The Molen Compiler for Reconfigurable Processors

ELENA MOSCU PANAINTE, KOEN BERTELS and STAMATIS VASSILIADIS
TU Delft

In this paper, we describe the compiler developed to target the Molen reconfigurable processor and programming paradigm. The compiler automatically generates optimized binary code for C applications, based on pragma annotation of the code executed on the reconfigurable hardware. For the IBM PowerPC 405 processor included in the Virtex II Pro platform FPGA, we implemented code generation, register and stack frame allocation following the PowerPC EABI (Embedded Application Binary Interface). The PowerPC backend has been extended to generate the appropriate instructions for the reconfigurable hardware and data transfer, taking into account the information of the specific hardware implementations and system. Starting with an annotated C application, a complete design flow has been integrated to generate the executable bitstream for the reconfigurable processor. The flexible design of the proposed infrastructure allows to consider the special features of the reconfigurable architectures. In order to hide the reconfiguration latencies, we implemented an instruction scheduling algorithm for the dynamic hardware configuration instructions. The algorithm schedules in advance the hardware configuration instructions, taking into account the conflicts for the reconfigurable hardware resources (FPGA area) between the hardware operations. To verify the Molen compiler, we used the multimedia video frame M-JPEG encoder of which the extended Discrete Cosine Transform(DCT*) function was mapped on the FPGA. We obtained an overall speedup of 2.5 (about 84 % efficiency over the maximal theoretical speedup of 2.96). The performance efficiency is achieved using automatically generated non-optimized DCT* hardware implementation. The instruction scheduling algorithm has been tested for DCT, Quantization and VLC operations. Based on simulation results, we determine that, while a simple scheduling produces a significant performance decrease, our proposed scheduling contributes for up to 16x M-JPEG encoder speedup.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation*; *Compilers*; *Optimization*; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable architectures*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Gate arrays*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Instruction scheduling, Reconfigurable computing, FPGA

1. INTRODUCTION

Reconfigurable computing (RC) is becoming increasingly popular as it bears the promise of combining the flexibility of software with the performance of hardware. Some concern can be expressed because the current state of the art of FPGA developement tools assumes that the developer has a deep understanding of both software and hardware development before the benefits of this technology can be exploited. This justified concern underlines the necessity to intensify research and development efforts to support the designer in this process. The Delft Workbench is an initiative that investigates the integration and development of tools supporting the different design phases starting at code profiling, synthesis and ending at the generation of binary code. The idea is to automate as much as possible the design exploration and the final development process. This paper addresses an important part of the tool chain, namely the construction of the Molen compiler that targets such a hybrid platform. The compiler allows on the basis of function annotations, the automatic modification of applications to generate the appropriate binaries.

The latest commercial FPGA platforms now offer support for partial and dynamic hardware configurations. Nevertheless, one of their main drawback remains large reconfiguration latencies. In order to hide these latencies, compiler support is fundamental to automatically schedule and optimize the compiled application code for efficient reconfigurable hardware usage. When dealing with reconfigurable hardware, the compiler should also be aware of the competition for the reconfigurable hardware resources (FPGA area) between multiple hardware operations during the application execution time. A new type of conflict - called in this paper FPGA-area placement conflict - emerges between the hardware configurations that cannot coexist together on the target FPGA.

The current paper reports on the completed Molen compiler targeting the Molen implementation on the Virtex II Pro platform FPGA. The contributions of the paper can be summarized as follows :

—A compiler backend targeting the PowerPC processor included in the Molen prototype has been developed. The theoretical compiler extensions presented in [Moscu Panainte et al. 2003] have been implemented and adjusted to the target Field-programmable Custom Computing Machine (FCCM) features. Software/hardware development tools have been integrated to automate the design flow phases.

—A general instruction scheduling algorithm that minimizes the number of required hardware configurations taking into account both the FPGA-area placement conflicts and the code profile information for the compiled software application is proposed. More specifically, the algorithm anticipates the hardware configurations in less frequently executed application points avoiding the FPGA-area placement conflicts.

—The presented compiler has been used for the compilation of the Motion JPEG encoder application for the Molen reconfigurable processor, with the DCT* function executed on the reconfigurable hardware. We measured a speedup of 2.5 against a maximal speedup of 2.96, which represents 84 % performance efficiency using automatically generated but non-optimized DCT* hardware implementation. The instruction scheduling algorithm contributes for 43 % up to 94 % performance improvement compared to the pure software execution, while a simple scheduling algorithm produces a significant performance decrease.

The paper is organized as follows. In the following section, we present background information and related work. In Section 3, we describe the Molen compiler required for the PowerPC processor and the Molen prototype. A formal description of our scheduling problem and the instruction scheduling algorithm are included in Section 4. The experimental results are discussed in Section 5 and finally, we present conclusions and future work.

2. BACKGROUND AND RELATED WORK

In this section, we briefly discuss the Molen programming paradigm [Vassiliadis et al. 2003], describe the Molen machine organization that supports it and discuss related work.

The Molen programming paradigm [Vassiliadis et al. 2003] is a sequential consistency paradigm for programming FCCMs possibly including a general-purpose computational engine(s). The paradigm allows for parallel and concurrent hardware execution and is intended (currently) for single program execution. For a given ISA, a one time architectural extension (based on the co-processor architectural paradigm) comprising 4 instructions

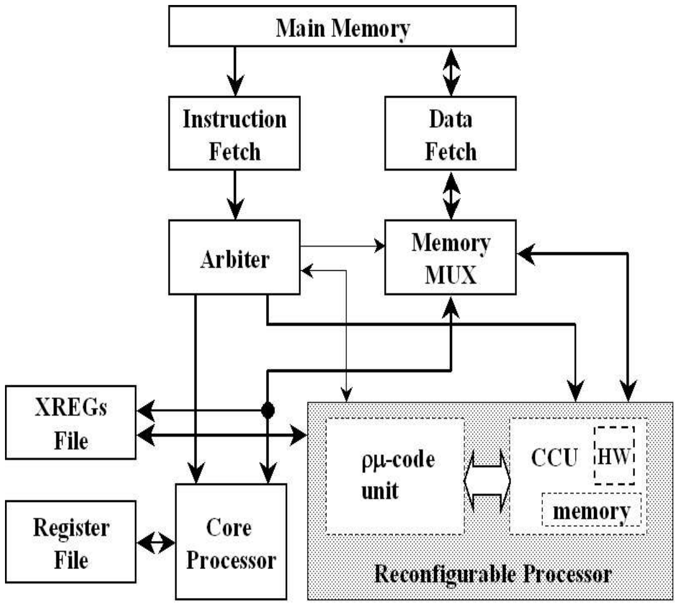

Fig. 1. The Molen machine organization

(for the minimal πISA as defined in [Vassiliadis et al. 2003]) suffices to provide an almost arbitrary number of operations that can be performed on the reconfigurable hardware. The four basic instructions needed are **set**, **execute**, **movtx** and **movfx**. By implementing the first two instructions (**set/execute**) a hardware implementation can be loaded and executed on the reconfigurable processor. The **movtx** and **movfx** instructions are needed to provide the communication between the reconfigurable hardware and the general-purpose processor (GPP).

