Graph Theory and Application Specific Processors

Carlo Galuzzi   Koen Bertels   Stamatis Vassiliadis

Computer Engineering, EEMCS
Delft University of Technology

{carlo, k.l.m.bertels, s.vassiliadis}@ewi.tudelft.nl

Abstract

In this paper, we discuss the issues involved in designing ASIPs and more specifically when combined with GPP's. Such a design process involves several steps and here we focus on one of them: the program transformation phase, the second step of the design process that is required when certain parts of the application are extracted and will be executed on a reconfigurable component. Over the last years, many algorithms have been developed and proposed. In this paper we provide an overview of existing program transformation approaches and for each of these, we present a critical evaluation, underlining the differentiating features as well as the strengths and weaknesses. We discuss to what extent they can be used for reconfigurable hardware and propose potential improvements.

Keywords

Graph theory, ASIP, Reconfigurable computing, Instruction-set extension, Program-transformation

I. Introduction

Electronic devices are very common in everyday life. It's enough to think about mobile phones, digital cameras, electronic protection systems in the cars, etc. This great variety of devices can be implemented using different approaches and technologies. Usually these functionalities are implemented using either General Purpose Processors (GPPs) or Application Specific Integrated Circuits (ASICs) or Application Specific Instruction-Set Processors (ASIPs). GPPs can be used in many different applications and contrast with ASICs that are processors designed for a specific application such as the processor in a TV set top box.

The main difference between GPPs and ASICs is in terms of flexibility. The programmability of GPPs supports a broad range of possible applications but leads to more power use of due to the inefficient units consumption. On the other side, ASICs are able to satisfy specific constraints such as size, performance and power consumption using an optimal architecture for the application, but designing and manufacturing an ASIC today presents some problems [9]. Due to shrinking geometries, this design complexity grows exponentially and the high mask and testing costs constitute a significant part of the manufacturing cost.

The last years have shown an increased popularity of processors with a customizable architecture, also known as Application Specific Instruction-Set Processors (ASIPs). ASIPs are situated in between GPPs and ASICs: they have a *partially customizable Instruction Set* and perform only a limited number of tasks. Table I summarizes the main differences between ASIPs and the other two approaches.

These processors offer the possibility to extend their instruction set for a specific application, introducing customized functional units for a given domain. Customization allows increased performance reducing power consumption by not having unnecessary functional units. In that way, they combine the flexibility of software with the performance of hardware.

Consequently, designers are shifting toward software solutions even at some loss of design quality measured by area, delay, power. Moreover it is clear that writing and debugging software is cheaper than designing, debugging and manufacturing a complete processor. The programmability of ASIPs therefore avoids the development of a new complete processor and it allows to concentrate the design efforts exclusively on the special datapath.

What the designers want to achieve is to design an instruction set which minimizes some metric (typically run time, memory size, etc.). The starting point of the design process of this processor is application code written in a high-level language such as C. The first step consists of profiling the application software to find the computation intensive segments of the code. The original application is then transformed to incorporate the new instructions and to guarantee the equivalence of the modified program after which it

	Power consumption	Programmability	Strengths	Weakness
GPP	High	Complete	- Flexibility	- Power - Efficient use of functional components
ASIP	Medium	Partially customisable IS	- Customization - Reduced power consumption	- Not yet a mature discipline
ASIC	Low	None	- Power consumption - Possibility to satisfy specific constraints - Performance	- Design complexity due to shrinking geometry - Manufacturing costs

TABLE I

Main differences between GPPs, ASIPs and ASICs.

is compiled to the heterogeneous platform. What we want to examine is the program transformation phase that is required when certain parts of the application are extracted and will be executed on the reconfigurable component. This phase consists of two parts: a part which selects pieces of code to extract and a node collapsing part which collapses the piece of code in a single node.

Over the years, many algorithms have been developed and proposed. Roughly the problem consists in detecting clusters of operations which, when implemented as a single complex instruction, maximize some metric. Such clusters have to satisfy some constraints such as the number of inputs and the number of outputs. A large number of these approaches have a graph theoretical approach. In the remainder of the paper we present a critical evaluation of a number of these approaches. The next section is dedicated to the features of the existing approaches. Section III provides a critical discussion. We conclude with section IV.

II. EXISTING APPROACHES

We can distinguish between two categories addressing the problem of Instruction-Set extension: a first approach consists of finding frequently used patterns in the dataflow graph of the application code. The patterns studied are rather small and they allow multiple input and output (e.g. [6], [7], [3], [11]).

