Instruction Scheduling for Hiding Reconfiguration Latency

Elena Moscu Panainte, Koen Bertels, and Stamatis Vassiliadis
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2600 GA Delft, The Netherlands
Phone: +31 15 2786249 Fax: +31 15 2784898
E-mail: {elena|koen|stamatis}@ce.et.tudelft.nl

Abstract— **Reconfigurable computing (RC) is becoming increasingly popular as it bears the promise of combining the flexibility of software with the performance of hardware. Although the huge reconfiguration latency of the available FPGA platforms is a well-known shortcoming of the current Field-programmable Custom Computing Machines (FCCMs), little research in instruction scheduling has been undertaken to eliminate or diminish its negative influence on performance. In this paper, we address such a hardware configuration instruction scheduling when a predefined FPGA area allocation is imposed for the operations executed on the reconfigurable fabric. The algorithm is based on advanced data-flow analyses to anticipate the hardware configurations as soon as possible. The result of the proposed scheduling algorithms are promising as the loop-invariant SET instructions can be moved outside the loop body and redundant hardware configuration instructions (when the FPGA is already configured for the target operation) may be eliminated. In consequence, for real applications when the code efficiency prevails over the compilation time, the proposed scheduling offers an appropriate solution for hiding the huge reconfiguration latency of the current FPGAs. More powerfull compiler techniques are required in order to prevent the current scheduling algorithm from introducing SET instructions inside loops.**

Keywords— **Reconfigurable architecture, compiler, instruction scheduling, reconfiguration latency**

I. 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 assumes that the developer has a deep understanding of both software and hardware development before the benefits of this technology can be exploited. Moreover, detailed knowledge about the target applications are required for an efficient usage of the reconfigurable hardware. The Delft Workbench is an initiative that investigates the integration and development of tools supporting the different design phases starting at code profiling, compilation, 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 compiler instruction scheduling of hardware configurations. An important drawback of RC paradigm is the huge reconfiguration latency of the actual FPGA platforms. In consequence, powerful algorithms for instruction scheduling of the hardware configuration instructions that take into account the characteristics of the reconfigurable hardware are highly required. When multiple operations are performed on the reconfigurable hardware, they may conflict each other due the limited FPGA area available or to the predefined FPGA area allocation imposed by the different reconfigurable hardware implementations. Thus, the instruction scheduling of hardware configurations must analyze this new type of conflict (called in the rest of the paper "FPGA-area placement conflict"). The scheduling algorithm proposed in this paper is based on advanced data-flow analysis to determine how far hardware configurations can be anticipated without "FPGA-area placement conflicts".

The paper is organized as follows. In the following section, we present background information and related work. In section III, we describe the problems for instruction scheduling of hardware configurations and the solution proposed in this paper. Section IV introduces the instruction scheduling algorithm in details. Finally, we discuss conclusions and ongoing work.

II. Background and Related Work

In this paper, we assume the Molen programming paradigm [1] for RC, which 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 (for the minimal πISA as defined in [1])

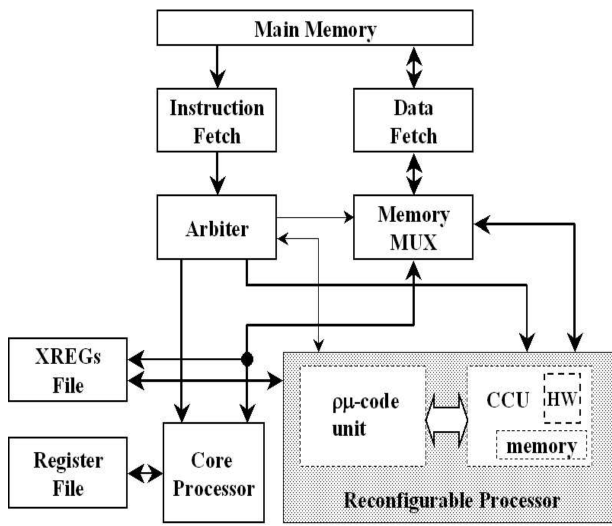

Fig. 1. The Molen machine organization

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**) an hardware implementation can be loaded and executed in the reconfigurable processor. The **movtx** and **movfx** instructions are needed to provide the communications between the reconfigurable hardware and the general-purpose processor (GPP). The Molen machine organization [2] 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. A detailed description of how the Molen organization and programming paradigm compare with other approaches is presented in [3].

