Interprocedural Optimization for Dynamic Hardware Configurations

Elena Moscu Panainte, Koen Bertels, and Stamatis Vassiliadis

Computer Engineering Lab
Delft University of Technology, The Netherlands
{E.Panainte, K.Bertels, S.Vassiliadis}@et.tudelft.nl

Abstract. Little research in compiler optimizations has been undertaken to eliminate or diminish the negative influence on performance of the huge reconfiguration latency of the available FPGA platforms. In this paper, we propose an interprocedural optimization that minimizes the number of executed hardware configuration instructions taking into account constraints such as the ”FPGA-area placement conflicts” between the available hardware configurations. The proposed algorithm allows the anticipation of hardware configuration instructions up to the application's main procedure. The presented results show that our optimization produces a reduction of up to 3 - 5 order of magnitude of the number of executed hardware configuration instructions.

1 Introduction

The combination of a general purpose processor (GPP) and a Field Programmable Gate Array (FPGA) is becoming increasingly popular (e.g. [1], [2], [3], [4], [5] and [6]). Reconfigurable computing (RC) is a new style of computer architecture which allows the designer to combine the advantages of both hardware (speed) and software (flexibility). However, an important drawback of the RC paradigm is the huge reconfiguration latency of the actual FPGA platforms. As presented in [7], the potential speedup of the kernel hardware executions can be completely wasted by the repetitive hardware configurations that produce a performance decrease of up to 2 order of magnitude.

When targeting reconfigurable architectures, 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 an interprocedural optimization that anticipates hardware configuration instructions up to the application's main procedure. The optimization takes into account constraints such as the ”FPGA-area placement conflicts” between the available hardware configurations. The presented results show that a reduction of up to 3 - 5 order of magnitude of the number of executed hardware configuration instructions is expected for MPEG2 and M-JPEG multimedia applications.

This paper is organized as follows. Section 2 presents background information and related work for compiler optimizations targeting dynamic hardware configuration instructions, followed by a motivational example in Section 3. The proposed interprocedural optimization algorithm is introduced in Section 4. Experimental results for two multimedia applications are provided in Section 5, and Section 6 presents the concluding remarks.

2 Background and Related Work

In this paper, we assume the Molen programming paradigm ([8], [9])which is a sequential consistency paradigm for programming Field-Programmable Custom Computing Machines(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. It requires only a one time architectural extension of few instructions to provide a large user reconfigurable operation space. The added instructions include **SET** $< address >$ for reconfigurable hardware configuration and **EXECUTE** $< address >$ for controlling the executions of the operations on the reconfigurable hardware. In addition, two MOVE instructions for passing values to and from the GPP register file and the reconfigurable hardware are required.

In order to achieve significant performance improvement for real applications, more operations are usually executed on the reconfigurable 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, resulting in ”FPGA-area placement conflicts”. Two hardware operations have an ”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.

Several aproaches have been proposed for reducing the impact of the reconfiguration latency on performance. 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. In [11], the reconfiguration overhead is reduced by using manual interprocedural optimizations such as localizing memory accesses, partial hardware reuse and pipeling. Our approach is different as the optimization is automatically applied in the compilation phase and it minimizes the number of performed hardware configurations without specific information about the target FPGA and hardware operations. The instruction scheduling approach presented in [12] uses data-flow analyses and profile information for reducing the number of executed hardware configurations. In this paper, we extend this approach at interprocedural level, taking into account all procedures of the target applications. As a consequence, the impact of the proposed optimization is significantly increased as presented in Section 5.