In [6], the authors want to add special single- and multiple-cycle instructions to a small set of primitive instructions. The exploration of the design space is limited by the complexity of the special instructions. They transform the problem of instruction generation in a modified subset-sum problem to generate an optimal Instruction-Set including multi-cycle complex instructions as well as single-cycle complex instructions. In other words, the problem becomes: given a set S which represents the gains of the instructions, find a subset of S whose sum is limited and such that the generated instructions corresponding to the selected gains should be used as much as possible in the application and the number of different instructions corresponding to the gains should be as small as possible. To solve this type of problem, the authors exploit the subset-sum problem solver ([14]) to synthesize application specific instructions. A similar approach is described in [7] although it is limited to single-cycle complex instructions.

The second category deals with growing clusters having single output (e.g. [5]) or multiple outputs (e.g. [10], [8], [1], [15]) until some constraint is violated. Other approaches (e.g. [2]) combine the automatic identification of IS extension and symbolic algebraic manipulation.

An other method to address the problem could be found in dynamic ISA augmentation ([12]). Here the authors determine at runtime which instructions could be collapsed into a single instruction eliminating interlock. They are limited in the number of input and output and function to perform the problem. Moreover they consider critical paths of few machine cycles and they are limited in the scope of instruction text which they consider for dynamic ISA augmentation.

In order to illustrate the respective approaches, we discuss one of each in more detail.

The first approach we examine is related to a general design methodology and not to a specific application. In [5], the authors explore the possibility of enabling a partial customizability of the VLIW

processors for embedded applications, by exploiting FPGA technology.

The target architecture is a VLIW processor with Functional Units (FUs) grouped in clusters and each of these contain also the register file that the FUs share. The IS architecture provides instructions that copy values between register files of different clusters. Some FUs are application-specific reconfigurable functional units (RFUs) implemented through FPGA technology. Furthermore they consider a partially customizable CPU including a certain set of FUs, capable of a native IS that is customized via the reprogrammable FUs.

They present a methodology to select the critical parts of an application that are best suited to be implemented on the RFUs so that it is possible to reduce the overall execution time. A RFU represents an additional datapath of the processor which executes a specialized operation implementing the critical part extracted from the application algorithm and mapped onto the FPGA.

The critical parts are identified by analyzing the connected Direct Acyclic Graph (DAG) created of the critical basic blocks identified in the profiling phase of the application algorithm. The nodes represent primitive operations and can have at most two inputs (they represent assembler-like instructions) and one output which can be used as input to multiple destination nodes. The edges represent data dependencies.

A critical part of a DAG is a subgraph and theoretically the speedup obtained by its special implementation on FPGA is directly proportional to its size. For this reason, the algorithm is trying to identify the largest possible subgraph. To simplify synchronization between RFU and CPU, the RFUs are designed with some restrictions: they cannot have direct access to the data cache and to simplify register file structure and interaction with the other functional units, only multiple read and single write operations are allowed.

This results in graphs having an unlimited number of inputs and a single output, called MISO (Multiple Input Single Output).

Let $G^i = (V^i, E^i)$ a subgraph where V^i represents the set of nodes and E^i represents the set of edges departing from such nodes. If for each node $v_k^i \in V^i$, except the output node v_o^i, all edges originating from v_k^i end on an other node belonging to V^i then G^i is called MISO (note that incoming edges in V^i could be originated from nodes not belonging to V^i). A MISO not completely included in any other MISO is called maximal MISO and is indicated as MaxMISO so max-

imality refers to the impossibility of including other nodes which can lead to violate the output constraint. The MaxMISO therefore satisfies two properties:

- Two MaxMISOs cannot partially overlap.
- All MISOs in the DAG are either MaxMISOs or contained in a MaxMISO.

The algorithm analyzes the DAG looking for MaxMISO: it starts from a node and recursively tries to include its parents until a non legal node is encountered. A non legal node is defined as a node whose inclusion violates constraints such as violating the output constraint of a single output. When a MaxMISO is found, its nodes are removed from the set of nodes under analysis because of the first property given that two MaxMISOs cannot partially overlap. This implies another aspect related to the complexity of the algorithm: each node is visited only once and so the complexity is linear with the number of nodes.

The set of MaxMISOs is evaluated under different aspects and isomorphic MaxMISOs are tagged. We remember that two graphs are isomorphic if there is a bijection between the edges of the two graphs. If there are physical limitations on the number of possible implementations, the possible candidates are weighted in terms of occurrences in the DAG and in terms of the potential speedup obtained.

An other approach in the same direction is described in [8]. The goal is similar: to present an algorithm that automatically extracts operation from the application code to implement on customized functional units.

