Graph Covering for generating instruction specific application instructions: an overview of some existing methods

Carlo Galuzzi Koen Bertels Stamatis Vassiliadis
Computer Engineering, EEMCS
Delft University of Technology
{carlo, k.l.m.bertels, s.vassiliadis}@ewi.tudelft.nl

Abstract

The execution time of an application can be considerably reduced by implementing parts of the application in hardware instead of software. Graph Theory can be used for selecting which parts of the application are suitable for an hardware implementation. To this purpose, the application is represented as a directed graph, called a subject graph, and the selection problem can then be described as the search and selection of subgraphs having particular properties.

The subgraph identification problem requires the evaluation of two separate issues: a coverage problem of the subject graph and a selection problem of the subgraphs. The complexity of the problems involved has led researchers to provide both exact solutions as well as heuristic solutions representing a trade-off between the goodness of the solution provided and the resources used to obtain it. Although a heuristic solution is usually used to limit the search space, how much is left uncovered is what influences the goodness of the solution. This depends on the parameters that are taken in consideration to evaluate the solution as area, delay, etc.

In this paper we provide an overview of the main issues related to the coverage of the search space pointing out the goodness of the solutions provided as well as the main features of the method applied to obtain it.

Keywords

Covering problem, Subgraph identification, Graph Theory.

I. Introduction

Our life is more and more surrounded by electronic devices that everyday become smaller and increase the number of tasks they can perform. Although in the past years ASICs and GPPs were the main ways adopted to implement the functionalities of these devices, over the last years their use has seen a progressive shift toward the use of reconfigurable architectures ([27], [29], [20]) which are gaining ground day by day and are progressively replacing them in many cases. These architectures have the property to be reconfigurable totally or partially and this allows them to be used not only for the original task but for new tasks as well, avoiding the design-process of a complete new device and reducing costs.

An example of reconfigurable architecture can be realized combining a GPP with a reconfigurable part as an FPGA ([1]). When a program is executed on this architecture, a certain number of instructions are hardwired whereas the rest of the operations are implemented by software. The reconfigurable part can then be used to hardwire some instructions of those implemented by software. It doesn't exist basic procedure to share out the instructions among hardware and software; what exists it's a huge number of different types of approaches guided by different prospects and constraints. Although the final goals are often different and constitute a huge variety, all the approaches follow a common guideline. Goal of this paper is to present this guideline showing how this selection can be done and which are the metrics involved in this choice.

The shift of one instruction from software to hardware can reduce some metrics like the execution time, and then a right balance between hardware and software can improve and maximize performance. This balance, often addressed as hardware/software partitioning problem or hardware/software codesign problem ([6], [25]), it is an interesting problem whose solution is frequently branched out into many different fields ranging from graph theory to selection problems. The complexity of the problem, how we see NP ([28]), has an influence on the solution which most of the time is strictly related to the formulation of the problem in a particular way, with certain conditions, and then not always suitable for a general method to adopt to solve the problem.

Goal of this paper is to provide a general view of the problem which doesn't focus on too specific cases or subproblems, includable among general cases, but favors the view of the problem in its completeness. Although the problem has been set out as unique it can be divide in two subproblems: generation and selection. In the following Section II we present the first part of the problem: how to identify instructions suitable for an hardware implementation whereas in Section III we analyze the problem of choice of an optimal subset of instructions together with an overview of the metrics used as parameters of choice. The paper concludes with Section IV with some considerations on future directions.

II. First step: generation.

In the previous section we introduced the main idea: given a program to implement on one architecture, find inside the code sequences of operations for which a shift from a software implementation to an hardware one can improve performance. The process starts with a high level code, like C, that specifies the application and manually or automatically parts of the code are selected to be tailored as new capabilities improving performance. Although a manual creation of new instructions can be supported by the human ingenuity creating high quality results, performance and time-to-market requirements have led to the search of an automatic design flow ([10], [5], [17]). The main steps, which will be subjects of a subsequently analysis, are the following:

- ▶ the choice of an helpful representation of the problem;
- ▶ the coverage of the design space;
- ▶ the generation of a set P of possible new instructions;
- ▶ the selection of an optimal subset $P_{Opt} \subseteq P$.