The Molen machine organization [Vassiliadis et al. 2001] that supports the Molen programming paradigm is described in Figure 1. The two main components in the Molen machine organization are the 'Core Processor', which is a GPP and the 'Reconfigurable Processor' (RP). Instructions are issued to either processors by the 'Arbiter' by means of a partial decoding of the instructions received from the instruction fetch unit. The support for the SET/EXEC instructions required in the Molen programming paradigm is based on *reconfigurable microcode*. The reconfigurable microcode is used to emulate both the configuration of the Custom Computing Unit (CCU) and the execution of implementations configured on the CCU.

In the last decade, several approaches have been proposed for coupling an FPGA to a GPP. In the rest of this section, we present the software support for programming such hybrid architectures and a discussion of how they relate to the research reported in this paper.

Reconfigurable Architectures Performance: Several reconfigurable architectures have been proposed in the last decade (see [Sima et al. 2002] for a classification). The reported performance improvements are mainly based on simulation or estimation (e.g. [Kastrup et al. 1999] [Rosa et al. 2003]) results. Eventhough some implementations exist [Lee et al. 2000], in most cases the performance is just estimated. In this paper, we present the validation of the Molen approach based on a real and running implementation of the Molen reconfigurable processor platform.

Compilers for Reconfigurable architectures: When targeting hybrid architectures to improve performance, the applications must be partitioned in such a way that certain computation intensive kernels are mapped on the reconfigurable hardware. Such mapping is not simple as it assumes deep understanding of both software and hardware design. Several approaches (e.g. [Campi et al. 2003] [Rosa et al. 2003]) use standard compilers to compile the applications to FCCMs. As standard compilers do not target reconfigurable architectures, the kernel computations implemented in hardware are manually replaced by the appropriate instructions for communication with and controlling the reconfigurable hardware. This replacement is done manually making it a time-consuming [Stefanov et al. 2004] and error-prone process. Automated tools (compilers) should be used in order to facilitate the design and development process [Gokhale and Stone 1998] [Kastrup et al. 1999] [Ye et al. 2000]. In the DEFACTO project [Bondalapati et al. 1999], compiler analyses and transformations are involved in HW/SW partitioning, while standard C compilers are used to generate the machine code for the target GPP. However, the extensions of the cited compilers mainly aim to generate the instructions for the reconfigurable hardware and they are not designed to easily support new optimizations that exploit the possibilities of the reconfigurable hardware. The Molen compiler presented in this paper, is based on a a flexible and extensible infrastructure that allows to add easily new optimization and analysis passes that take into account the new features of the reconfigurable target architecture.

Compiler Support for Hiding the Reconfiguration Latency: One of the major drawbacks of the reconfigurable hardware is the huge reconfiguration latency [Bolotski et al. 1994] [Sima et al. 2002]. Different techniques such as configuration caching and prefetching (e.g. [Vassiliadis et al. 2001]) have been proposed to reduce the reconfiguration latency. These hardware techniques should be combined with compiler optimizations that provide an efficient instruction scheduling to use the available parallelism between different FCCMs components in the hardware reconfiguration phase. Nevertheless, many FCCMs do not expose a specific instruction for hardware reconfiguration (see [Sima et al. 2002] for FCCMs classification), thus impeding compiler support for hiding reconfiguration latency.

In [Moscu Panainte et al. 2004a], it has been reported that redundant hardware configuration produces significant performance decrease due to the huge reconfiguration latency of current FPGA. In order to prevent this performance loss, inappropriate scheduling of the hardware configuration instructions (e.g., inside loops) should be avoided. The recent scheduling algorithm discussed in [Moscu Panainte et al. 2004b] addresses this problem for the particular case when only one hardware operation has to be scheduled.

However, in order to achieve significant performance improvement for real applications, more than one operation is usually implemented in hardware. As the available area of the reconfigurable platforms is limited, the coexistence of all hardware configurations on the FPGA for all application execution time may be restricted. Moreover, hardware implementations of these operations can be developed by different IP providers that can impose a predefined FPGA area allocated for each operation, resulting in FPGA-area placement conflicts. Two hardware operations have such a FPGA-area placement conflict (or just conflict in the rest of the paper) if i) their combined reconfigurable hardware area is larger than the total FPGA area or ii) the intersection of their hardware areas is not empty.

A compiler approach that considers the restricted case of two consecutive and non-conflicting hardware operations is presented in [Tang et al. 2000]. In this approach, the hardware execution of the first operation is scheduled in parallel with the hardware configuration of the second operation. Our approach is more general as it performs scheduling for any number of hardware operations at procedural level and not only for two consecutive hardware operations. The performance gain produced by our scheduling algorithm results from reducing the number of performed hardware configurations.

3. THE MOLEN COMPILER

The Molen compiler comprises the Stanford SUIF2[SUIF2] (Stanford University Intermediate Format) Compiler Infrastructure for the frontend and the Harvard Machine SUIF framework[MachineSUIF] for developing compiler backends and optimizations. In [Moscu Panainte et al. 2003], the theoretical compiler extensions target a virtual Molen reconfigurable architecture including an *x86* processor as a GPP. In this section we present the implemented compiler backend and extensions required for the Molen hardware implementation on the Virtex II Pro platform FPGA which includes a PowerPC processor.

PowerPC Backend: The first step is to have a backend C-compiler that generates the appropriate binaries to be executed on the PowerPC processor integrated on the Virtex II Pro board. As the current MachineSUIF backends excluded the backend for PowerPC architecture, we developed a PowerPC compiler backend and implemented the PowerPC instruction generation, register allocation, stack frame allocation and software floating-point emulation following the PowerPC EABI. Additionally, in order to exploit the opportunities offered by the reconfigurable hardware, the PowerPC backend has to be extended in several directions, as described in the rest of this section.

Compiler Extensions for Molen Implementation: The Molen organization previously presented has been adopted in this paper. We have implemented the minimal instruction set extension, containing the following:

—SET/EXECUTE instructions are included in the MIR (Medium-level Intermediate Representation) and LIR (Low-level Intermediate Representation) of the Molen compiler. In order not to modify the PowerPC assembler and linker, the compiler generates the instructions in binary form. For example, for the instruction *exec 0x80000C* the generated code is *.long 1A000031* where the encoding format (presented in [Kuzmanov and Vassiliadis 2003]) is recognized by the arbiter.