Let $S \subseteq G$ be a subgraph of a graph G. If every path from $u \in S$ to $v \in S$ involves only nodes belonging to S then S is called *convex*. A *cut* S of a graph G is a subgraph of G. Let $M(S)$ be the function which represents the merit of the cut S i.e. the estimation of the speedup achievable by implementing S as a special instruction.

The problem that the algorithm proposed by the authors tries to solve is the following: given the graphs G_i, the DAGs representing the dataflow of the basic blocks, find N_{instr} cuts S_j which maximize $\sum_j M(S_j)$ under three constraints: number of inputs of $S_j \leq N_{in}$, number of outputs of $S_j \leq N_{out}$ and S_j convex; where N_{in} and N_{out} are the number of register-file read and write ports which can be used by the special instruction S_j. The property of convexity is a theoretical constraint which is needed to ensure the existence of a feasible scheduling which respects the dependencies of S.
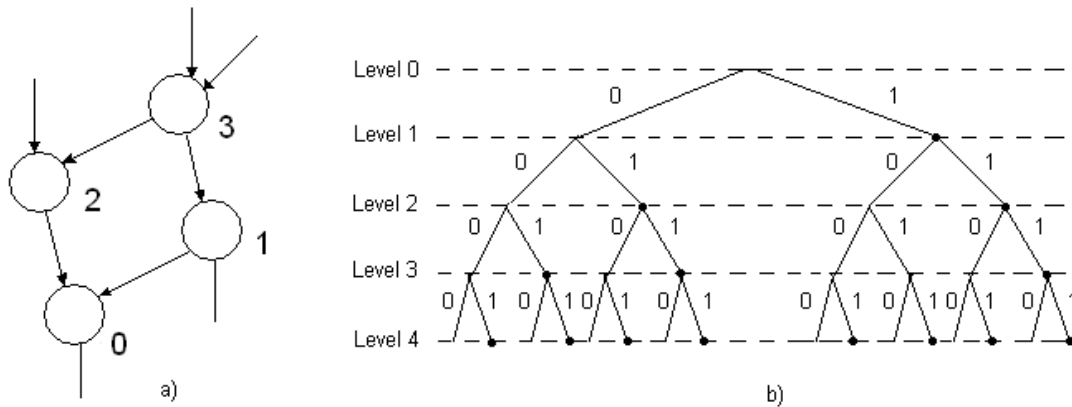
Fig. 1

A subgraph (a) and its search tree (b)

The authors propose two algorithms, both of them based on an algorithm for single cut identification whose extension can be used to find an optimal and a near-optimal set of nonoverlapping cuts in several basic blocks. The two algorithms are called *Optimal* and *Iterative*.

The starting point in both cases is a topological sort on nodes of G: if (u, v) is an edge of G then u appears after v in the ordering. The algorithm uses a recursive search function based on this ordering to explore an abstract search tree. The search tree is a binary tree of nodes representing possible cuts. Figure 1 shows an example of subgraph (a) and its search tree (b). There are 4 nodes in the subgraph and so there 2^4 possible cuts represented by the nodes of the tree. Starting from level 0, the root of the tree, which represents the empty cut, it is possible to include (1-branch) or not (0-branch) the first node. Once the node is included or not in the cut, we can include or not the second node. Iteratively at every level i it is possible to include the node having topological order i to the cut represented by its parent node. The nodes of the search tree that follow a 0-branch represent the same cut as their parent node, and can be ignored in the search. Taking constraints into consideration (number of inputs, outputs and convexity), it is possible to prune the design space: if the inclusion of one node violates some constraint, the inclusion of a later node in the topological order cannot overcome the constraint violation and so it is unnecessary to search other nodes. In the subgraph of Figure 1, we said that there are 2^4 possible cuts but convexity erases some cuts. For example a cut S including only node 0, 1 and 3 is not possible

because node 2 is not included thereby violating the convexity constraint because there is a path from 3 to 0 that includes a node (2) not belonging to S.

Although the complexity of the algorithm is proportional to the number of subgraphs, making it exponential, the search space is tangibly reduced taking constraints into account. This algorithm allows to find the optimal single cut in a single basic block.

The *Optimal* variant identifies multiple cuts from a single graph. Let M be the number of cuts to identify in a single basic block. If $M = 1$ we have the single-cut identification algorithm; in this case, at every level of the search tree there are two branches ($2 = M + 1$). For $M > 1$ it's enough to build a similar tree where each node branches $M + 1$ times instead of 2. In this way nodes of the search tree represent M cuts: n-branches at level i make the node with index i in the n-th cut be included.