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. [4] [5]) 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 and it is a time-consuming [6] and error-prone process. In order to facilitate the design and development process, much effort is put in the development of automated tools (compilers) to perform such tasks [7] [8] [9]. 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 [10] 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 target reconfigurable architecture.

In Figure 2, we present an example of code generated by the Molen compiler for a function call when the target function is performed on the reconfigurable hardware. This function is annotated with a pragma directive *call_fpga* as presented in Figure 2(a). In Figure 2(b), it is presented the standard function call while in the last part of the picture it is depicted the code generated by the compiler extended for the Molen programming paradigm. The function call is replaced with the appropriate instructions for sending parameters to the reconfigurable hardware in XRs, hardware reconfiguration, execution of the operation and the transfer of the result back to the GPP. In the rest of the paper, the consecutive generated SET/EXECUTE instructions are referred as the simple scheduling.

III. MOTIVATION

As for scheduling the hardware configuration instruction, a number of observation can be made, as shown in Figure 3. The first observation is that a straightforward approach where hardware configuration immediately precedes hardware execution would produce a large number of redundant hardware configuration. Additionally, it can be noticed that moving upwards (anticipating) the SET instruction outside the loop body (a) or just removing consecutive SET instructions for the same operation (b) would significantly reduce the number of performed hardware configurations. Taking into account that the hardware configuration is extremely slow for the available FPGA platforms [11], these simple transformations may reflect in significant performance gain. 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

C code

```
#pragma call_fpga op1
int f(int a, int b){
int c,i;
c=0;
for(i=0; i<b; i++)
   c = c + a<<i + i;
c = c>>b;
return c;
}
void main(){
int x,z;
z=5;
x= f(z, 7) ;
}
```

Original MIR code

```
main:
mrk  2,13
ldc   $vr0.s32  <– 5
mov  main.z <– $vr0.s32
mrk  2, 14
ldc   $vr2.s32 <– 7
cal  $vr1.s32 <– f(main.z, $vr2.s32)
mov main.x <– $vr1.s32
mrk  2, 15
ldc   $vr3.s32 <– 0
ret   $vr3.s32
.text_end main
```

MIR code extended with instructions for FPGA

```
mrk     2, 14
mov     $vr2.s32 <– main.z
movtx $vr1.s32(XR) <– $vr2.s32
ldc       $vr4.s32 <– 7
movtx $vr3.s32(XR) <– $vr4.s32
set       address_op1_SET
exec     address_op1_EXEC
movfx $vr8.s32 <– $vr5.s32(XR)
mov     main.x <– $vr8.s32
```

Fig. 2. Code Generation for a function implemented on the reconfigurable hardware

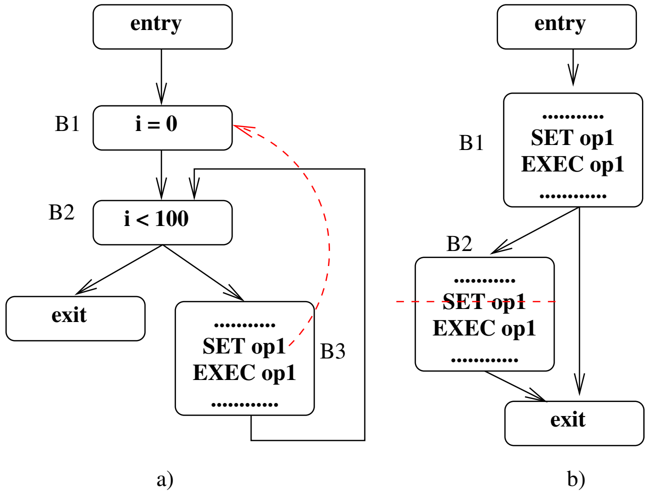

Fig. 3. Anticipation of the hardware configuration, when there is only one operation performed on the reconfigurable hardware

operations can be developed by different IP providers that can impose a predefined FPGA area allocated for each operation, thus ”FPGA-area placement conflicts” between hardware operations are to be expected. In Figure 4(a) we present a possible FPGA area allocation for three operations performed on the FPGA. 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).