—MOVTX/MOVFX: The equivalent PowerPC instructions are *mtdct/mfdcr*. Moreover, the XRs (exchange registers) are not physical registers but they are mapped at fixed memory addresses.

In Figure 2, we present the code generated by the Molen compiler for the DCT* function call executed on the reconfigurable hardware. In order to correctly generate the instructions for hardware configuration and execution, the compiler needs information about the DCT* hardware implementation. This information is described in an FPGA Description File, which contains for the DCT* operation the fields presented in Figure 3. Line 2 defines the start memory address from where the XRs are mapped. In line 3, the compiler is informed that there is a hardware implementation for the DCT* operation with the microcode addresses for SET/EXECUTE instructions defined in lines 4-5. The *sync* instruction from Figure 2 is a PowerPC instruction that ensures that all instructions preceding *sync* in program order complete before *sync* completes. The sequences of *sync* and *nop* instruction

```
la 3, 12016(1)            #load the address of the first param
la 12, 12016(1)           #load the address of the second param
mtdcr 0x00000056,3        #send the address of the first parameter
mtdcr 0x00000057,12       #send the address of the second parameter
sync                      #
nop                       #synchronization
nop                       #
nop                       #
bl main._label0           #instr. required by the arbiter impl.
main._label0:
.long 0x1A000031          #exec 0x8000C
nop                       #synchronization
```

Fig. 2. Code generated by the Molen compiler for the reconfigurable DCT* execution

```
1: NO_XRS          = 512        # number of available XRs
2: START_XR        = 0x56       # the memory address of the first XR
3: OP_NAME         = dct        # info about the DCT* operation
4: SET_ADDR        = 0x39A100   # the address of the DCT* SET microcode
5: EXEC_ADDR       = 0x80000C   # the address of the DCT* EXEC microcode
6: END_OP                       # end of the info about the DCT* operation
................................                # info about other operations
```

Fig. 3. Example of an FPGA Description File

are used to flush the processor pipeline. The SET instruction is not included in the above example because it has been scheduled earlier by the Molen compiler scheduling algorithm presented in the following section.

4. INSTRUCTION SCHEDULING FOR DYNAMIC HARDWARE CONFIGURATIONS

In this section, we propose a general approach for intraprocedural instruction scheduling of the hardware configuration instructions taking into account the FPGA-area placement conflicts. It is based on the state-of-art compiler optimization for partial expression redundancy elimination presented in [Cai and Xue 2003]. In order to incorporate the FPGA-area placement conflicts between the hardware operations, we introduce a new type of data-flow analysis as described in this section. Additionally, it can switch for one operation from hardware execution to its software execution when the hardware operation provides no performance improvement even after the scheduling phase.

In order to illustrate the instruction scheduling algorithm, we present a motivational example that emphasizes the main features of the proposed algorithm.

Example: The motivational example is presented in Figure 4(b) and it shows the control-flow graph of a procedure, when op1, op2 and op3 operations are performed on the reconfigurable hardware. The numbers associated with each edge of the graph represent the execution frequency of the edge. The FPGA area allocation for the considered hardware operations is presented in Figure 4(a). As mentioned in Section 2, we observe that op1 and op2 cannot fit together on the FPGA (thus op1 conflicts with op2) while op2 and op3 have a common overlapping area (thus op2 conflicts op3).

One first observation is the redundant repetitive execution of SET op1 instruction from B5 in the loop B4-B5-B6. Additionally, it should be noticed that moving this SET op1

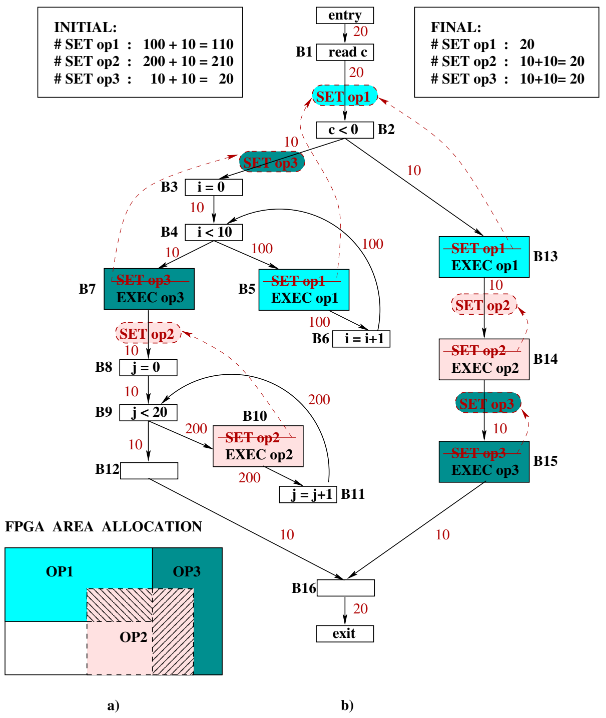

Fig. 4. Motivational example for instruction scheduling of hardware configurations (b) with FPGA-area placement constraints (a)

instruction on (B1, B2) edge will also make redundant the SET op1 instruction from B13. In the initial simple scheduling, the FPGA is configured for op1 100 times in B5 and 10 times in B13. As a result of our scheduling algorithm, the hardware configuration for op1 will be executed only 20 times. The hardware configuration for op2 from B10 cannot be moved further then B7, as it will change the hardware configuration for op3 that must be performed in B7.

The scheduling algorithm can also select a software execution for an operation, when its reconfigurable hardware execution does not produce the expected performance improvement. In our example from Figure 4(b), there are no redundant configurations for op3, thus the hardware execution of op3 has to be preceded each time by the hardware configuration. When the hardware configuration consumes all the performance gain produced by the hardware execution of op3, the scheduler can switch to software execution on the GPP.

4.1 Instruction Scheduling Problem Statement

We represent the control flow graph (CFG) of a procedure as a directed graph G $<N,E,w>$ where the nodes N represent the basic blocks, the edges E represent the control flow dependencies and the weight function w: E $\rightarrow R^{+}$ represents the execution frequency of each edge. The operations implemented in hardware are included in HW set. We define DEF_{op} as the set of basic blocks n $\in$ N that contain an instruction *SET op* immediately followed by *EXEC op* instruction. A node n $\in$ DEF_{op} is called a definition node for op. In our example from Figure 4, B5 and B13 are definition nodes for op1. An

FPGA-area placement conflict between two operations op1 and op2 is represented as op1 $\nleftrightarrow$ op2. The information about these conflicts is provided by a symmetric function f : HW x HW $\rightarrow$ {0,1}, where f(op1, op2) = 1 if op1 $\nleftrightarrow$ op2, and 0 otherwise. We define $Conflict_{op} = \{n \in N | \exists op_i \in HW, n \in DEF_{op_i} \wedge op \nleftrightarrow op_i\}$. A node n $\in$ $Conflict_{op}$ is called a conflict node for op. In Figure 4, B10 and B14 are conflict nodes for both op1 and op3.

