A DAG-Based Design Approach for Reconfigurable VLIW Processors

Cesare Alippi William Fornaciari Laura Pozzi Mariagiovanna Sami
Dip. di Elettronica e Informazione, Politecnico di Milano, ITALY
{alippi,fornacia,lpozzi,sami}@elet.polimi.it

Abstract

This paper explores the possibility of enabling a partial customisability of the Instruction Set of Very Long Instruction Wold processors for embedded applications, by exploiting Field Programmable Gate Arrays technology. A formal methodology is presented leading to selection of the application critical parts, whose RFUs (Reconfigurable Functional Units) implementation allows the reduction of overall execution time. Experiments performed on representative benchmarks show the applicability of the proposed approach.

1 Introduction and Target Architecture

The typical tradeoff in designing an embedded system is not only related to cost and performance, but also includes an evaluation of the flexibility in terms of architecture and methodology. Present commercial solutions allow the designer to customise a CPU by choosing inside a library of "extended opcodes" and corresponding functional units (the customised CPU is finally totally hardwired) . Possibility of integrating conventional C-MOS circuitry and FPGA sections on the same chip leads us to increase design flexibility, by envisioning a Very Long Instruction World (VLIW) processor whose native Functional Units (FUs) are augmented by application-specific Reconfigurable FUs (RFUs) implemented on the FPGA section.

The native Instruction Set (IS) of the core CPU relies on hardwired FUs, and is extended via application-specific opcodes implemented on the RFUs. RFUs are run-time reconfigurable, but the choice of optimum extended opcodes for the various segments of the application is statically made at compile time.

The novelty of our solution with respect to the previous proposals [1],[2] is that it relates to a general design methodology, rather than a specific application. We identify the analysis framework to set-up a design flow able to select the best candidate functions to be implemented via the RFUs.

A skeleton of the proposed design flow is depicted in Figure 1. A profiling activity identifies the distribution of the computational workload among the different functions composing the application and provides the guidelines for the following architectural analysis that identifies the set of functions to be implemented onto the RFUs. During architectural analysis the following aspects are evaluated:

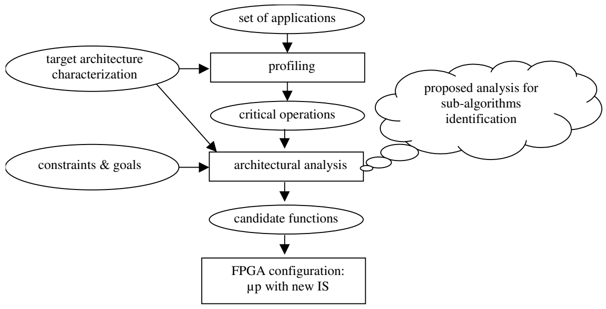

Figure 1. The proposed design flow

- granularity of the candidate functions;
- expected speedup, balanced with reconfiguration overhead;
- analysis of the cross-relations between the compilation strategy for the VLIW core and the modified IS;
- identification and optimisation of the configuration scope, i.e. the part of the application where the same RFU configuration is used.

We focus here on the construction of a theoretical model and identification of the candidate solution to be implemented via RFUs.

The target architecture is a VLIW processor organised in *clusters*, consisting each in a number of functional units and a register file that these FUs share. The IS architecture provides instructions that copy values between register files of different clusters. The RFU represents an additional datapath of the processor executing a specialised operation, which implements a segment of computation extracted from the application algorithm and mapped onto the FPGA. A corresponding opcode, the *fpga-opcode*, is generated and it replaces the relevant segment of computation in the translation from high level code into machine code.

Assuming this target architecture, our problem becomes that of extracting from the application algorithm the segments of computation that are best candidates to be implemented as *fpga-opcodes*.

2 DAG Analysis and MISOs

Identification of *candidate functions* to be mapped onto the RFU is performed by analysing the DAG of the application algorithm or, more properly, the DAGs of the *critical basic blocks* identified in the profiling phase. Our aim is to identify *sub-algorithms* critical for application performances, suitable for RFU mapping. The DAG nodes have two inputs at most (they represent assembler-like operations); fanout greater than one is allowed, i.e. the output of a node can be input to multiple destination nodes.