It's helpful to note that although bigger is P, harder is the selection of $P_{Opt} \subseteq P$, a big size of P in terms of possible new instructions can be useful when the constraints are changed, shrunk or relaxed allowing different choices of the subset P_{Opt} satisfying the new constraints, then benefitting the reconfigurability of the system. The first three steps are analyzed in this section whereas the last one will be the subject of the next section.

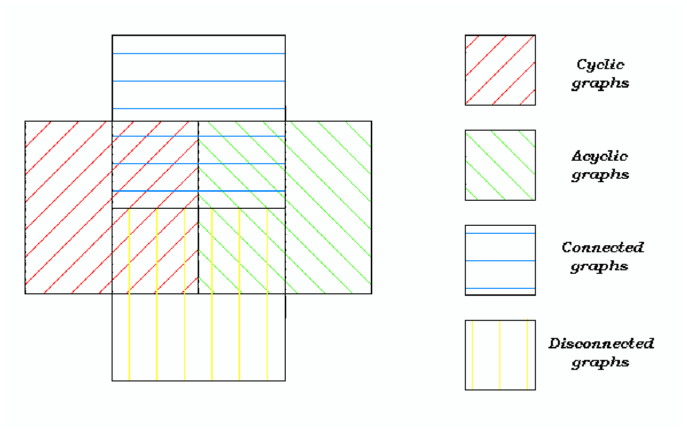

Fig. 1
Families of graphs and their intersection

A. The choice of a representation.

Although the problem is independent from its representation, the choice of a representation rather than an another can benefit and simplify the problem. A helpful way can be a graph representation of the problem. The code of the application is analyzed and a directed graph, usually called the **subject graph**, is built: the nodes represent the operations and the edges represent data dependencies. The identification of a sequence of operations inside the code then translates into the recognition of a subgraph inside the subject graph.

The shape of the graph, in its own way, can be seen as a watershed: the strategy of analysis is usually shape-dependent and then a method suitable for a graph with peculiar properties could be not appropriate in presence of different ones. This assertion find evidence for example in the huge number of existing methods for tree-shaped graphs not extendable to more general graphs including, for example, cycles and loops.

Once the program is represented with a graph, its nodes have to be analyzed. The properties of the graph have an important influence on the type of study of the problem. These types can be subdivided in four families: connected, disconnected, cyclic and acyclic graphs. Clearly these families have intersections that are not empty (Fig. 1) and usually connected and acyclic graphs represent the starting point. The next step is then the gradual inclusion in the study of disconnected graphs and of cycles and loops. The study of the problem in presence of disconnected graphs allows to exploit the parallelism provided considering each connected component at the same time ([5], [7]) but in many cases the authors have addressed only the study of connected graphs ([4], [6], [11], [12], [26]) reducing the study of a disconnected one to the study of the single connected components.

Although the problem to manage a disconnected graph, in a certain way, can be skipped with the study of the single components, cyclic graphs represent a real obstruction. The management of cycles and loops within a graph is an important and no easy task addressed only by very few authors who has often implemented the entire cycles or loops as custom instructions. The majority has focused only on *Directed Acyclic Graph* also known as **DAG** ([21], [24], [15]). A DAG representation of the problem allows to define a topological order (or a reverse topological order) of the nodes ([28]) whereas a different graph joins the trouble to define a one-to-one order of the nodes to the complexity of the problem.

It's worthy of note to observe that a graph representation helps not only because it's an easy way to visualize the problem but also because there exist many results in graph theory which can find a useful application to this problem.