In order to simplify this discussion, we make the following assumptions. We assume that there is a single **entry** node with no predecessor (pred(entry) = $\emptyset$, where pred(n)={m $\in$ N | (m,n) $\in$ E}) and a single **exit** node with no successor (succ(n) = $\emptyset$, where succ(n)={m $\in$ N | (n,m) $\in$ E}). Also, we assume that a node cannot be simultaneous in DEF_{op} and $Conflict_{op}$ (e.g. a node cannot be a definition node for two conflicting operations). In consequence, when more conflicting operations are included in the same basic node, this node must be split into a set of nodes, one for each operation. The final assumption is that only the SET/EXECUTE instructions included in the CFG affect the reconfigurable hardware.

For each operation op, we consider a set of insertion edges $\delta_{op} \subseteq E$. The merit of δ_{op} is measured by the function $W_\delta = \sum_{e \in \delta_{op}} w(e)$. Loosely stated, the objective of our algorithm is to move upwards the SET instructions from DEF_{op} on less frequently executed edges δ_{op}, in order to reduce the total number of performed SET instructions. A formal description of this problem is as follows:

PROBLEM *Given a directed, weighted graph $G < N, E, w >$ and a set of hardware operations HW, each defined in $DEF_{op} \subseteq N$ and with conflicts in $Conflict_{op} \subseteq N$, find a set of insertion edges $\delta \subseteq E$ for each op $\in$ HW which minimizes W_δ under the following constraints:*

—*$\forall$ n $\in$ DEF_{op}, for all paths from* **entry** *to n, there is an insertion edge (u,v) $\in$ δ,*

—*$\nexists$ k $\in$ $Conflict_{op}$ such that k is included in any subpath from v to n*

The minimization of W_δ assures that a smaller or equal number of SET instructions will be performed in the final CFG graph than in the input graph. The first constraint reflects the requirement that hardware must be first configured (using the SET instruction) on all paths before the operation can be performed (using the EXECUTE instruction). The second constraint assures that no conflict operation will change the hardware configuration before the operation execution.

4.2 Instruction Scheduling Algorithm

The problem of removing redundant hardware configurations is similar to the well-known problem of removing redundant expressions. As hardware configurations do not cause any exception, we can use an aggressive speculative scheduling for the hardware configurations in order to anticipate them on less executed paths and thus, to make redundant the hardware configurations from frequently executed paths. We introduce the scheduling algorithm that solves the problem defined in the previous section in three steps. In the first step, the subgraphs where the hardware configurations can be anticipated are constructed. Next, a minimum s-t cut algorithm is applied to find the optimal insertion edges δ_{op} for each hardware operation. Finally, a switch from hardware to software execution is introduced for the cases when the expense of hardware configurations in the newly inserted nodes still exceeds the performance gain of hardware execution.

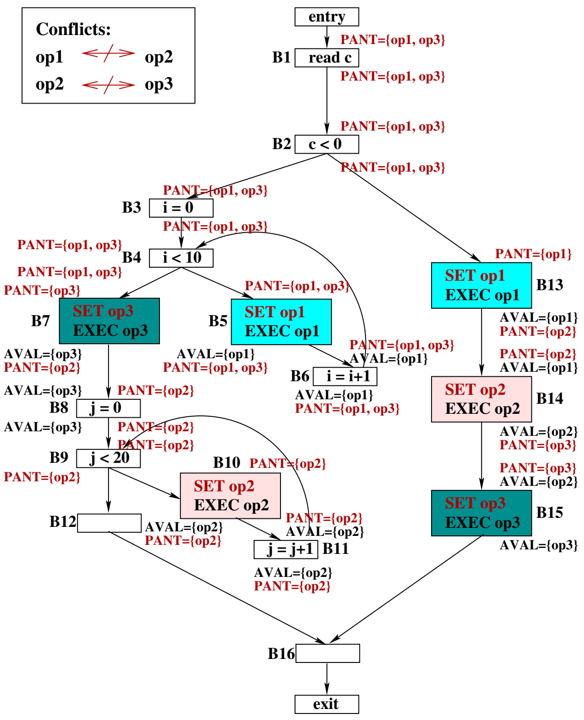

Fig. 5. Set of PANT and AVAL values for the input graph from Figure 4

4.2.1 *Step 1: The Anticipation Subgraph.* Constructing the anticipation graph is a key step in our algorithm. The main goal is to eliminate from the initial graph the edges that cannot propagate upwards the hardware configurations due to hardware conflicts. This step contains two uni-directional data-flow analyses (for partial anticipability and availability) and one pass for constructing the anticipation subgraph by removal of non-essential edges.

Partial Anticipability A hardware configuration for operation op is partially anticipated in a point m if there is at least one path from m to the exit node that contains a definition node for op and none of the paths from m to the first such definition node contains a conflict node for op.

A confluence conflict node n is a node with two successors s1 and s2 such that op1 is partially anticipated at the entry point of s1, op2 is partially anticipated at the entry point of s2 and op1 $\nleftrightarrow$ op2. Due to hardware conflicts, op1 and op2 cannot be both anticipated in the confluence conflict node n. We consider a restricted partial anticipability analysis where the confluence conflict nodes limit the partial anticipability for both op1 and op2. This is a backward data-flow problem, where the data-flow equations for a basic block i are defined as follows:

$$PANTin(i) = Gen(i) \cup (PANTout(i) - Kill(i))$$

$$PANTout(i) = \biguplus_{j \in Succ(i)} PANTin(j)$$

$$PANTout(exit) = \emptyset$$

In the first equation, GEN(i) is the set of hardware operations generated in the basic block i. A hardware operation op1 is generated in a basic block i if i $\in DEF_{op}$. The set Kill(i) includes all hardware operations that are in conflict with the operations generated in the basic block i. A hardware operation op $\in$ PANTin(i) is partially anticipated at the entry of a basic block i if it is generated in i or if it is partially anticipated at the exit of i and it is not killed in i.

The second equation differs from standard data-flow equations involved in iterative data-flow analysis where the join operator is $\bigcup$ or $\bigcap$. The operator $\uplus$ is a conditional reunion that excludes the conflicting hardware operations and defined as follows:

$A \uplus B = \{x \in A \bigcup B | \nexists y \in A \bigcup B, x \nleftrightarrow y\}$

This operator is used to stop the partial anticipability of the operations with hardware conflicts at confluence points. A hardware operation op $\in$ PANTout(i) is partially anticipated at the exit of a basic block i if it is partially anticipated at the entry of any successor of i and i is not a conflict confluence node for op. In Figure 5, we present the values for PANT for the input graph presented in Figure 4. For the basic blocks where these values are missing, there are implicitly assumed as $\emptyset$.