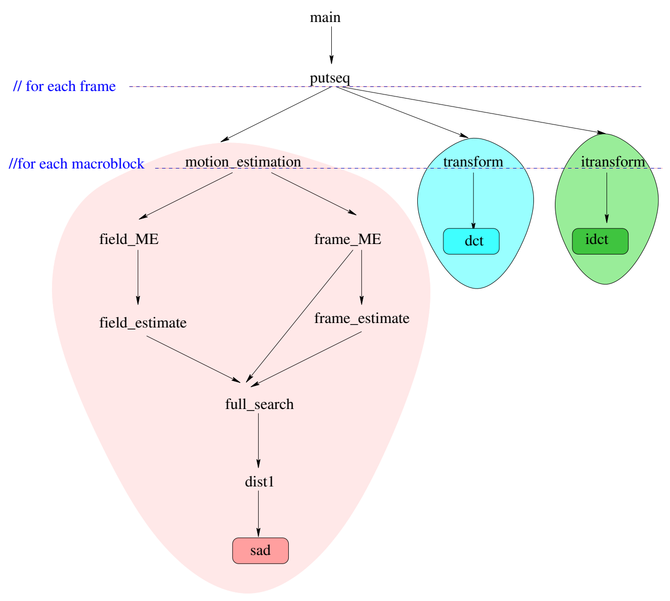

Fig. 1. Motivational example for MPEG2 encoder

3 Motivation and Contribution

In order to illustrate the goals and the main features of the proposed interprocedural optimization, we present in Figure 1 a motivational real example. The presented subgraph is included in the call graph of the MPEG2 encoder multimedia benchmark where an edge $< p_i, p_j >$ represents a call from procedure p_i to procedure p_j. We consider that the procedures SAD, DCT and IDCT are executed on the reconfigurable hardware and that initially the hardware configuration (a SET instruction) is performed before each hardware execution (an EXEC instruction). One first observation is that the configuration for the SAD operation can be safely anticipated in the **motion_estimation** procedure. This anticipation will significantly reduce the number of performed hardware configurations as it will not be performed for each macroblock but only for each frame of the input sequence. This observation also holds for the DCT configuration in **transform** and the IDCT configuration in **itransform**. Moreover, the SAD configuration from **motion_estimation** can be moved upwards in the **putseq** procedure, immediately preceding the call site of **motion_estimation** in **putseq**. Additionally, it can be noticed that the propagation of the SAD configuration from **putseq** to the **main** procedure depends on the FPGA area allocation for SAD, DCT and IDCT. When the SAD operation does not have any FPGA-area placement conflict with the other two hardware operations DCT and IDCT, its configuration can be safely performed only once, at the entry point in the **main** procedure.

The contribution of this paper includes the following. The optimization proposed in this paper allows to anticipate the hardware configurations at interprocedural level, while prior work was limited to optimizations at procedural level (intraprocedural).

Secondly, although the interprocedural optimizations are considered to provide little benefit and significantly increase the compiler complexity, we show that our optimization significantly reduces the number of hardware configurations (a major drawback of the current FPGAs).

4 Interprocedural Optimization for Dynamic Hardware Configurations

The main goal of the proposed interprocedural optimization presented in this section is to anticipate the dynamic hardware configuration instructions taking into account the hardware conflicts between the available hardware operations. As such hardware configuration does not cause an exception, a speculative algorithm is used for anticipating the hardware configuration instructions. The interprocedural optimization consists of three steps. In the first step, the program's call graph is constructed based on an interprocedural control-flow analysis. Next, the set of live hardware configurations for each procedure is determined using an interprocedural data-flow analysis. Finally, the hardware configuration instructions are anticipated in the call graph taking into account the available conflicting operations.

```
Interprocedural Optimization Algorithm

INPUT: Call graph $G =< N,S,r >$, hardware conflicts $f : HWxHW -> \{0,1\}$
OUTPUT: Insertion edges L
1. //Verify assumptions for G
    check if G is DAG

2. //RMOD computation
    traverse G in reverse topological order
        compute for each procedure p
        $RMOD(p) = LRMOD(p) \bigcup_{s \in Succ(p)} RMOD(s)$

 //Compute CF
    for each procedure p
        $CF(p) = \{op_1 \in RMOD(p) | \exists op_2 \in RMOD(p), op_1 \nleftrightarrow op2\}$
3. //Compute the insertion edges
    $L = \emptyset$
    for each edge $< p_i, p_j >$
        for each $op \in [RMOD(p_j) - CF(p_j)] \cap CF(p_i)$
            $L = L \cup < p_i, p_j, op >$
    for each $op \in [RMOD(r) - CF(r)]$
        $L = L \cup < r, r, op >$
```

