the hardware components. It contains a list of OP elements that include the following:

- SW_PROFILE describes the software features of the application and is composed of:
  - NAME, for the name of the software operation
  - SW_EXEC_CYCLES, for the number of cycles when the operation is executed on the master processor
  - SW_CALLS, for the number of executions of the operation
- a list of IMPLEMENTATION elements which describe the hardware features of the specific implementation and has an ”id” attribute which is used by the tools that associated an unique identifier to each hardware implementation. It contains the following elements:
  - NAME, for the name of the implementation. Ex. SAD
  - COMPONENT, for the name of the component where this implementation is placed. Ex. FPGA
  - SIZE, for the size of implementation. Ex. 100 slices

– START_INPUT_XR, START_OUTPUT_XR, for the first XRs with the input /output parameters
– SET_ADDRESS, EXEC_ADDRESS, for the memory addresses of the microcode associated with SET/EXEC
– SET_CYCLES, EXEC_CYCLES, for the number of component cycles associated with hardware configuration/ execution phase
– FREQUENCY - the frequency of the implementation

```
Example:
<OP ID="SAD">
  <SW_PROFILE>
    <NAME> SAD </NAME>
    <SW_EXEC_CYCLES> 10000 </SW_EXEC_CYCLES>
    <SW_CALLS> 100 </SW_CALLS>
  </SW_PROFILE>

  <IMPLEMENTATION ID="11">
    <NAME> SAD_V1 </NAME>
    <COMPONENT> FPGA </COMPONENT>
    <SIZE> 100 </SIZE>
    <START_INPUT_XR> 3 </START_INPUT_XR>
    <START_OUTPUT_XR> 10 </START_OUTPUT_XR>
    <SET_ADDRESS> 0X00000000 </SET_ADDRESS>
    <EXEC_ADDRESS> 0X00000000 </EXEC_ADDRESS>
    <SET_CYCLES> 100 </SET_CYCLES>
    <EXEC_CYCLES> 200 </EXEC_CYCLES>
    <FREQUENCY> 150MHz </FREQUENCY>
  </IMPLEMENTATION>
</OP>
```

Tools Interaction with the XML Description File: In the presented Architecture Description File, there is a clear delimitation about the hardware/software features of the target architecture/application. The information included in a SW_PROFILE element has to be provided by the hArtes profiling tools, while the information contained in an OP element has to be provided by the automatic synthesis tools that generates a specific implementation for a specific operation. Finally, the information for the HARDWARE element are general and should be introduced by the hardware provider.

It is important to mention that this is the initial Architecture Description File and additional relevant information can be easily added when required by different components of the hArtes toolchain.

4. Profiling and Estimation Model

In this section, we discuss profiling and estimation framework. The goal of this framework is to analyze the program statically or/and dynamically in order to determine relevant information for design exploration, hardware/software partitioning, and optimization.

Profiling: The first step in our framework is program analysis and profiling. The goal of this step is to determine relevant information for design exploration, partitioning, and optimization. The conceptual view of the profiling stage is depicted in Figure 2. As it can be observed, a distinction between dynamic and static profiling is made. Static profiling comprises all compile-time program analysis, while in the dynamic profiling the application run-time behavior is considered.