Availability We use the standard forward data-flow analysis for availability described by the set of data-flow equations:

$$AVALout(i) = Gen(i) \cup (AVALin(i) - Kill(i))$$
$$AVALin(i) = \bigcap_{j \in Pred(i)} AVALout(j)$$
$$AVALin(entry) = \emptyset$$

This analysis is used to eliminate the hardware configurations when they are already available. The values for AVAL for our example graph are presented in Figure 5.

Constructing the Anticipation Graph Based on the previously presented data-flow analysis results, for each operation op $\in$ HW, we eliminate from the initial graph the nodes which are not essential as follows. We call an edge (u,v) an essential edge for op if $Ess(u,v) = (u,v) \in E \wedge op \notin AVALout(u) \wedge op \in PANTin(v)$. The reduced graph G_{rd} contains the nodes $N_{rd} = \{n \in N | \exists m \in N, Ess(n,m) \vee Ess(m,n)\}$ and the edges $E_{rd} = \{(u,v) \in E | Ess(u,v)\}$. The reduced graph may contain a set of disconnected subgraphs. In order to connect them, we introduce a new pseudo entry node (called s) and a pseudo exit node (called t) and the edges $E_{st} = \{(s,n) | n$ has no predecessor in $N_{rd}\} \bigcup \{(n,t) | n$ has no successor in $N_{rd}\}$ with infinite execution frequency. For our example from Figure 4, the anticipation graphs are presented in Figure 6.

4.2.2 *Step 2: Minimum s-t Cut.* In this step, the set of insertion edges from our problem definition is determined by applying a minimum s-t cut algorithm. The purpose of the min cut algorithm is to select the less frequently executed edges from the anticipation graph on all paths to the definition nodes. In consequence, the min cut algorithm assures the minimization requirement and the first constraint from our problem definition, while the construction of the anticipation graph secures the second constraint.

One of the important advantages of using a min cut algorithm is to avoid moving upwards SET instructions on edges inside loops. In our implementation, we used Edmonds-Karp minimum s-t cut algorithm. For the three hardware operations from Figure 4, their minimum cuts are presented in Figure 6. We notice that for op3 (depicted in Figure 6 (c)), the SET instruction from B7 can propagate further then B2 (on edge (B1, B2)). The minimum cut algorithm chooses the edge (B2, B3) as its execution frequency is smaller (10 versus 20 for (B1,B2)).

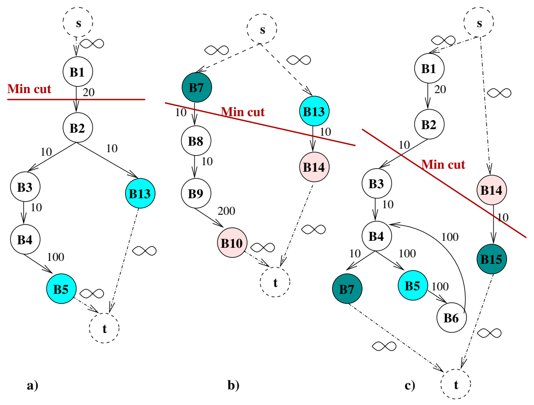

Fig. 6. The anticipation graph for op1 (a), op2 (b) and op3 (c) from Figure 4

4.2.3 *Step 3: Selection of Software/Hardware Execution.* When, even after our scheduling, the hardware configuration and execution is more expensive than the pure software execution, the scheduling algorithm can switch for this operation from hardware execution to software execution. In this case, all the SET instructions for this operation are eliminated and its EXECUTE instructions are replaced by standard calls to the associated software function. In our example from Figure 4, op3 may be in this case if one hardware configuration and one hardware execution is more expensive than one software execution. For the rest of the hardware operations, all associated SET instructions will be removed and new equivalent SET instructions will be introduced on the corresponding insertion edges. For example, in Figure 4, the SET op1 instructions from B5 and B13 will be replaced by a new SET op1 instruction on the (B1, B2) edge as a result of the min-cut algorithm from Figure 6 (a).

This step requires additional information about hardware/software execution/configuration time to be provided in the FPGA description file. As the selection of the software execution for one hardware operation significantly reduces the number of FPGA-area placement conflicts, our future work will address algorithms for the optimal selection of HW/SW execution.

5. EXPERIMENTAL RESULTS

The target FCCM of our experiments is the Molen prototype implemented on the Virtex II Pro platform FPGA of Xilinx described in [Kuzmanov and Vassiliadis 2003]. In this implementation, the GPP is the IBM PowerPC 405 processor at 250 MHz immersed into the FPGA fabric. The application domain of these experiments is video data compressing. We consider a real-life application namely Motion JPEG (M-JPEG) encoder which compresses sequences of video frames applying JPEG compression to each frame. The M-JPEG implementation is based on the public domain implementation described in ”PVRG-JPEG CODEC 1.1”, Portable Video Research Group, Stanford University.

The experimental validation comprises two steps. We first validate the Molen compiler by presenting a complete design flow experiment and measuring the performance improvement for the Molen implemention. The second part assesses the performance impact of the proposed scheduling algorithm for the hardware configuration instructions.

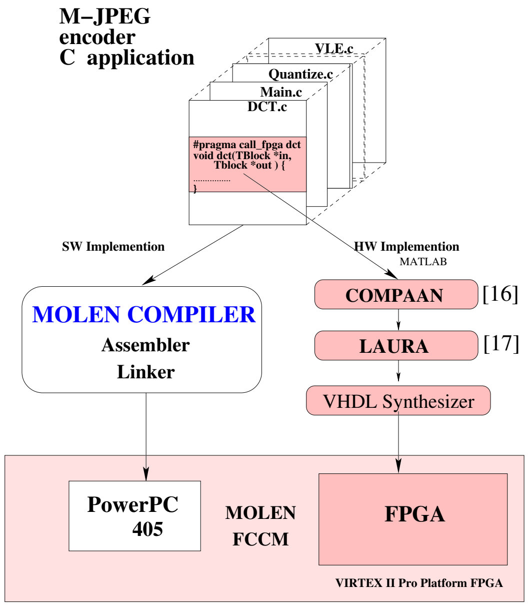

Fig. 7. The design flow

5.1 Molen Compiler Validation

In this subsection, we report the performance improvements of the Molen implementation on the Virtex II Pro for the multimedia video frame M-JPEG encoder. The instruction scheduling algorithm presented in Section 4 is not included in this experiment.