B. The coverage of the design space.

Once the graph is sorted in a one-to-one way, the analysis of the nodes has to start. During the analysis we identify subgraphs which in the general case are Multiple Input Multiple Output graphs (MIMOs) ([6], [5], [4]), family of graphs that clearly contains the subfamily of Multiple Input Single Output graphs (MISOs) ([3]). We identify these two types of graphs for a specific reason: the sequence of instructions to move from hardware to software can be seen as a multivalued function: given $n \geq 1$ inputs the function produces $m \geq 1$ outputs:

$$(Out_1, ..., Out_m) = f(In_1, ..., In_n) \tag{1}$$

which can be written in a short way as $\underline{Out} = f(\underline{In})$ that becomes $Out = f(\underline{In})$ in the case of MISOs ($m = 1$). Up till now the methods favor MIMOs with a limited number of inputs and outputs ([6], [4], [19], [29], [12]) and MISOs ([3], [12] with limitations on the number of inputs).

The subgraph search aims at the identification of graphs with peculiar properties to satisfy; this can be done in two ways: knowing in advance operations suitable for an hardware shift (because they fill properly an available area, or minimize the delay, or are low power consumption, etc.) or not and then the design exploration has to identify these operations following particular criteria. A set of program statements (or nodes in the context of graphs) that is a candidate for implementation as a custom instruction, i.e. the criteria are satisfied, it is called **template**. In the first case then we speak about **template identification** ([9], [19]) and in the second one about **template generation** ([3], [27], [6]). An example of template can be for example a sequence of operations as an addition followed by a multiplication or something more complicated involving many different instructions.

The use or not of templates makes a big difference. While in the first case the design exploration is reduced to a problem of graph isomorphism ([8], [16]), the second one joins to that the problem to generate templates. The generation of templates, as the problem of selection, is based on the use of a function which in this context is called **guide function** whereas for in the context of selection it is called **cost function**. These two functions are strictly related and both are used to help the search; this means that a solution of the problem can benefit from a right division of the constraints that the solution has to satisfy, between the parameters taken into account by the two functions. Example of parameters for the guide functions can be the following:

- *number of inputs/outputs*;
- *number of operations*;
- *properties of the graph.*

The maximum number of inputs and outputs are introduced to ensure the feasibility of the instruction/subgraph identified when there are physical limitations on the number of inputs and/or outputs of the new operations to generate; a limitation on the number of operations on the contrary can be related to the available area for the new complex instructions. About the properties of the graph, these can range from the connectedness of the subgraph to the identification of more specific properties as the convexity ([5], [30]). We note that whereas the connectedness is a property dividing the graphs in two disjoint sets: connected or disconnected graphs, convexity is a properties inherent only to connected graphs. In few words a graph is convex if there exists no path from a node of the graph to an other node which involves a node not belonging to the graph. This property ensures a feasible scheduling of the new instructions which respects the dependencies and whereas a MIMO can satisfy or not this property, this is certainly satisfied by a MISO.

During the template generation, the guide function takes the parameters and the nodes of the graph as input and generate as output a set of functions suitable for a further analysis mostly part of the selection step which will yield an optimal subset in terms of performance. In the case of a graph with topological order of the nodes, their analysis can be done with a depth-first search or a breadth-first search for example ([28]).

The **coverage** of the design space, i.e. the analysis of the nodes to generate templates or to identify isomorphic templates within the graph, can be pursued exactly or heuristically depending on the type of solution is searched. As for a set of n elements there exist 2^n subsets ranging from the empty set $\emptyset$ to the entire set, for a graph with n nodes there exist 2^n subgraphs. The complexity of an exhaustive search of all nodes becomes in this way exponential and an exponential complexity of the problem turns into an exponential time to solve it.

