Instruction Scheduling for Dynamic Hardware Configurations

Elena Moscu Panainte Koen Bertels Stamatis Vassiliadis

Computer Engineering
Delft University of Technology, The Netherlands
http://ce.et.tudelft.nl
E-mail: {E.Panainte, K.Bertels, S.Vassiliadis}@et.tudelft.nl

Abstract

Although the huge reconfiguration latency of the available FPGA platforms is a well-known shortcoming of the current FCCMs, little research in instruction scheduling has been undertaken to eliminate or diminish its negative influence on performance. In this paper, we introduce an instruction scheduling algorithm that minimizes the number of executed hardware reconfiguration instructions taking into account the "FPGA area placement conflicts" between the available configurations. The algorithm is based on compiler analyses and feedback-directed techniques and it can switch from hardware execution to software execution for an operation, when the reconfiguration latency could not be reduced. The algorithm has been tested for the M-JPEG encoder application and the real hardware implementations 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.

1. Introduction

The latest commercial FPGA platforms now offer support for partial and dynamic hardware configurations. Nevertheless, one of their main drawback remains the huge reconfiguration latency. In order to hide this latency, 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 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 two hardware configurations that cannot coexist together on the target FPGA.

In this paper, we propose a general instruction scheduling algorithm that automatically minimizes the number of required hardware configurations taking into account both the "FPGA area placement conflicts" and the characteristics of the compiled software application. More specifically, the algorithm anticipates the hardware configurations in less frequently executed application points avoiding the "FPGA area placement conflicts".

The paper is organized as follows. In the following section, we present background information and related work. In section 3, we describe the goals and the contribution of this paper. A formal description of our scheduling problem is included in Section 4. Section 5 introduces the instruction scheduling algorithm. The M-JPEG case study is discussed in Section 6 and finally, we present conclusions and future work.

2. Background and Related Work

In this paper, we assume the Molen programming paradigm [11] [12] for FCCMs (Field-programmable Custom Computing Machines) where the reconfigurable hardware is controlled by two instructions: i) SET for hardware configuration and ii) EXECUTE for hardware execution. The code generated for a hardware operation (an operation performed on the reconfigurable hardware) includes i) parameter passing, ii) the SET instruction, iii) the EXECUTE instruction and iv) returning the computed results. This sequence of instructions where the SET instruction is immediately followed by the associated EXECUTE instruction is referred to in the rest of this paper as the "simple scheduling".

In [5], it has been reported that this simple scheduling produces significant performance decrease due to the huge reconfiguration latency of current FPGA. In order to deal with this drawback, a recent instruction scheduling algorithm has been proposed in [6] for a particular case when there is only one hardware operation in the whole application. The main idea is to move the SET instructions outside

loops in order to eliminate redundant hardware configurations.

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 "FPGA-area placement conflicts". Two hardware operations have an "FPGA-area placement conflicts" (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. In Figure 1(a) we sketch 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).

A compiler approach that considers the restricted case of two consecutive and non-conflicting hardware operations is presented in [10]. 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. Motivation and Contribution

Figure 1(b) shows the control-flow graph of a procedure, when op1, op2 and op3 operations are performed on the reconfigurable hardware and they are placed on the FPGA as presented in Figure 1(a). The numbers associated with each edge of the graph represent the execution frequency of the edge. 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 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. There are no redundant configurations for op3, thus the hardware execution of op3 has to be preceded each time

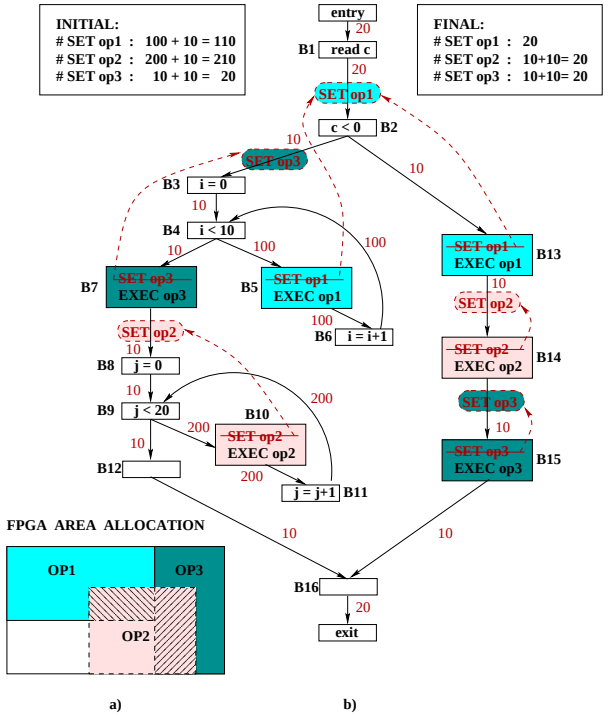

Figure 1. Motivational example for instruction scheduling of hardware configurations (b) with FPGA-area placement constraints (a)

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 its software execution on the GPP (General-Purpose Processor).