Table 1. The interprocedural optimization algorithm for hardware configuration instructions

4.1 Step 1: Interprocedural Control-Flow Analysis for Dynamic Hardware Configurations

Starting point of the proposed optimization is the construction of the program's call graph. Given a program P consisting of a set of procedures $< p_1, p_2, ..., p_n >$, the program's call graph of P is the graph $G =< N, E, r >$ with the node set $N = \{p_1, p_2, ..., p_n\}$, the set $E \subseteq N$ x N, where $< p_i, p_j > \in E$ denotes a call site in p_i from which p_j is called, and the distinguished entry node $r \in N$ representing the main entry procedure of the program . An example of a real call (sub)graph is presented in Figure 1.

The construction of the call graph for a program written in C is straightforward as there are no higher-order procedures in the C programming language. For this purpose, we used the *sbrowser_cg* library included in the *suifbrowser* package available in the SUIF environment. The constructed call graph is the input of the optimization algorithm presented in Table 1. As explained in the next subsection, the constructed graph is required to be a DAG (Directed Acyclic Graph) (see Table 1, step 1).

4.2 Step 2: Interprocedural Data-Flow Analysis for Dynamic Hardware Configurations

The goal of the interprocedural data-flow analysis is to determine what hardware operation can modify the FPGA configuration as a side effect of a procedure call. We define LRMOD(p) (Local Reconfigurable hardware MODification) as the set of hardware operations associated with a procedure p. In order to simplify this discussion, we assume that there is at most one hardware operation that can be associated with a procedure. More specifically, $op_1 \in LRMOD(p)$ if there is a pragma annotation that indicates that procedure p is executed on the reconfigurable hardware and its associated hardware operation is named op_1. $RMOD(p)$, Reconfigurable hardware MODification, represents the set of all hardware operations that may be executed by an invocation of procedure p and it can be computed using the following data-flow equation:

$$RMOD(p) = LRMOD(p) \bigcup_{s \in Succ(p)} RMOD(s) \tag{1}$$

A hardware operation op may be performed by calling procedure p if op is associated with procedure p (i.e. $op \in LRMOD(p)$) or if it can be performed by a procedure that is called from procedure p. For an efficient computation, the RMOD values should be computed in reverse topological order (i.e. reverse invocation order) when the call graph does not contain cycles (see step 2 from Table 1). The RMOD values for the example presented in Figure 1 are shown in Figure 2. For the basic blocks where LRMOD values are missing, they are implicitly assumed as $\emptyset$. We notice that by calling *putseq* procedures, all three hardware operations sad, dct and $idct$ may be executed on the reconfigurable hardware.

Due to the increasing complexity of the interprocedural data-flow analysis, this step is performed only when the call graph G satisfies the following criteria. We assume that there are no *indirect procedure calls* (using pointer to functions). These limitations can be eliminated by considering all candidate set of functions that have the same prototype.

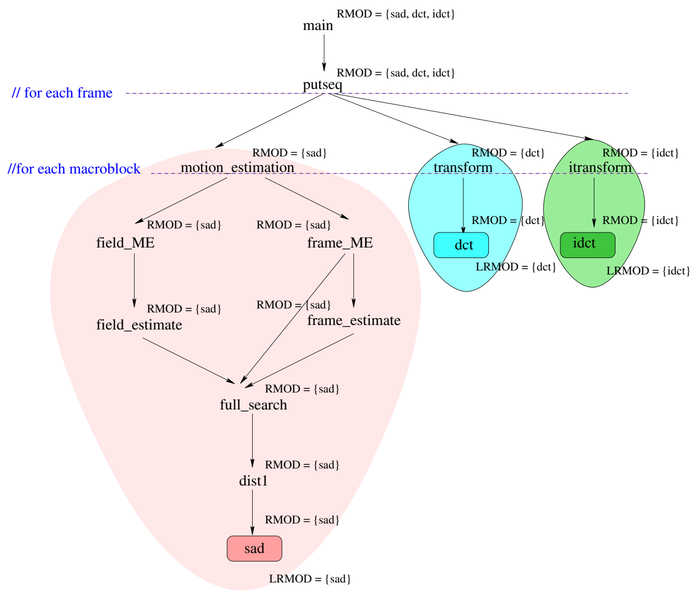

Fig. 2. Interprocedural data-flow analysis for MPEG2 encoder

Another limitation concerns the data-flow equations for procedures with recursive procedure calls (when the call graph contains cycles). In this case, the strongly connected components (scc) should be computed and the data-flow equations should be collapsed for each scc into a single equation. The proposed optimization is applied only when the call graph is a DAG.