Figure 4(b) shows the control-flow graph of a procedure, when op1, op2 and op3 operations are allocated on the reconfigurable hardware as previously presented. Our scheduling algorithm will move upwards the SET instructions for op1 and op3 at the procedure entry point, while the SET op2 instruction cannot be scheduled before B4 as it will change the configuration for op3 before the its execution. The execution of SET op1 at the program entry point implies that this instruction is executed only once per function call (instead of execution at every loop iteration) and the GPP may perform in parallel with the FPGA configuration.

IV. 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 by moving them upwards in the control flow graph. One main aim of our scheduling algorithm is to move outside the loops the hardware configurations of non-conflicting operations. Another important issue is to exploit the parallelism between the GPP and the FPGA. Taking into account that a SET instruction does not stall the GPP, by moving upwards the hardware configuration relative to the hardware execution, the FPGA will be configured in parallel with the execution of GPP instructions between the configuration and execution instructions. In consequence, the reconfiguration latency can be (partially) hided.

In the rest of this section, we first define the control flow graph used in our algorithm. We introduce our scheduling algorithm in two steps. In the first step, the algorithm includes the data-flow analyses for partial anticipability in order to determine how far an hardware configuration can be partially anticipated. Based on the results of the first step, the placement of the SET instructions is computed in the second step.

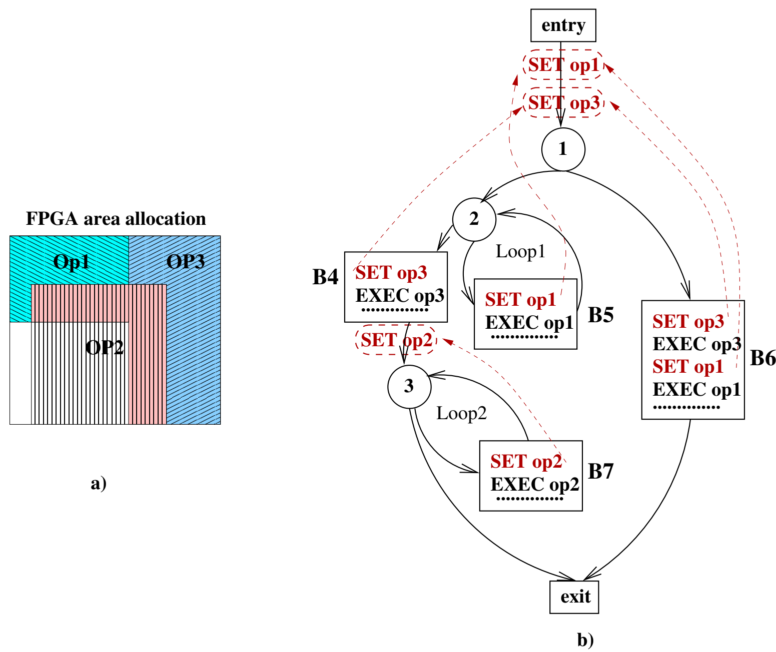

Fig. 4. Motivational example for reducing the number hardware configurations

A. Control Flow Graph

We represent the control flow graph (CFG) of a procedure as a directed graph G $< N, E >$ where the nodes N represent the basic blocks and the edges E represent the control flow dependencies. We define Succ(n)={m ∈ N | (n,m) ∈ E}. A hardware operation op is defined in the basic blocks n ∈ N if n contains an instruction *SET op* immediately followed by *EXEC op* instruction. The basic block n is called a definition node for op. In our example from Figure 4, B4 and B6 are definition nodes for op3. An "FPGA-area placement conflict" between two operations op1 and op2 is represented as op1 $\nleftrightarrow$ op2. We assume that there is no basic block that is a definition node for two conflicting operations; in such cases, the basic block is first divided in multiple basic blocks, one for each conflicting operation. A basic block n is a conflict node for the hardware operation op1 if n is a definition node for a conflicting hardware operation op2, op1 $\nleftrightarrow$ op2. In Figure 4, B4, B5 and B6 are conflict nodes for op2.