Iterative as the name says, is an iterative application of the single-cut identification algorithm on the same basic block. Every time a cut is identified this is collapsed in a single node and excluded from forthcoming identification steps.

In [2], the graph based approach is combined with symbolic algebra. The issue is to present a solution to the problem of clustering operations in the code to implement as new complex instructions on new functional units of an extensible ASIP processor. This problem is addressed in two steps. A first step which identifies possible candidates and a second step of algebraic manipulations to map dataflow sections of the code to complex instructions available on the proces-

sor.

To extract possible candidates, the authors implement two different algorithms: the MaxMISOs extractor ([5]) and the Optimal algorithm ([1]) analyzed before. Both of them are used to select MISO, although Optimal is able to find also multiple-output subgraphs. Once potential instructions are extracted, they are synthesized in order to estimate their cost and execution time.

After which starts the second step which exploits polynomial properties. It requires two different sets of inputs: a set of polynomials representing the basic blocks of the application and a set of polynomials representing the complex dataflow instructions.

The goal of this step is to decompose the polynomial representation of the basic blocks exploiting the minimum number of available instructions. Basic blocks extracted from the code could be of two types: block that calculate a polynomial function or blocks which perform a series of bit manipulation or Boolean function. In the first case we have directly the polynomial representation of the basic block otherwise using interpolation-based algorithm ([13]) is possible to obtain the equivalent polynomial representation.

To manipulate the polynomial expressions it is possible to use specific algebra software packages to solve polynomial optimization such variable elimination in a set of polynomials or other problems involving particular branches of geometry ([4]).

To determine the minimum number of operations necessary to represent the basic blocks extracted, they propose an algorithm of decomposition based on a polynomial representation of available instructions. If S is a basic block and L is the set of polynomials corresponding to the instruction set, the algorithm tries to simplify S. If the simplification is identical to a polynomial of L a possible solution is found and the corresponding tree node is marked accordingly otherwise the same step is applied recursively. The goal is to use the minimum number of operations. The algorithm prunes the design space in this way: a bounding function represented by the number of instructions is used to calculate the basic block. In other words if a solution is found which uses only two instructions it doesn't look for solutions which use more than two instructions. Furthermore all the solutions with two instructions are uncovered and the best one (in terms of previous analysis) is chosen.

After the Instruction Set has been chosen, the original software code is automatically transformed to use the new instructions assisted by symbolic polynomial manipulation algorithms. The output is optimized C code with intrinsic function calls automatically inserted.

III. Comparative evaluation

In this section we analyze the strengths and weaknesses of the approaches presented in the previous section and Table II summarizes the main differences. The algorithms presented show various differences and now we underline the mains, which could be represented by: complexity, constraints which the clusters have to satisfy, the connectivity or not of the subgraphs found and if there are limitations on the size of the graphs that the algorithms analyze.

The strength of **MaxMISOs** is given by its complexity. In a graph G with n nodes there are 2^n subgraphs. Theoretically the exploration of the whole design space looking for subgraphs therefore presents exponential complexity but in this case, the algorithm exploits the maximality of the subgraphs which cannot overlap and when a graph is found its nodes are removed from the node to analyze. In this way the complexity is linear with the number of nodes and not exponential so that the algorithm represents a good tradeoff between complexity of exploration and effectiveness of the resulting extracted instructions.

There aren't limitations on the number of input of the clusters but an interesting feature of this algorithm is that the number of inputs of the graphs identified is always reasonably small. The experimental results of the authors show that the MaxMISOs extracted from various application programs have in only few cases a number of input that exceeds ten.

As far as the weaknesses are concerned, the approach is not capable of identifying disconnected graphs thereby eliminating the possibility of exploiting their parallelism by implementing the graphs as part of the same instruction. Moreover the maximal size doesn't represent optimality under constraints on the number of inputs given a particular MISO and so MaxMISO's definition doesn't take the number of input into account.

As far as the second approach is concerned (**Optimal** and **Iterative**), the efficient design exploration is a strong point. However, in the worst case, the approach is confronted with an exponential complexity. The authors claim however that in all practical cases they used for testing, the observed complexity is within polynomial bounds.

Algorithm	Complexity	Number of inputs	Number of outputs	Disconnected graphs	Size
MaxMISO	Linear complexity	Multiple	Single	No	Not underlined
Optimal Iterative	Exponential tendency	Multiple	Multiple	Yes	Not underlined

TABLE II
Comparison of the algorithms.