Design Flow: The design flow used in our experiments is depicted in Figure 7. In the target application written in C, the software developer introduces pragma annotations for the functions implemented on the reconfigurable hardware. These functions are translated to Matlab and processed by the COMPAAN[Kienhuis et al. 2000]/LAURA[Zissulescu et al. 2003] toolchain to automatically generate the VHDL code. The commercial tools can then be used to synthesize and map the VHDL code on the target FPGA. The application is compiled with the Molen compiler and the executable is loaded and executed on the target Molen FCCM.

M-JPEG, Software and Hardware Implementations: The most demanding function in M-JPEG application is 2D DCT extended with preshift and bound transforms (named in this paper as DCT*). In consequence, DCT* is the first function candidate for hardware implementation. The only modification of the M-JPEG application that indicates the reconfigurable DCT* execution is the introduction of the pragma annotation as presented in Figure 7. The hardware implementation for execution of the DCT* function on the reconfigurable hardware is described in [Stefanov et al. 2004]. The VHDL code is automatically extracted from the DCT* application code using COMPAAN[Kienhuis et al. 2000]/LAURA[Zissulescu et al. 2003] tools. The Xilinx IP core for DCT and the ISE Foundation [ISE] are used to synthesize and map the VHDL code on the FPGA. After the whole application is compiled with the Molen compiler, in the final step we use the GNU assembler and linker and the C libraries included in the Embedded Development Kit

Name	# frames	Resolution [pixels]	Format	Color/BW
tennis	8	48x48	YUV	color
barbara	1	48x48	YUV	color
artemis	1	48x48	YUV	color

Table I. M-JPEG video sequences

		tennis [0-7]		barbara	artemis
		MIN	MAX		
Pure Software Execution (a)	M-JPEG	33,671,821	33,779,813	34,014,157	34,107,893
	DCT*	1,242,017	1,242,017	1,242,017	1,242,017
	DCT* cumulated	22,356,306	22,356,306	22,356,306	22,356,306
	Maximal improvement	66.18%	66.39%	65.73%	65.55%
Execution on Molen prototype (b)	M-JPEG	13,443,269	13,512,981	13,764,509	13,839,757
	DCT* HW	4,125	4,125	4,125	4,125
	DCT* HW + Format conv.	102,589	102,589	102,589	102,589
Comparison (c)	Practical speedup	2.50	2.51	2.47	2.46
	Theoretical speedup	2.96	2.98	2.92	2.90
	Efficiency	84.17%	84.65%	84.70%	84.91%

Table II. M-JPEG execution cycles and comparisons

(EDK) [EDK] from Xilinx to generate the application binary files.

Performance Measurements: The current Molen implementation is a prototype version, which imposes the following constraints:

—the memory size for *text* and *data* sections is currently limited to maximum 64K. In order for the M-JPEG executable to fulfill these limitations, we rewrote the original application preserving only the essential features for compressing sequences of video frames. Moreover, these limitations also restrict the size of the input video frames to 48x48 pixels. The input video-frames used in these experiments are presented in Table I.

—dynamic reconfiguration is not supported (yet) on the Molen prototype. In consequence, we could not measure the impact on performance of repetitive hardware configurations.

In addition, the performance measurements have been performed given the following additional conditions:

—the input/output operations are extremely expensive for the current Molen prototype, due to the standard serial connection implemented by UART at 38400 bps between the Molen prototype and the external environment; this limitation can be removed by the implementation of faster I/O system. We therefore did not include the I/O operation impact in our measurements as they are not relevant for the RC paradigm.

—the DCT* hardware implementation requires a different format for the input data than the software implementation. Consequently, an additional data format conversion is performed in software before and after the DCT* execution on reconfigurable hardware.

—taking into account that the target PowerPC processor included in the Virtex-II Pro platform does not provide hardware floating-point support and that the required floating-point software emulation is extremely expensive, the integer DCT is used for both software and hardware implementation to allow a fair comparison.

The execution cycles for M-JPEG encoder and comparisons are presented in Table II. As we considered a sequence of 8 video frames for *tennis* input sequence, we present only the minimal and maximal values for each measurement in order to avoid redundant information.

Pure Software Execution: As a point of reference, we present in Table II(a), the results of our measurements performed on the PowerPC processor from the Virtex II Pro platform, without DCT* hardware acceleration. In row 1, the number of cycles used for executing the whole M-JPEG application is given. In row 2, the cycles consumed by one execution of the DCT* function are given and the next row contains the total number of cycles spent in DCT*. From these numbers, we can conclude that 66% of the total execution time is spent in the DCT* function, given the input set. This 66% represents the maximum (theoretical) improvement that can be obtained by hardware acceleration of the DCT* function. The corresponding theoretical speedup - using *Amdahl's law* - is presented in Table II(c), row 2. The ratio of the measured speedup to this theoretical speedup is referred in this paper as the performance efficiency.

Execution on the Molen prototype: In Table II(b), we present the number of cycles for the M-JPEG execution on the Molen prototype. From row 1 we can conclude that an overall speedup of 2.5 (Table II(c), row 1) is achieved. The DCT* execution on the reconfigurable hardware takes 4125 cycles (row 2) which is around 300 times less than the software based execution on the PowerPC processor (Table II(a), row 2). However, due to the data format conversion required by the DCT* hardware implementation, the overall number of cycles for one DCT* execution becomes 102,589 (Table II(b), row 3), resulting in a 10 fold speedup for DCT* and a 2.5 speedup globally. The performance efficiency is about 84% as presented in Table II(c), last column. It is noted that this efficiency is achieved even though i) the hardware implementation has been automatically but non-optimally obtained (using COMPAAN[Kienhuis et al. 2000]/LAURA[Zissulescu et al. 2003] tools) and ii) additional software data conversion diminished the DCT* speedup in hardware. From these measurements, we can conclude that even non-optimized implementation can be used to achieve considerable performance improvements. In addition, taking into account that only one function (DCT*) is executed on the reconfigurable hardware, we consider that an overall M-JPEG speedup of 2.5x from the theoretical speedup of 2.96 x confirm the viability of the presented approach.