B. 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 is a definition node for op1. 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 ∈ 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 $\biguplus$ is a conditional reunion that excludes the conflicting hardware operations and defined as follows: $A \biguplus B = \{x \in A \bigcup B | \nexists y \in A \bigcup B, x \nleftrightarrow y\}$

This operator is used to stop the partial anticipability of the operations with hardware conflicts at confluence points. A hardware operation op ∈ 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

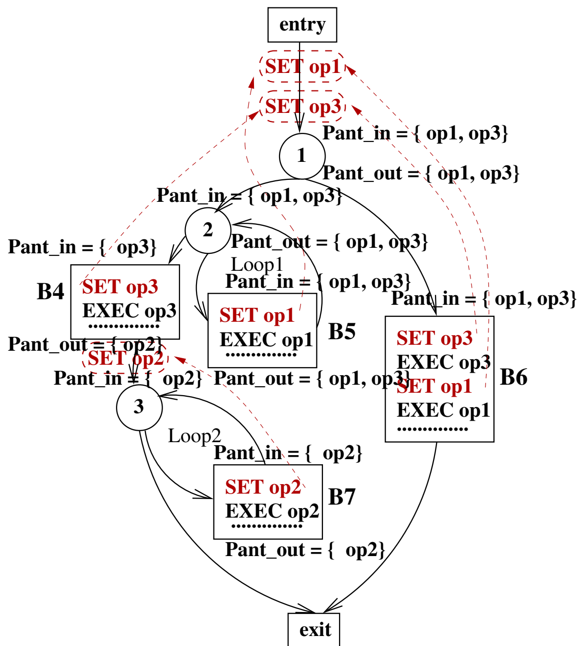

Fig. 5. Set of PANT values for the input graph from Figure 4

```
1   remove_all_SET()

2   for each basic block n ∈ N do
3       if n = entry then
4           //entry node
5           succ = succ(entry)
6           for each op ∈ Pant_in(succ) do
7               insert SET op on the edge (entry, succ)
8       else
9           for each op ∈ Pant_out(n) do
10              if op ∉ Pant_in(n) then
11                  //op killed in n
12                  insert SET op at the end of n

13  for each edge(u,v) ∈ E do
14      for each op ∈ Pant_in(v) do
15          if op ∉ Pant_out(u) then
16              //u = confluence node
17              insert SET op on the edge (u,v)
```

TABLE I
PLACEMENT OF SET INSTRUCTION

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 $\emptyset$.

C. Hardware Configuration Instruction Placement

Based on the results of the previous step, the SET instructions are moved upwards in the control flow graph. In this step, a single SET instruction can be replaced by multiple SET instructions before conflict of confluence conflict nodes, while redundant SET instructions will be eliminated.

The algorithm for the placement of SET instructions is presented in Table I. Initially, all SET instruction of the simple scheduling are removed (line 1). The operations that are partially anticipated at the procedure entry point are inserted in a new basic block following the entry node (lines 3 -7). Insertion of a basic block on one edge includes the appropriate settings of its successors and predecessors. For the operations anticipated at the end of a basic node n but are killed in the same block, the SET instructions are inserted at the end of the basic block n (before control instructions) as presented in lines 9 -12. Finally, at the confluence conflict nodes, the SET instructions for the conflicting operations are inserted on the corresponding edges (lines 13 - 17). In our example from Figure 4, SET op2 from B7 will be moved after B4, while SET op1 and SET op3 instructions are placed after the procedure entry point.

D. Open Issues

As presented in the example from Figure 4, our scheduling algorithm moves outside loops the hardware configurations of non-conflict operations, thus usually reducing the number of required hardware configurations. Another important benefit is the elimination of consecutive hardware configurations. In consequence, moving upwards hardware configurations may produce important performance improvement for the considered cases. Nevertheless, our scheduling algorithm does not take into account the execution frequencies of the inserted hardware configurations. For example, while moving upwards hardware configurations, our scheduling algorithm may insert such hardware configurations inside a loop, as presented in Figure 6. The SET op3 instructions from B3 which was previously executed only once is moved inside the loop (B1, B3, B4), thus, after the instructions scheduling it is performed 100 times. In order to avoid the performance decreasing produced in such cases, more powerfull compiler techniques and graph algorithms are required.