4.3 Step 3: Interprocedural Scheduling for Dynamic Hardware Configuration Instructions

In this step, the hardware configuration instructions are anticipated in the call graph taking into account the possible hardware conflicts discovered in the previous step. In the first phase, the set of conflicting operations $CF(p)$ is computed for each procedure included in the call graph based on the *RMOD* values as follows:

$$CF(p) = \{op_1 \in RMOD(p) | \exists op_2 \in RMOD(p), op_1 \not\leftrightarrow op2\} \tag{2}$$

Next, for each edge of the call graph $< p_i, p_j >$, if there is an hardware operation op which does not have conflicts in p_j ($op \notin CF(p_j)$) but it has conflicts in the calling function p_i ($op \in CF(p_i)$), then a SET op instruction is inserted at all call sites of p_j from p_i. Finally, for all non-conflicting operations of the entry node of the call graph G (i.e. $RMOD(r) - CF(r)$), the corresponding SET instructions are inserted at the beginning of the r procedure (see step 3 from Table 1).

The CF values for the example presented in Figure 1 are shown in Figure 3, for the case where all considered hardware operations conflict with each other. For the basic blocks where CF values are missing they are implicitly assumed as $\emptyset$. It can be noticed that the hardware configuration instructions cannot simultaneously propagate upwards of *putseq* procedure due to the considered hardware conflicts.

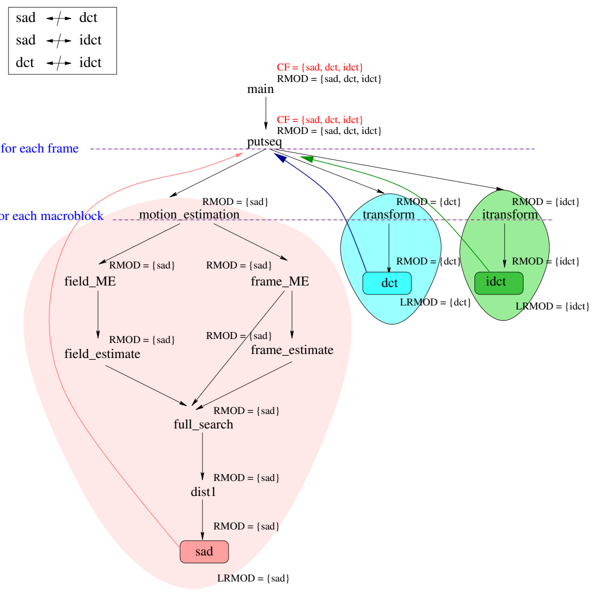

Fig. 3. Interprocedural optimization for MPEG2 encoder

5 Results

In order to present the results of our algorithm with respect to the number of performed hardware configurations, we first describe the experimental setup, including the target architecture and applications. Next, we concentrate on the impact of the optimization on the number of hardware configurations. Finally we present several important observations about the presented results and possible improvements of the optimization algorithm.

5.1 Experimental Setup

The application domain of these experiments is video data compressing. We consider two real-life applications namely Motion JPEG (M-JPEG) encoder which compresses sequences of video frames applying JPEG compression for each frame and the MPEG2 encoder multimedia benchmark. The input test sequence for M-JPEG contains 30 color frames from ”tennis” in YUV format with a resolution of 256x256 pixels. For MPEG2 encoder, we used the standard test frames included in the benchmark. The operations considered for execution on the FPGA for the M-JPEG applications are DCT (2-D Discrete Cosine Transform), Quantization and VLC (Variable Length Coding), while for MPEG2 the candidates are SAD (Sum of Absolute-Difference), DCT and IDCT (inverse DCT).