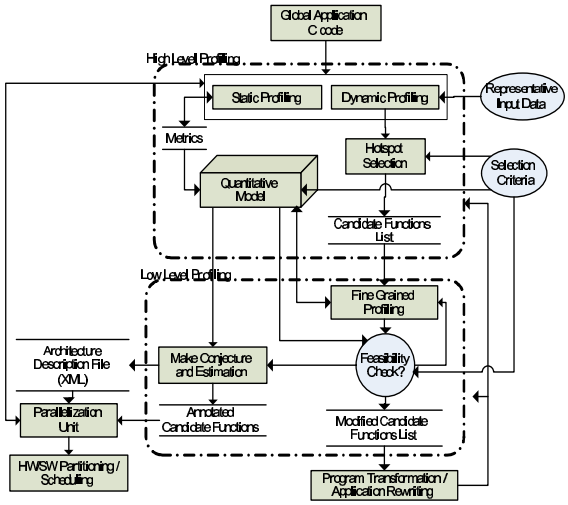

Figure 2. Profiling Framework

The main goal of the dynamic code analysis is to determine which parts of an application are computationally intensive when executed on a GPP. The coarse grain profiling inspects the application at function level and gathers information such as CPU time or the execution frequency of these functions. The result of such analysis can be a large list of kernel functions for complex applications. Based on the selection criteria provided (such as performance, area, power consumption, memory etc.), the profiler identifies the most promising functions and constructs the candidate for acceleration function lists. These candidate functions can be analyzed further at fine-grain level (such as basic block or statement level) to gather detailed information such as number of memory accesses, loop nesting level, etc.

The profiler also helps to estimate the potential speedup that can be achieved when the kernels are executed on the reconfigurable hardware and estimates the initial cost of a hardware mapping. Additionally, it could be used to predict the upper and lower bounds of the speed up possible for the whole application. The output of the profiler is included in the architecture description file (see Section 3) and in the C code, where candidate functions are annotated with the profiling in-

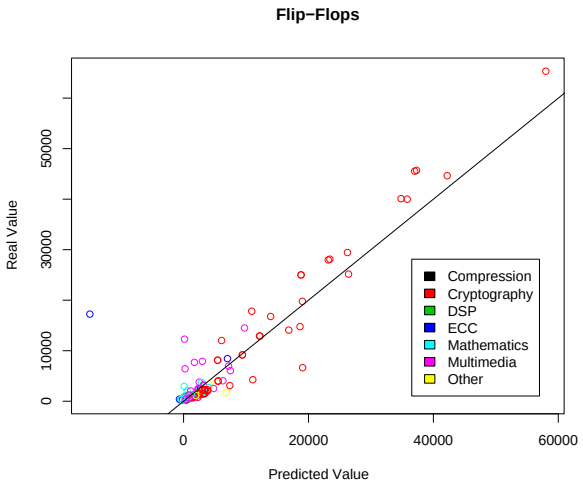

Figure 3. Actual versus predicted number of Flip-Flops

formation.

Estimation Model: In the current stage of the project, static profiling information is fed into a quantitative prediction model for hardware/software partitioning. This linear model predicts area and delay using software complexity metrics, which are values that represent application structure. An example is the Cyclomatic Complexity [13], which stands for the number of decisions in the code and thus gives some indication of the control intensity of the code.

The first version of this model is now complete and it is based on a set of 126 C-functions from a broad range of application domains, like Compression, Multimedia, and Cryptography. The functions were compiled to VHDL by the DWARV C-to-VHDL compiler (see Section 6) and synthesized. The resulting area was used to fit the model. The model predicts several area metrics, like the number of LUTs and Flip-Flops. As an example the Flip-Flop model has an estimated error of $28,1\%$ and its general performance can be seen in Figure 3.

In our future work we aim to extend this model further by adding prediction of delay and reconfiguration latency. Furthermore, specific cases might benefit from specialized models. Also, we can incorporate advanced software complexity metrics to improve our model quality.

We notice that not all types of software code are suitable for hardware implementation. Certain code segments (such as I/O access) impose constraints for the hardware mapping. The hardware mapping of these code segments, instead of increasing expected performance, they decrease the performance due to the overhead associated with mapping. This characteristic of the software code makes it unsuitable for hardware mapping. The feasibility check uses the linear model described above and excludes these notorious functions which are not profitable for hardware mapping. In the subsequent phase of the tool chain, these candidate function modules can be aggregated/segregated for optimum fit in order to satisfy the constraints (such as area, power, memory ports) imposed on chip.

5. MOLEN Compiler

In this section, we present the design flow and the organization of the MOLEN compiler which aims to allow easy compilation and optimizations of applications for the reconfigurable architecture.

The compiler uses the XML Architecture Description File discussed in section 3 in order to extract profile information and software/hardware features of the kernels proposed for execution on the reconfigurable hardware. Based on these features the compiler can apply the appropriate optimizations, transformations and scheduling algorithms which are specifically designed for such hybrid architectures.