5.2 Instruction Scheduling Impact on Performance

The presented instruction scheduling algorithm has been implemented as a MachSUIF pass [MachineSUIF] within the Molen compiler. The overall time-complexity of the scheduling algorithm is mainly determined by the two data-flow analyses and the min s-t cut algorithm. For the data-flow analyses, the time complexity in the worst-case is $O(|N|x(b+2))$, where b is the maximum number of back edges on any acyclic path in the graph G. The time complexity of the Edmonds-Karp algorithm is $O(|N|x|E|^2)$, thus the proposed scheduling algorithm has an overall polynomial time complexity. The compilation time spent in the scheduling phase for the considered applications is negligible (around 1 s) compared to the

	HW Execution			SW Execution	
Op Name	EXEC [cycle]	Area [slice]	SET [cycle]	One call [cycle]	%Total M-JPEG
DCT	416	848	431771	44396	80 %
Quant	73	397	202073	1494	3 %
VLC	272	193	98237	6921	12.5 %

Table III. HW/SW features for the operations that candidate for hardware implementation

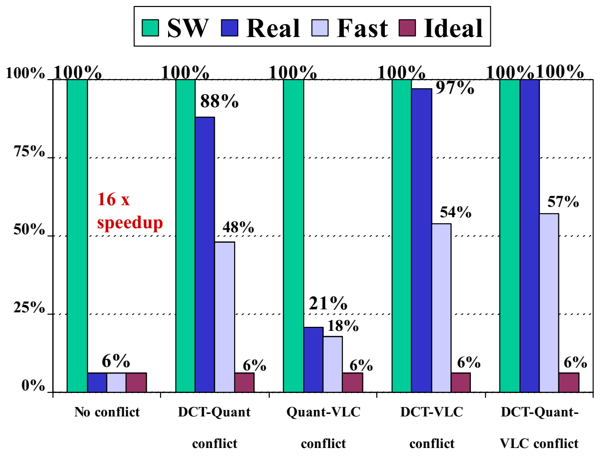

Fig. 8. Comparison of estimated performance for our scheduling algorithm for M-JPEG benchmark

overall compilation time.

The input sequence for the multimedia benchmark M-JPEG encoder in this case study contains 30 color frames from "tennis" in YUV format with a resolution of 256x256 pixels. The operations performed on the FPGA are DCT (2-D Discrete Cosine Transform which is different from DCT* with preshift and bound transforms), Quantization and VLC (Variable Length Coding), which are the main kernels in the considered application. The Xilinx IP cores for DCT [Pillai 2002], Quantization [Pillai 2003a] and VLC [Pillai 2003b] are used for hardware implementations. Due to the limitation of the Molen prototype (e.g. lack of support for dynamic reconfiguration and reduced image size), the results presented in this subsection are not measured on the real board as in the previous experiment, but they are based on simulation using the PowerPC simulator from Simics [Magnusson et al. 2002].

We present in Table III the characteristics of DCT, Quantization and VLC hardware and software executions. Based on the characteristics of the XC2VP20 chip, for which a complete configuration of 9280 slices takes about 20 ms, we estimated the configuration time for each operation (Table III, column 4) in terms of PowerPC processor cycles. Comparing the values from Table III (column 4 and 5), we notice that the hardware configuration alone is about 10 times more expensive than the complete software execution. Using Amdhal's law, we determine that the simple scheduling for DCT will slowdown the M-JPEG benchmark up to 10x. For this reason, we compare the performance of our scheduling algorithm to the pure software approach rather than the inefficient simple scheduling.

The estimated performance for the M-JPEG application for different possible conflicts between the three hardware operations are presented in Figure 8. The considered conflicts are to be expected for different FPGA area sizes or predefined FPGA area allocations. The standard unit of this comparison is the pure software execution (SW) when the M-JPEG benchmark is completely performed on the GPP alone. The performance of our instruction scheduling algorithm for the real Xilinx hardware implementations is denoted as REAL. As recently some hardware approaches [Blodget et al. 2004] have been proposed for reducing the hardware configuration time, we also analyze the impact of our scheduling algorithm when the hardware configuration is accelerated by a factor of 20x[1] compared to the current values from Table III, column 4. The performance of our instruction scheduling algorithm combined with this faster hardware configuration is presented in Figure 8 as FAST. For completeness, we also present the IDEAL case when the hardware configuration is performed in zero cycles.

We notice that for the ”no conflict” case, the performance improvement is about 94 % (equivalent to a 16x speedup) for both REAL and FAST scheduling and very close to the IDEAL performance. In this case, the instruction scheduling algorithm moves the hardware configurations for all three operations at the procedure entry point. In consequence, there is only one hardware configuration for each hardware operation, thus the difference between REAL and FAST is negligible.

For the rest of the ”conflict” cases, the scheduler for REAL will switch from hardware execution to software execution for the conflicting operations. For example, when there is a DCT - Quantization conflict, the scheduler will move both DCT and Quantization operation in software, while the third non conflicting operation VLC remains in hardware; its hardware configuration needs to be performed only once, at the procedure entry point.

For the FAST scheduling, even when one operation has a conflict, it may remain in hardware, thanks to the 20x faster hardware configuration. For the case with DCT - Quantization - VLC conflicts, both DCT and VLC are performed in hardware and produce a performance improvement of 43 % as the fast hardware configuration does not consume all performance gain of the hardware execution. The scheduler selects the software execution for Quantization, in order to prevent a performance decrease produced by its hardware configuration and execution (16 % for Quantization). Therefore, the performance improvement for simple scheduling (all operations executed on the reconfigurable hardware) and 20x faster reconfigurations is 27 % while our scheduling algorithm contributes to a performance improvement between 43 % and 94 % when compared to a pure software .

In consequence, we notice that for the non-conflict case, our algorithm capitalizes the maximum performance gain that can be obtained by hardware execution of the considered operations. Finally, the results presented in Figure 8 emphasize the important performance impact of our scheduling algorithm even for the future faster FPGAs.

6. CONCLUSIONS

In this paper, we presented the implemented compiler support for the Molen prototype on the Virtex II platform FPGA. The compiler allows the automatic modification of the application source code based on pragma annotation and using the extensions following the Molen Programming Paradigm. We have also introduced a general scheduling algorithm for hardware configuration instructions. This algorithm takes into account specific features of the reconfigurable hardware such as the FPGA-area placement conflicts and the

[1] The factor has been chosen arbitrarily. Mutatis mutandis, similar observations will then hold.

reconfiguration latencies of each hardware operation. Based on the characteristics of the compiled application, the scheduling reduces the number of performed hardware configurations preserving the application semantics. It combines advanced compiler techniques with powerful graph theory algorithms.

The results of our case studies show that the performance is dramatically improved by using our scheduling algorithm, and this improvement will hold for future faster FPGA platforms. The experiment also evaluated the effectively realized speedup of reconfigurable hardware execution of the DCT* function of the M-JPEG application. The generated code was executed on the Molen prototype and showed a 2.5 speedup. This speedup consumed 84% of the total achievable speedup which amounts to 2.9. Taking into account that hardly any optimization was performed and only one function ran on the reconfigurable fabric, a significant performance improvement was nevertheless observed.

Further research on the compiler will address optimizations for dynamic configurations and parallel execution on the reconfigurable hardware. Another issue is to improve the scheduling algorithm by extending the data-flow analysis to propagate a conflicting operation beyond the confluence conflict points. We also investigate the integration of FPGA area allocation in our scheduling.