The described optimization algorithm has been implemented in the Molen compiler, more specifically in the SUIF compiler frontend. We used the *link_suif* pass that combines all input SUIF files, making their global symbol tables consistent and the *sbrowser_cg* library included in the *suifbrowser* package available in the SUIF environment for the construction of the interprocedural call graph. The call graph for the M-JPEG encoder includes 47 nodes (i.e. the applications contains 47 procedures), while the call graph for MPEG2 encoder (a subgraph is presented in Figure 1) has 111 nodes.

5.2 Interprocedural Optimization Results

The aim of the proposed optimization is to significantly reduce the number of the executed SET instructions for each hardware operation. In the results presented in the rest of this section, we compare the number of executed hardware configurations with and without our optimization (denoted as SET_OPT and respectively NO_SET_OP cases).

M-JPEG Encoder Results Table 2 shows the number of hardware configurations required in the M-JPEG encoder multimedia application for the SET_OPT (columns 3-7) and NO_SET_OPT (column 2) cases. When measuring the effects of the proposed optimization (Table 2, columns 3-7), we consider different possible conflicts between DCT, Quant and VLC; in the best case there is no conflict (column 3), while in the worst case all hardware operations are in conflict with each other (column 7). The first observation is that, for the *no conflict* case, our optimization algorithm eliminates all hardware configurations and introduces at the application entry point only one hardware configuration for each hardware operation; thus, all the hardware configurations but one from the initial application (Table 2, column 2) are redundant. A second observation is that our optimization reduces the number of DCT hardware configurations with at least 75 % for all conflict cases. Finally, we notice that even for the worst case (Table 2, columns 7), the proposed optimization reduces the number of executed SET instructions for DCT configuration by 4x. This reduction is due to the anticipation of DCT hardware configuration at the macroblock level, while the configurations for Quant and VLC are already performed at this level and cannot be anticipated upwards due to the hardware conflicts.

MPEG2 Encoder Results The number of hardware configurations for the considered functions in the MPEG2 encoder benchmark is presented in Figure 3. One important observation is the 3-5 order of magnitude decrease of the number of hardware configurations produced by our optimization algorithm for all conflict cases. The main cause of this decrease is the particular features of the MPEG2 algorithm where the SAD, DCT and IDCT hardware configurations can be anticipated out to the frame level rather than macroblock level (see Figure 3). In consequence, due to our optimization algorithm, the hardware configuration is transformed from a major bottleneck in a negligible factor on performance.

In order to conclude this section, four points should be noticed regarding the presented results and optimization. Firstly, the reduction of the number of hardware con-

	Initial	With interprocedural SET optimization				
HW op	[# SETs]	No conflict	DCT Quant conflict	DCT VLC conflict	Quant VLC conflict	DCT Quant VLC conflict
DCT	61440	1	15360	15360	1	15360
Quant	15360	1	15360	1	15360	15360
VLC	15360	1	1	15360	15360	15360

Table 2. The impact of the interprocedural optimization on the number of required hardware configurations in M-JPEG encoder

	Initial	With interprocedural SET optimization				
HW op	[# SETs]	No conflict	SAD DCT conflict	SAD IDCT conflict	DCT IDCT conflict	SAD DCT IDCT conflict
SAD	117084	1	3	3	1	3
DCT	1152	1	3	1	3	3
IDCT	1152	1	1	3	3	3

Table 3. The impact of the interprocedural optimization on the number of required hardware configurations in MPEG2 encoder

figurations depends on the characteristics of the target applications. As previously presented, the impact of our optimizations for MPEG2 encoder is substantial, while for other applications (e.g. M-JPEG) it depends on the possible hardware conflicts between operations. Second, it should be mentioned that this optimization can also increase the number of hardware configurations, e.g. when the considered procedure associated to the hardware operations have multiple call sites and conflicting operations. Flow-sensitive data-flow analysis and profile information can be used to prevent this situation. Nevertheless, taking into account that the hardware configuration can be performed in parallel with the execution of other instructions on the GPP, the reconfiguration latency may be (partially) hidden. The final major point is that a significant reduction of the number of executed hardware configurations is directly reflected in a significant reduction in power consumption, as the FPGA reconfigurations is a main source of power consumption.