GNU C compiler extensions: The compiler is based on GNU C compiler infrastructure version 4.1 for the PowerPC GPP and the following extensions have been implemented in order to accommodate the Molen Programming Paradigm [18].

In the first step, the functions annotated with pragma directives for execution on the reconfigurable hardware are identified in the compiler frontend. Next the calls to the functions identified in the previous step are replaced with associated SET/EXECUTE pseudo-functions. Finally based on the Architecture Description File, the compiler generates the corresponding SET/EXECUTE instructions, instructions to transfer parameters, instructions to obtain the return result and instructions for synchronization. Moreover a set of 512 transfer register (denoted as XRs) associated to the reconfigurable hardware have been added as a separate Register File.

For a C application that contains the code presented in Figure 4, the generated code is depicted in Figure 5. The XR registers are implemented as memory address (0x000a, 0x000b and 0x0014 in the example). The instruction sequence composed of ”nop”, ”sync” and ”bl” is for synchronization. The encoding for SET/EXECUTE instructions includes the memory address of the reconfigurable microcode that is used for the hardware configuration and execution of a specific operation.

```
#pragma call_fpga op
int func(int x,int y) {
  ...
}

void main(int argc,char **argv) {
  ...
  c = func(1,2);
  ...
}
```

Figure 4. Application source code

```
  .long 8388899   # encoding for SET
   li 9,1
   mtdcr 0x000a,9
   li 9,2
   mtdcr 0x000b,9
   creqv 6,6,6
   sync
   nop
   nop
   nop
   bl .L5
.L5:
   .long 8389718   # encoding for EXECUTE
   nop
   mfdcr 3,0x0014
```

Figure 5. Generated assembly code

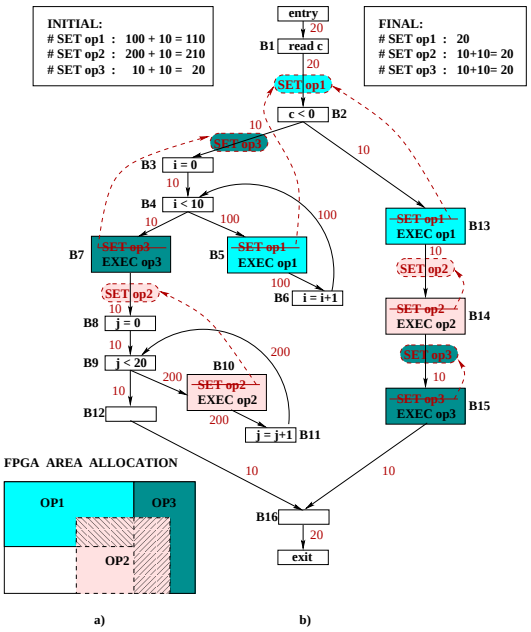

Figure 6. a) Area allocation b) Application control flow graph

MOLEN specific optimizations: In order to reduce the reconfiguration overhead, which is a main shortcoming of the current FPGAs, an algorithm that combines scheduling (for hiding reconfiguration latency) and hw/sw codesign has been proposed. The main idea of the algorithm is to make an intra-procedural scheduling of SET instructions in order to anticipate the reconfiguration well in advance of hardware execution (and thus 'hiding' the reconfiguration latency). The algorithm is aware of the placement constraint and using the Architecture Description File (Section 3) can decide that the implementation of a certain function in hardware does not provide performance improvement and switch back to its software implementation.

An example is shown in Figure 5. The graph in the figure represents the control flow graph of a program that uses reconfigurable operations op1, op2 and op3. The labels on the edges represent the execution frequency of the edges. An FPGA area allocation is also presented and it can be seen that op2 conflicts with both op1 and op3.

We can easily observe that repetitive configurations of op1 and op2 in the loops (blocks B5 and B10) are redundant. Also after moving the configuration of op1 (to block B2) the configuration from block B13 becomes useless, so it will be eliminated.

The proposed algorithm is based on two data flow anlyses, graph algorithms, profile information and includes the following:

- construction of anticipability graph. This is a subset of the control flow graph in which the edges that cannot 'propagate' upwards the set instructions are eliminated.

- minimum s-t cut. This phase has the role to select those edges that minimize the number of required reconfigurations. The total number can be computed in various ways, using static or dynamic profiling.