V. Conclusions and Future Work

In this paper, we have presented an instruction scheduling algorithm that addresses an important drawback of the current FPGA platforms, namely the huge reconfiguration latency. The proposed algorithm tries to hide this latency taking into account the FPGA area placement conflicts between the hardware operations used in one application. Based on the data-flow analysis, the anticipation subgraph for each operation is computed and the corresponding SET

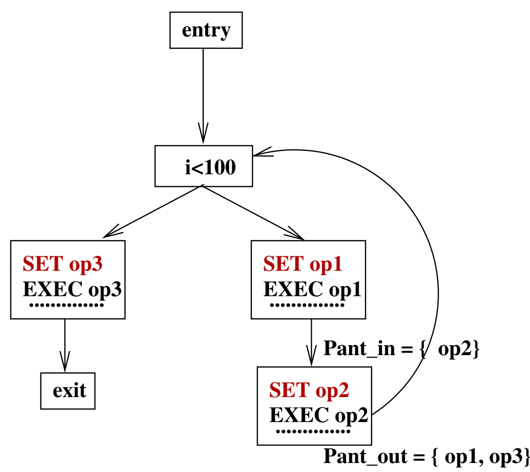

Fig. 6. Limitations of the instruction scheduling algorithm

instructions are moved upwards in the computed subgraph. One important benefit of this transformation is the loop invariant SET instruction motion outside loops. Nevertheless, additional extensions of the scheduling algorithm are required in order to avoid insertion of SET instructions inside loops.

Future work will include compiler optimizations and scheduling of the reconfigurable instructions. Another issue is to exploit the parallelism between hardware operations performed on the FPGA and between the FPGA and GPP. We will also look at interprocedural instruction scheduling of hardware configurations.

References

[1] S. Vassiliadis, G.N. Gaydadjiev, K. Bertels, and E. Moscu Panainte. The Molen Programming Paradigm. In *Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 1–7, Samos, Greece, July 2003.

[2] S. Vassiliadis, S. Wong, and S. Cotofana. The MOLEN $\rho\mu$-Coded Processor. In *11th International Conference on Field Programmable Logic and Applications (FPL)*, volume 2147, pages 275–285, Belfast, UK, Aug 2001. Springer-Verlag Lecture Notes in Computer Science (LNCS).

[3] E. Moscu Panainte, K. Bertels, and S. Vassiliadis. Compiling for the Molen Programming Paradigm. In *13th International Conference on Field Programmable Logic and Applications (FPL)*, volume 2778, pages 900–910, Lisbon, Portugal, Sep 2003. Springer-Verlag Lecture Notes in Computer Science (LNCS).

[4] F. Campi, A. Cappelli, R. Guerrieri, A. Lodi, M. Toma, A. La Rosa, L. Lavagno, and C. Passerone. A reconfigurable processor architecture and software development environment for embedded systems. In *Proceedings of Parallel and Distributed Processing Symposium*, pages 171–178, Nice, France, Apr 2003.

[5] Albert La Rosa, Luciano Lavagno, and Claudio Passerone. Hardware/Software Design Space Exploration for a Reconfigurable Processor. In *Proc. of DATE 2003*, pages 570–575, Munich, Germany, March 2003.

[6] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System Design using Kahn Process Networks: The Compaan/Laura Approach. In *Proc. of DATE 2004*, pages 340–345, Paris, France, Feb 2004.

[7] Maya B. Gokhale and Janice M. Stone. Napa C: Compiling for a Hybrid RISC/FPGA Architecture. In *Proceedings of FCCM'98*, pages 126–137, Napa Valley, CA, April 1998.

[8] Bernardo Kastrup, Arjan Bink, and Jan Hoogerbrugge. Concise: A compiler-driven cpld-based instruction set accelerator. In *Proceedings of FCCM'99*, pages 92–100, Napa Valley CA, April 1999.

[9] Zhi Alex Ye, Nagaraj Shenoy, and Prithviraj Banerjee. A C Compiler for a Processor with a Reconfigurable Functional Unit. In *ACM/SIGDA Symposium on FPGAs*, pages 95–100, Monterey, California, USA, Feb 2000.

[10] Elena Moscu Panainte, K. Bertels, and S. Vassiliadis. The powerpc backend molen compiler. In *FPL*, volume 3203, pages 434–443, Antwerp, Belgium, September 2004. Springer-Verlag Lecture Notes in Computer Science (LNCS).

[11] Elena Moscu Panainte, K. Bertels, and S. Vassiliadis. Dynamic hardware reconfigurations: Performance impact on mpeg2. In *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, volume 3133, pages 284–292. Springer-Verlag Lecture Notes in Computer Science (LNCS), July 2004.