We consider as candidate functions only *connected* Multiple Input Single Output subgraphs (MISOs); formally, a MISO M^i is a subgraph $< V^i, E^i >$ where, for each node $v_k^i \in V^i$ excepting one, v_O^i, all edges originating from v_k^i end on other nodes belonging to V^i; v_O^i is the output node of M^i.

We further define a MAXMISO MM^i, as a MISO that is not completely included in any other MISO. It is proved (see [3], extended version of this paper) that:

Theorem 1 *Two MAXMISOs MM^i, MM^j cannot partially overlap.*

The above property is the basis for a linear complexity algorithm extracting MAXMISOs from critical blocks in a DAG (described in [3]). Each MAXMISO is then checked for *feasibility*, in terms of number of input variables and complexity of resulting structure (roughly evaluated by high level synthesis of the MAXMISO itself, thus achieving a preliminary area and latency estimation). A search for feasible MISOs can also be performed, *restricted to unfeasible MAXMISOs*; in fact:

Lemma 1 *All MISOs in the DAG are either MAXMISOs or contained in a MAXMISO.*

Reconfiguration overhead excludes the possibility of FPGA mapping for more than one MAXMISO within a single basic block; on the other hand, we choose to have only one *fpga-opcode* active at any given time, to simplify instruction decoding (semantics of the *fpga-opcode* change with each reconfiguration).

3 Experimental Results

Experimentation consisted in compiling C code by SUIF [4] to generate an assembler-like intermediate representation, extracting critical basic blocks by means of a profiler, and generating the MAXMISOs. The following table summarises some features of the 3 main MAXMISOs found in a DES cryptography algorithm. For results on more applications refer to [3].

	#N	#I	#occ
MM 1	7	6	16
MM 2	12	7	16
MM 3	11	5	32

We characterise each MM with its number of nodes (#N), number of inputs (#I) and number of distinct occurrences in the DAG (# occ). It is interesting to note the significant occurrence of MAXMISOs (up to 32 times in the same critical block) and the fact that the number of inputs is always reasonably small; this makes the MMs good candidates for RFU implementation.

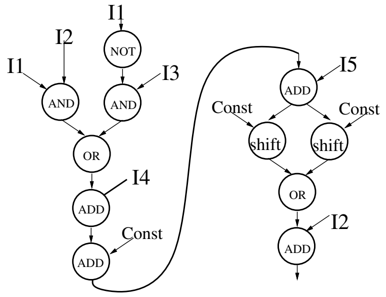

Figure 2. MM 3 for the cryptography algorithm

MAXMISO 3 is thus particularly representative, its DAG is shown in Figure 2. Implementing it by a single specialised functional unit -optimised on the basis of high-level synthesis techniques- allows to sensibly reduce the number of cycles required by its execution; in fact, intermediate results are then simply stored in latches (or not even stored, when chaining is possible) and do not require access to the register file. The modified DAG in which each occurrence of MAXMISO 3 is collapsed into one *fpga_node* is then used for final scheduling; note that the collapsing never decreases available parallelism.

4 Conclusions

Augmenting a processor with one or more Reconfigurable (and therefore customisable on the fly) Functional Units results in the possibility to better tailor resources *to each* different application served. A methodology for extraction, from a DAG, of segments of code to be executed on the RFU has been presented and experiments of its application are shown.

References

[1] R Razdan and M. D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. *Proc. MICRO-27*, pages 172–180, 1994.

[2] S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao. The Chimera Reconfigurable Functional Unit. *IEEE Proc. FCCM*, 1997.

[3] C.Alippi, W.Fornaciari, L.Pozzi, and M.G.Sami. A DAG-Based Design Approach for Reconfigurable VLIW Processors. Technical Report 98-104, Politecnico di Milano.

[4] M.W.Hall et al. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, Dec 1996.