- selection of hardware/software execution. If the reconfiguration overhead could not be sufficiently reduced even after the above presented scheduling, the algorithm decides to switch to pure software execution of an operation. This selection requires information about hardware conflicts and execution time must be available.

The proposed optimization is going to be implemented in the GCC compiler which provides support for data-flow analysis and scheduling. Additionally the proposed algorithm could be extended for interprocedural SET scheduling which would further improve the performance gains. Finally we notice that the FPGA

area allocation has a significant impact on the considered algorithm and the compiler can be used to provide directives for the FPGA area allocator.

In our future work, we will address the ”memory wall” problem, which is very important in the context of the reconfigurable computing. Additional microcode prefetch instructions should be carefully6 scheduled in order to reduce the data transfer overhead. Finally, we will address the parallel execution of multiple operations on the reconfigurable hardware.

6. HDL Generation

After the hardware/software partitioning is performed, HDL designs for the hardware segments have to be developed. The manually crafted designs are the best solution from a performance/quality point of view. Nevertheless, such a design takes several men months to be developed. Another possible solution that could speed-up the HW design process is the use of already existing IP cores. However, the IP library solution also has some shortcomings. First, the necessary core might not be available in the library. Another problem with the IP cores is that they are not designed for a particular application, rather to be used in multiple scenarios. Hence, they include overhead in functionality, which translates to area overhead. Moreover, the IP cores have various interfaces. Thus, wrappers have to be implemented for each IP core. The third possibility of automated HDL generation cannot provide the quality of the manually crafted designs. Nevertheless, by applying set of optimizations, designs with a reasonable quality can be derived within seconds. Moreover, such designs are tailored for the particular application and hardware organization.

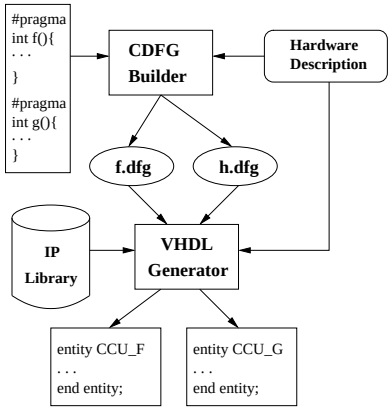

Figure 7. DWARV Toolset

Due to these reasons, in the DelftWorkBench project we envision all three approaches for hardware generation. The most critical kernels are implemented manually. The kernels, for which IP cores are available, are instantiated from the library. For non-critical kernels, as well as for the purposes of fast prototyping and fast performance estimation, automated HDL generation is considered. This generation is performed by DWARV[1] toolset (see Figure 7). The input of DWARV is annotated C code. The initial emphasis of DWARV was to provide straightforward generation of VHDL designs considering actual software/hardware co-execution on the MOLEN polymorphic processor. Therefore, considering also the tool's early stage of development, several restrictions are imposed on the input code. Currently, only one dimensional memory addressing is supported. Structures, unions, and floating point types are not supported yet. The iteration and selection statements are limited to for and if statements, respectively. Additionally, control jumps and function calls are not allowed. Although syntactically restricted, this C subset does not impose severe limitations on the supported functionality. The unsupported constructs can be substituted with others preserving the statement semantic (e.g., $while$ loop can be substituted with for loop). An additional requirement of the toolset is the code segment that has to be implemented in the hardware to be separated in a pragma annotated function. The input code first is processed by a CDFG Builder. This tool is currently implemented as a pass within the SUIF2 compiler framework. The purpose of this module is to perform high-level hardware-independent optimizations on the code and to transform it into intermediate representation (IR), suitable for hardware mapping. Currently, the set of implemented optimizations includes simplified scalar replacement, static single assignment, common sub-expression elimination, and dead code elimination. The output IR is a hierarchical data-flow graph (HDFG) in a binary format. This is a directed acyclic graph with two types of nodes: simple and compound. The simple nodes represent arithmetic and logic operations and registers. The compound nodes represent the loops in the input code and contain sub-HDFG of the loop body. The edges of the graph represent the data dependencies and the precedence order (not shown in the figure) between the operations. The HDFG is further process by a VHDL Generator. This tool is currently implemented as a stand-alone console application. Its purpose is to perform low-level hardware-dependent optimizations and to generate the final VHDL code. Currently, the tool performs only ASAP scheduling on the input graph. The memory bandwidth and access times are provided to the VHDL Generator as addi-

[1] DelftWorkBench Automated Reconfigurable VHDL Generator

tional input. The currently selected computation model is FSM-based.