6 Conclusions

In this paper, we have proposed an interprocedural optimization algorithm for hardware configuration instructions. This algorithm takes into account specific features of the target applications and of the reconfigurable hardware such as the ”FPGA area placement conflicts”. It allows the anticipation of hardware configuration instructions up to the application's main procedure. The presented results show that our optimization produces a reduction of up to 3 - 5 order of magnitude of the number of executed hardware configuration instructions for the MPEG2 and M-JPEG multimedia benchmarks.

Future research will focus on compiler optimizations to allow for concurrent execution. We also intend to extend the compiler to provide information for an efficient FPGA area allocation of the different hardware operations in order to eliminate the FPGA-area placement conflicts.

References

1. Campi, F., Cappelli, A., Guerrieri, R., Lodi, A., Toma, M., Rosa, A.L., Lavagno, L., Passerone, C.: A reconfigurable processor architecture and software development environment for embedded systems. In: Proceedings of Parallel and Distributed Processing Symposium, Nice, France (2003) 171–178
2. Sima, M., Vassiliadis, S., S.Cotofana, van Eijndhoven, J., Vissers, K.: Field-Programmable Custom Computing Machines - A Taxonomy. In: 12th International Conference on Field Programmable Logic and Applications (FPL). Volume 2438., Montpellier, France, Springer-Verlag Lecture Notes in Computer Science (LNCS) (2002) 79–88
3. Becker, J.: Configurable Systems-on-Chip : Commercial and Academic Approaches. In: Proc. of 9th IEEE Int. Conf. on Electronic Circuits and Systems - ICECS 2002, Dubrovnik, Croatia (2002) 809–812
4. Gokhale, M.B., Stone, J.M.: Napa C: Compiling for a Hybrid RISC/FPGA Architecture. In: Proceedings of FCCM'98, Napa Valley, CA (1998) 126–137
5. Rosa, A.L., Lavagno, L., Passerone, C.: Hardware/Software Design Space Exploration for a Reconfigurable Processor. In: Proc. of DATE 2003, Munich, Germany (2003) 570–575
6. Ye, Z.A., Shenoy, N., Banerjee, P.: A C Compiler for a Processor with a Reconfigurable Functional Unit. In: ACM/SIGDA Symposium on FPGAs, Monterey, California, USA (2000) 95–100
7. Moscu Panainte, E., Bertels, K., Vassiliadis, S.: Dynamic hardware reconfigurations: Performance impact on mpeg2. In: Proceedings of SAMOS. Volume 3133., Samos, Greece, Springer-Verlag Lecture Notes in Computer Science (LNCS) (2004) 284–292
8. Vassiliadis, S., Gaydadjiev, G., Bertels, K., Moscu Panainte, E.: The Molen Programming Paradigm. In: Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation, Samos, Greece (2003) 1–7
9. Vassiliadis, S., Wong, S., Gaydadjiev, G.N., Bertels, K., Kuzmanov, G., Moscu Panainte, E.: The Molen Polymorphic Processor. IEEE Transactions on Computers **53(11)** (2004) 1363–1375
10. Tang, X., Aalsma, M., Jou, R.: A Compiler Directed Approach to Hiding Confguration Latency in Chameleon Processors. In: FPL. Volume 1896., Villach, Austria, Springer-Verlag Lecture Notes in Computer Science (LNCS) (2000) 29–38
11. Mei, B., Vernalde, S., De Man, H., Lauwereins, R.: Design and Optimization of Dynamically Reconfigurable Embedded Systems. In: Proceedings of Engineering of Reconfigurable Systems and Algorithms (ERSA), Las Vegas, Nevada, USA (2001) 78–84
12. Moscu Panainte, E., Bertels, K., Vassiliadis, S.: Instruction Scheduling for Dynamic Hardware Configurations. In: Proceedings of Design, Automation and Test in Europe 2005 (DATE 05), Munich, Germany (2005) 100–105