In this paper, 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 [2]. 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 Section 5. 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.

4. 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} the set of basic blocks n ∈ N that contain an instruction *SET op* immediately followed by *EXEC op* instruction. A node n ∈ DEF_{op} is called a definition node for op. In our example from Figure 1, 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 ∈ $Conflict_{op}$ is called a conflict node for op. In Figure 1, 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) = ∅, where pred(n)={m ∈ N | (m,n) ∈ E}) and a single **exit** node with no successor (succ(n) = ∅, where succ(n)={m ∈ N | (n,m) ∈ E}). Also, we assume that a node cannot be simultaneous in DEF_{op} and $Conflict_{op}$. 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, 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* ∈ *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),*
- $\nexists k \in Conflict_{op}$ *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 EXECUTE instruction). The second constraint assures that no conflict operation will change the hardware configuration before the operation execution.

5. 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 outperforms the performance gain of hardware execution.

5.1. 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 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 ∈ DEF_{op} . The set Kill(i) includes all hardware operations that are in conflict with the operations generated in the basic block i. A hard-

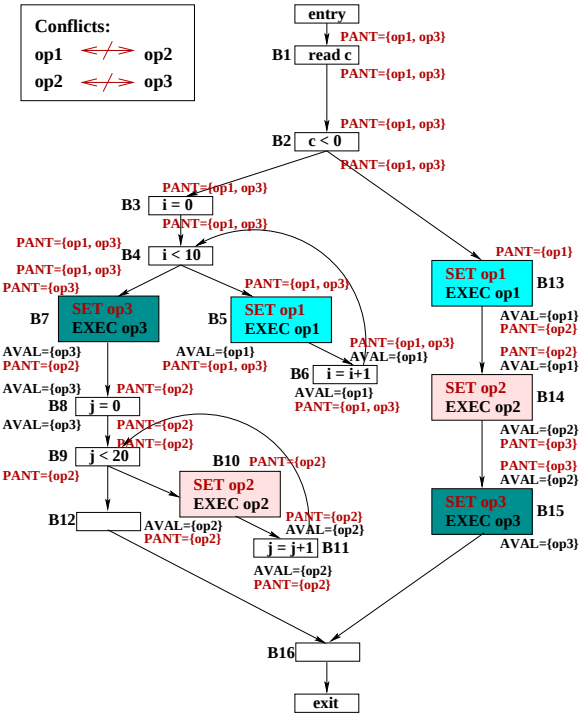

Figure 2. Set of PANT and AVAL values for the input graph from Figure 1

ware 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:

$$A \biguplus B = \{x \in A \bigcup B \mid \nexists y \in A \bigcup B, x \leftrightarrow y\}$$

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 a conflict confluence node for op. In Figure 2, we present the values for PANT for the input graph presented in Figure 1. 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:

$$AVALout(i) = Gen(i) \cup (AVALin(i) - Kill(i))$$

$$AVALin(i) = \bigcap_{j \in Pred(i)} AVALout(j)$$

$$AVALin(entry) = \emptyset$$

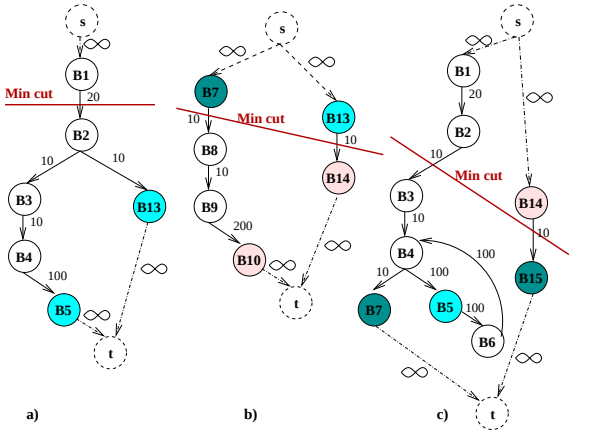

Figure 3. The anticipation graph for op1 (a), op2 (b) and op3 (c) from Figure 1

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 2.

Constructing the Anticipation Graph Based on the previously presented data-flow analysis results, for each operation op ∈ 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 1, the anticipation graphs are presented in Figure 3.

5.2. 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 algorithms 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 op-

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

erations from Figure 1, their minimum cuts are presented in Figure 3. We notice that for op3 (depicted in Figure 3 (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)).

5.3. Step 3: Selection of Software/Hardware Execution

In the cases 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 1, op3 may be in this case if one hardware configuration and one hardware execution is more expensive than one software execution.