An example of the performed translation process is presented in Figure 8. The input C code (Figure 8a) is the $fmult$ function of the G721 encoder application from the MediaBench benchmark suite [12]. The function is transformed into the HDFG, shown in Figure 8b by the CDFG Builder. The shaded area in the figure is the compound $loop$ node of the for-loop in the function with the loop body sub-DFG. The edges denote the data dependencies between the operations. The precedence edges are not shown in the figure. The generated graph is further processed by the VHDL generator and an FSM-based design is generated (Figure 8c).

The long term goal of DWARV is to provide optimized support for broad range of application domains. In order to achieve this goal, work will be carried out in several directions. In the first place, the input language restrictions will be relaxed. The targeted set includes almost all C language constructs, excluding dynamic memory routine, interrupt and exception handling. The CDFG Builder will be extended to perform advanced optimizations on the input code based on the information provided by the profiler or other preceding tools. A summary of the required input is given in Table 1. In addition, analysis of the semantic of the code is also considered. This semantic analysis would allow to select the most appropriate optimization set and computation model for the target application. The nodes of the HDFG will be extended with new types of compound nodes such as VLIW instruction-like nodes. The VHDL Generator will be extended to perform resource allocation and more sophisticated scheduling of the transformed algorithm. As an additional input, a component library will be provided. Such a library will contain available IP cores and DSP blocks and would allow for optimized exploitation of the underlying hardware. Optimal implementation of non-trivial operations (division, floating point arithmetic) will also be included in this library. The set of generated outputs will be extended. The algorithm will not be mapped to a single FSM-based model. Rather, the particular model will be selected based on the semantic analysis, performed by the CDFG Builder.

Software	**Hardware**
Pragma annotation of the segments to be implemented in the hardware[a]	Memory bandwidth and access time[a]
Data (un)aliasing	XREG bandwidth and access timetime[a]
Actual data size	Number of memory banks
Data alignment	Components (IP) library
Data distribution	Available area
IO parameters XREG numbers	DSP blocks

[a]Currently also required

Table 1. HDL Generator Input Requirements

7. Conclusions

In this paper, we presented the profiling, compilation and HDL generation tools from the DelftWorkBench project. The goal of those tools is to facilitated and automate the design and development of RC applications. The profiling framework supports the HW/SW partitioning suggesting code segments as candidates for hardware acceleration. In addition, the profiler provides information, required for advanced optimizations performed by the other tools of the hArtes toolchain. Transparently for the designer, the compiler handles the GPP-reconfigurable hardware interface and provides optimizations that reduce the reconfiguration and data transfer penalties. The HDL generator allows fast prototyping and early performance estimation. It also assits the designers to develop applications for reconfigurable architectures without in-depth hardware knowledge.

8. Acknowledgment

The DelftWorkBench related research is sponsored by:

- the hArtes project (IST-035143) supported by the Sixth Framework Programme of the European Community under the thematic area ”Embedded Systems”.
- the MORPHEUS project (IST-027342) supported by the Sixth Framework Programme of the European Community under the thematic area ”Embedded Systems”.
- the RCOSY project (DES-6392) supported by the STW (Dutch Science Foundation) and ACE (Associated Compiler Experts)

References

[1] C-based design. Online: http://www.mentor.com/products/c-based_design/.
[2] Delft workbench. Online: http://ce.et.tudelft.nl/DWB/.
[3] Handle-c language reference. Online: http://www.celoxica.com/.