REFERENCES

BLODGET, B., BOBDA, C., HUEBNER, M., AND NIYONKURU, A. 2004. Partial and dynamic reconfiguration of xilinx virtex-ii fpgas. In *FPL*. Vol. 3203. Springer-Verlag Lecture Notes in Computer Science (LNCS), Antwerp, Belgium, 801–810.

BOLOTSKI, M., DEHON, A., AND KNIGHT, J. T. F. 1994. Unifying FPGAs and SIMD arrays. In *ACM/SIGDA Symposium on FPGAs*. Berkeley, CA, 1–10.

BONDALAPATI, K., DINIZ, P. C., DUNCAN, P., GRANACKI, J., HALL, M., JAIN, R., AND ZIEGLER, H. 1999. DEFACTO: A design environment for adaptive computing technology. In *IPPS/SPDP Workshops*. 570–578.

CAI, Q. AND XUE, J. 2003. Optimal and efficient speculation-based partial redundancy elimination. In *ACM CGO*. San Francisco, California, 91–102.

CAMPI, F., CAPPELLI, A., GUERRIERI, R., LODI, A., TOMA, M., ROSA, A. L., LAVAGNO, L., AND PASSERONE, C. 2003. A reconfigurable processor architecture and software development environment for embedded systems. In *Proceedings of Parallel and Distributed Processing Symposium*. Nice, France, 171–178.

EDK. http://www.xilinx.com/ise/embedded/edk.htm.

GOKHALE, M. B. AND STONE, J. M. 1998. Napa C: Compiling for a Hybrid RISC/FPGA Architecture. In *Proceedings of FCCM'98*. Napa Valley, CA, 126–137.

ISE. "http://www.xilinx.com/ise_eval/index.htm".

KASTRUP, B., BINK, A., AND HOOGERBRUGGE, J. 1999. Concise: A compiler-driven cpld-based instruction set accelerator. In *Proceedings of FCCM'99*. Napa Valley CA, 92–100.

KIENHUIS, B., RIJPKEMA, E., AND DEPRETTERE, E. 2000. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proc. of CODES'2000*. San Diego, CA, 13–17.

KUZMANOV, G. AND VASSILIADIS, S. 2003. Arbitrating Instructions in an ρμ-coded CCM. In *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL'03)*. Vol. 2778. Springer-Verlag Lecture Notes in Computer Science (LNCS), Lisbon, Portugal, 81–90.

LEE, M.-H., SINGH, H., LU, G., BAGHERZADEH, N., AND KURDAHI, F. J. 2000. Design and Implementation of the MorphoSys Reconfigurable Computing Processor. *VLSI Signal Processing Systems 24*, 147–164.

MACHINESUIF. "http://www.eecs.harvard.edu/hube/software".

MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HALLBERG, G., HOGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. 2002. Simics: A full system simulation platform. *IEEE Transactions on Computers 35(2)*, 50–58.

MOSCU PANAINTE, E., BERTELS, K., AND VASSILIADIS, S. 2003. Compiling for the Molen Programming Paradigm. In *13th International Conference on Field Programmable Logic and Applications (FPL)*. Vol. 2778. Springer-Verlag Lecture Notes in Computer Science (LNCS), Lisbon, Portugal, 900–910.

MOSCU PANAINTE, E., BERTELS, K., AND VASSILIADIS, S. 2004a. Dynamic hardware reconfigurations: Performance impact on mpeg2. In *Proceedings of SAMOS*. Vol. 3133. Springer-Verlag Lecture Notes in Computer Science (LNCS), Samos, Greece, 284–292.

MOSCU PANAINTE, E., BERTELS, K., AND VASSILIADIS, S. 2004b. The PowerPC backend molen compiler. In *FPL*. Vol. 3203. Springer-Verlag Lecture Notes in Computer Science (LNCS), Antwerp, Belgium, 434–443.

PILLAI, L. 2002. Video compression using dct. In *Application Note: Virtex-II Series*. Xilinx, http://direct.xilinx.com/bvdocs/appnotes/xapp610.pdf.

PILLAI, L. 2003a. Quantization. In *Application Note: Virtex and Virtex-II Series*. Xilinx, http://direct.xilinx.com/bvdocs/appnotes/xapp615.pdf.

PILLAI, L. 2003b. Variable length coding. In *Application Note: Virtex-II Series*. Xilinx, http://direct.xilinx.com/bvdocs/appnotes/xapp621.pdf.

ROSA, A. L., LAVAGNO, L., AND PASSERONE, C. 2003. Hardware/Software Design Space Exploration for a Reconfigurable Processor. In *Proc. of DATE 2003*. Munich, Germany, 570–575.

SIMA, M., VASSILIADIS, S., S.COTOFANA, VAN EIJNDHOVEN, J., AND VISSERS, K. 2002. Field-Programmable Custom Computing Machines - A Taxonomy. In *12th International Conference on Field Programmable Logic and Applications (FPL)*. Vol. 2438. Springer-Verlag Lecture Notes in Computer Science (LNCS), Montpellier, France, 79–88.

STEFANOV, T., ZISSULESCU, C., TURJAN, A., KIENHUIS, B., AND DEPRETTERE, E. 2004. System Design using Kahn Process Networks: The Compaan/Laura Approach. In *Proc. of DATE 2004*. Paris, France, 340–345.

SUIF2. "http://suif.stanford.edu/suif/suif2".

TANG, X., AALSMA, M., AND JOU, R. 2000. A Compiler Directed Approach to Hiding Confguration Latency in Chameleon Processors. In *FPL*. Vol. 1896. Springer-Verlag Lecture Notes in Computer Science (LNCS), Villach, Austria, 29–38.

VASSILIADIS, S., GAYDADJIEV, G., BERTELS, K., AND MOSCU PANAINTE, E. 2003. The Molen Programming Paradigm. In *Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation*. Samos, Greece, 1–7.

VASSILIADIS, S., WONG, S., AND COTOFANA, S. 2001. The MOLEN $\rho\mu$-Coded Processor. In *11th International Conference on Field Programmable Logic and Applications (FPL)*. Vol. 2147. Springer-Verlag Lecture Notes in Computer Science (LNCS), Belfast, UK, 275–285.

YE, Z. A., SHENOY, N., AND BANERJEE, P. 2000. A C Compiler for a Processor with a Reconfigurable Functional Unit. In *ACM/SIGDA Symposium on FPGAs*. Monterey, California, USA, 95–100.

ZISSULESCU, C., STEFANOV, T., KIENHUIS, B., AND DEPRETTERE, E. 2003. Laura: Leiden Architecture Research and Exploration Tool. In *13th International Conference on Field Programmable Logic and Applications (FPL)*. Vol. 2778. Springer-Verlag Lecture Notes in Computer Science (LNCS), Lisbon, Portugal, 911–920.