6. M-JPEG Case Study

The presented instruction scheduling algorithm has been implemented as a MachSUIF pass [3] within the Molen compiler [6] which generates code for the Molen prototype on the Virtex II Pro FPGA platform. The target C application of this case study is the multimedia benchmark Motion JPEG (M-JPEG) encoder and the input sequence 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), Quantization and VLC (Variable Length Coding). The Xilinx IP cores for DCT [9], Quantization [7] and VLC [8] are used for hardware implementations. The GPP included in the Molen prototype is the IBM PowerPC 405 processor at 250 MHz.

We present in Table 1 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,

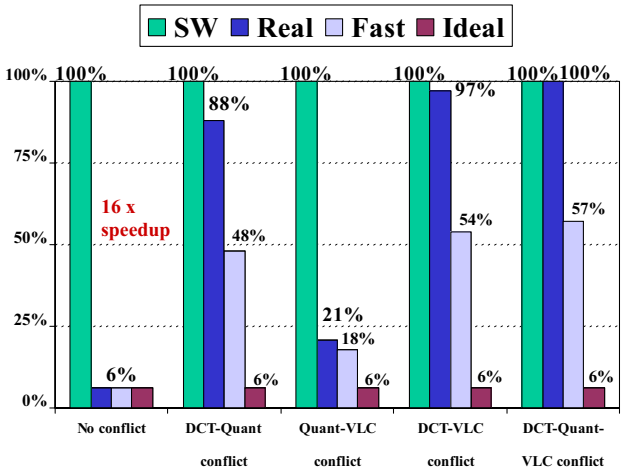

Figure 4. Comparison of estimated performance for our scheduling algorithm for M-JPEG benchmark

we estimated the configuration time for each operation (Table 1, column 4) in terms of PowerPC processor cycles. The profiling results for the software execution from Table 1 are based on simulations using the PowerPC simulator from Simics [4]. Comparing the values from Table 1 (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 4. 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 [1] 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 1, column 4. The performance of our instruction scheduling algorithm combined with this faster hardware configuration is presented in Figure 4 as FAST. For completeness, we also present the IDEAL case when the hardware config-

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

uration 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 % compared to SW.

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 4 emphasize the important performance impact of our scheduling algorithm even for the future faster FPGAs.

7. Conclusions

In this paper, we have 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 reconfiguration latencies of each hardware operations. 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 study show that the performance is dramatically improved by using our scheduling algorithm, and this improvement will hold for future faster FPGA platforms.

When confronted with the choice between the software or hardware execution, our future work will focus on defining the heuristics to guide this selection. Another issue is to allow the data-flow analysis to propagate a conflicting operation beyond the confluence conflict points. We are also looking at incorporating dynamic placement on the reconfigurable hardware in our scheduling.

References

[1] B. Blodget, C. Bobda, M. Huebner, and A. Niyonkuru. Partial and dynamic reconfi guration of xilinx virtex-ii fpgas. In *FPL*, volume 3203, pages 801–810, Antwerp, Belgium, September 2004. Springer-Verlag Lecture Notes in Computer Science (LNCS).

[2] Q. Cai and J. Xue. Optimal and effi cient speculation-based partial redundancy elimination. In *ACM CGO*, pages 91–102, San Francisco, California, 2003.

[3] http://www.eecs.harvard.edu/hube/software.

[4] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Transactions on Computers*, 35(2):50–58, February 2002.

[5] E. Moscu Panainte, K. Bertels, and S. Vassiliadis. Dynamic hardware reconfi gurations: Performance impact on mpeg2. In *Proceedings of SAMOS*, volume 3133, pages 284–292, Samos, Greece, July 2004. Springer-Verlag Lecture Notes in Computer Science (LNCS).

[6] E. 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).

[7] L. Pillai. Quantization. In *Application Note: Virtex and Virtex-II Series*, http://direct.xilinx.com/bvdocs/appnotes/xapp615.pdf.

[8] L. Pillai. Variable length coding. In *Application Note: Virtex-II Series*, http://direct.xilinx.com/bvdocs/appnotes/xapp621.pdf.

[9] L. Pillai. Video compression using dct. In *Application Note: Virtex-II Series*, http://direct.xilinx.com/bvdocs/appnotes/xapp610.pdf.

[10] X. Tang, M. Aalsma, and R. Jou. A Compiler Directed Approach to Hiding Confguration Latency in Chameleon Processors. In *FPL*, volume 1896, pages 29–38, Villach, Austria, Aug 2000. Springer-Verlag Lecture Notes in Computer Science (LNCS).

[11] S. Vassiliadis, G. N. Gaydadjiev, K. Bertels, and E. Moscu Panainte. The molen programming paradigm. In *Proceedings of SAMOS*, volume 3133, pages 1–10, Samos, Greece, July 2003. Springer-Verlag Lecture Notes in Computer Science (LNCS).

[12] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Moscu Panainte. The molen polymorphic processor. *IEEE Transactions on Computers*, 53(11):1363– 1375, November 2004.