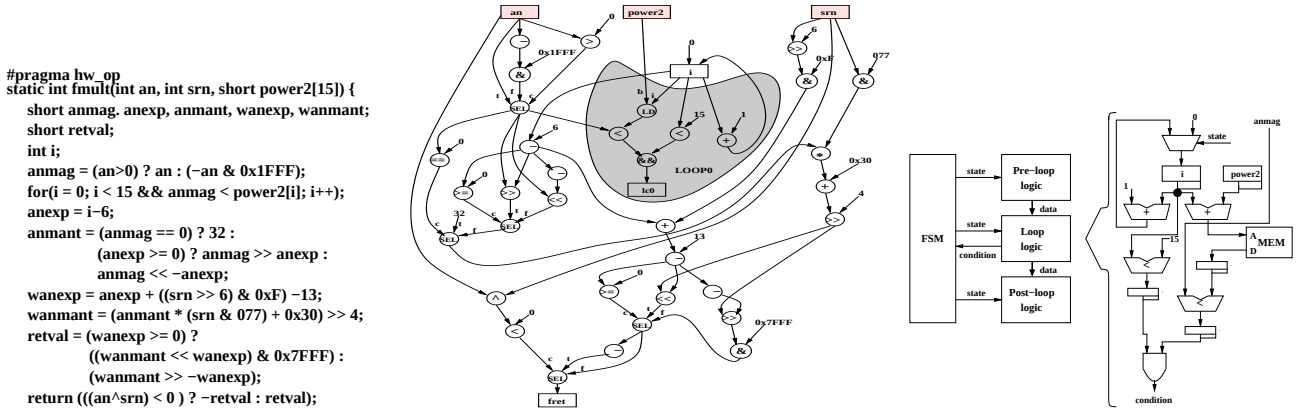

Figure 8. G721 encoder - $fmult$: (a) C code; (b) HDFG; (c) schematics

[4] Impulse-c. Online: http://www.impulsec.com/.

[5] Riversite optimizing complier for configurable computing. Online: http://www.cs.ucr.edu/ roccc/.

[6] Spark: A parallelizing approach to the high-level synthesis of digital circuits. Online: http://mesl.ucsd.edu/spark/.

[7] P. C. Diniz, M. W. Hall, J. Park, B. So, and H. E. Ziegler. Bridging the gap between compilation and synthesis in the defacto system. In *Proceedings of 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*, pages 52–70, 2001.

[8] J. Gokhale, M.B. Stone. Napa c: Compiling for a hybrid risc/fpga architecture. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 126, 1998.

[9] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[10] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas. Data dependency size estimation for use in memory optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(7):908–921, 2003.

[11] D. Kulkarni, W. A. Najjar, R. Rinker, and F. J. Kurdahi. Fast area estimation to support compiler optimizations in fpga-based reconfigurable systems. In *FCCM '02: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 239, Washington, DC, USA, 2002. IEEE Computer Society.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[13] T. J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[14] R. V. Peri, S. Jinturker, and L. Fajardo. A novel technique for profiling programs in embedded systems. In *Second ACM Workshop on Feedback-Directed and Dynamic Optimization*.

[15] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528–539, 2004.

[16] J. W. Timothy J. Callahan, John R. Hauser. The garp architecture and c compiler. *Computer*, 33, 2000.

[17] F. Vahid and D. D. Gajski. Incremental hardware estimation during hardware/software functional partitioning. In *Readings in hardware/software co-design*, pages 516–521. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[18] S. Vassiliadis, G. N. Gaydadjiev, K. Bertels, and E. M. Panainte. The molen programming paradigm. In *Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 1–10, July 2003.

[19] S. Vassiliadis, S. Wong, and S. D. Cotofana. The molen $\rho\mu$-coded processor. In *in 11th International Conference on Field-Programmable Logic and Applications (FPL), Springer-Verlag Lecture Notes in Computer Science (LNCS) Vol. 2147*, pages 275–285, August 2001.

[20] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The molen polymorphic processor. *IEEE Transactions on Computers*, pages 1363– 1375, November 2004.

[21] D. W. Wall. Systems for late code modification. In R. Giegerich and S. L. Graham, editors, *Code Generation - Concepts, Tools, Techniques*, pages 275–293. Springer-Verlag, 1992.

[22] J. Zhu and S. Calman. Context sensitive symbolic pointer analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circ uits and Systems*, 24(4):516–531, April 2005.