The algorithm doesn't have constraints on the number of inputs nor on the number of outputs but the tighter the constraints are, the faster the algorithm is. Moreover these algorithms are able to identify also disconnected graphs.

Experimental results show that for a low number of inputs and outputs, the algorithm performs comparable to the approach described in [5]. Once the number of inputs and outputs is increased substantially, its performance is substantial better than [5].

The approach presented in [2] is an extension of the algorithms described before. The main characteristic is the use of symbolic algebra to support extraction techniques that aim to automate the instruction set selection. In this sense, it is difficult to compare against the others. Its main strength is its performance. Once instructions are selected and optimized, they are implemented on added functional units which are pipelined or executed in one cycle. Experimental results have shown an average improvement of 41% at the expense of 9.2% increased area.

IV. Conclusions

In this paper we presented some approaches that address the problem of Instruction-Set extensions. Although there are different methods to deal with this problem, we have analyzed in more or less details two approaches. The main approach consists in detecting clusters of operations to collapse on one special instruction. All of them exploit properties of graphs and we analyzed the main differences.

We found that all of the algorithms have a series of limitations in term of number of inputs and outputs. Nor is it possible to exploit parallelism as suggested by the presence of disconnected subgraphs. A final limitation is that the algorithms have exponential complexity.

Future work will focus on the use of similar techniques in the context of the automatic extraction of functions that will be executed on reconfigurable processors in combination with a general purpose processor.

References

[1] M. Vuletić L. Pozzi A. K. Verma, K. Atasu and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. *In Proceedings of the 1st Workshop on Application Specific Processors*, November 2002.

[2] P. Ienne A. Peymandoust, L. Pozzi and G. De Micheli. Automatic instruction set extension and utilization for embedded processors. *In Proceedings of the 14th International Conference on Application-specific Systems, Architectures and Processors*, June 2003.

[3] M. Arnold and H. Corporaal. Design domain specific processors. *In Proc. of the 9th Intl. Workshop pn Hardware/Software Codesign*, pages 778–79, Apr. 2001.

[4] T. Becker and V. Weispfenning. *Gröbner Bases*. Springer-Verlag, new York, 1993.

[5] L. Pozzi M. Sami C. Alippi, W. Fornaciari. A DAG-based design approach for reconfigurable VLIW processors. *In Proc. of the Design, Automation and Test in Europe Conf. and Exhibition*, pages 778–79, Mar. 1999.

[6] C. W. Yoon I. C. Park S. H. Hwang H. Choi, J. S. Kim and C. M. Kyung. Synthesis of application specific instructions for embedded DSP software. *IEEE Transactions on Computers*, 48(6):603–14, June 1999.

[7] I. J. Huang and A. M. Despain. Synthesis of instruction-set for pipelined microprocessors. 31st *Design Automation Conference*, pages 5–11, 1994.

[8] L. Pozzi K. Atasu and P. Ienne. Automatic application-specific instruction-set extensions under microachitectural constraints. *In Proceedings of the 40th Design Automation Conference, Anaheim, Calif.*, June 2003.

[9] S. Malik K. Keutzer and A. R. Newton. From ASIC to ASIP: The next design discontinuity. *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference*, pages 84–90, 16-18 Sept. 2002.

[10] Y. Jiang Y. Pate R. K. Brayton M. Baleani, F. Gennari and A. Sangiovanni-Vincentelli. HW/SW partitioning and code generation of embedded control application on a reconfigurable architecture platform. *In Proc. of the 10th Intl.*

Workshop on Hardware/Software Codesign, pages 151–56, May 2002.

[11] S. Ogrenci Memik R. Kastner, A. Kaplan and E. Bozorgzadeh. Instruction generation for hybrid reconfigurable system. *ACM Transactions on Design Automation of Embedded Systems (TODAES)*, 7(4), Oct. 2002.

[12] J. Phillips S. Vassiliadis and B. Blaner. Interlock collapsing ALUs. *IEEE Transactions on Computers*, 42(7):825–839, July 1993.

[13] J. Smith and G. De Micheli. Polynomial circuit models for component matching in high-level synthesis. *IEEE Trans. on VLSI*, 9(6):783–800, Dec. 2001.

[14] C. E. Leiserson T. H. Cormen and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, McGraw-Hill Book Company, 1992.

[15] S. Hauck Z. A. Ye, A. Moshovos and P. Banerjee. CHI-MAERA: A high-performance architecture witha tightly-coupled reconfigurable functional unit. *In Proc. of the 27th Annual Intl. Symp. on Computer Architecture*, pages 225–35, June 2000.