The exponential complexity of the problems has led the authors to provide solutions which most of the time represent tradeoff between the goodness of the solution and the resources used to provide that. These solutions can then be provided with a complete or partial analysis of the design space. In the first case we speak about exact solutions and in the second one about heuristic solutions. While in the first case all the design space is analyzed, a heuristic solution is provided with a rule of thumb, simplification or educated guess that reduces or limits the search for solutions. The main problem with heuristic is that they don't guarantee optimal, or even feasible, solutions and are often used with no theoretical guarantee.

One way to solve exactly covering problem is by using a **branch-and-bound** approach ([13]). This approach starts with a search space potentially exponential in size, and reduce step by step the search space. The essence of this approach is that in the total enumeration, at any node, if it's possible to show that the optimal solution cannot occur in any of its descendants then there is no need to consider those descendant nodes. Then the search can be pruned at that node and more is pruned in the search space more is the possibility to reduce the problem to a computationally manageable size. Other covering approaches use dynamic programming which is a way of decomposing certain problems hard to solve into equivalent formats that are more amenable to solution. Basically a dynamic programming approach solves a multi-variable problem by solving a series of single variable problems. A drawback of dynamic programming is that it can only operate on tree-shaped subject graph and patterns, excluding directed graph with cycles. Thus the non-tree-shaped graph has to be decomposed into sets of disjoint trees. Another approach [4], is based on dynamic programming, although without the requirement that the subject graph and the patterns are trees.

Once the exact or the heuristic analysis of the design space is finished, the result of this analysis is represented by a set of subgraphs representing possible complex instructions to implement in hardware. Belong to this set all the subgraphs satisfying the parameters of the guide function; these include also isomorphic graphs which has to be identified to reduce the size of the set. The cutback of this set to a minimal set of elements is important to reduce the complexity of the selection analysis as well: less elements to process and then to be selected means less time needed to find a solution. The isomorphism problem which arises during the reduction of the set, is a well known NP-complete problem, then the size of the set is an important parameter to take into account ([16]). Once this set is reduced to a minimal set P the selection is ready to start.

III. Step 2: Selection

Goal of this step is: given a minimal set P select a subset $P_{Opt} \subseteq P$ which maximizes performance. More formally the problem can be set out as follow: given a set A of n objects $a_1, ..., a_n$ and m subsets $A_1, ..., A_m$ of objects of A with a cost associated to each subset A_i, select a family of subsets A_i with minimum cost, in such a way that it is possible to cover the elements $a_1, ..., a_n$. Moreover the subsets A_i selected has to be pairwise disjoint, i.e. each element has to belong to exactly one subset selected yielding in this way a partition of the elements $a_1, ..., a_n$ ([13], [14])

This selection is done with the aid of the cost function. There are many metrics that can be considered by the cost function to select the elements of P_{Opt}; the mains are the following [18]:

- *execution time*;
- *cycle count*;
- *static code size*;
- *dynamic code size*;
- *compile time*;
- *hardware design and manufacturing cost*;

- *delay*;
- *area*;
- *power consumption*;

The execution time and the cycle count are usually considered alternatively: not in all cases is it possible to obtain an accurate estimate of the system's cycle time and then the cycle count is taken in place of it. The static code size is a parameter introduced when the memory is limited for example by power or volume constraints. Although this is not always a parameter taken into account, a reduction in code size allows a program to fit in a limited amount of memory. On the other side dynamic code size is a parameter which measures the number of instructions fetched during the execution of the program and then if reduced can improve the execution speed. The compile time is proportional to the complexity of the instruction set: the harder the compilation, the greater the compilation time.

Together with these parameters, we can find the hardware design and manufacturing cost: the generation of particular new instructions has to include the non-recurring engineering costs as the one-time charge for photomask development, test, prototype tooling, associated engineering costs, etc. Lastly but not less important we find area and power consumption. These two metrics with difficulty limit the choice of a single instruction and then they are used as global parameters of choice: given a certain area and/or power consumption limit, choose a subset of P which fill properly this area and/or consume less that a certain amount of power. The generation of P_{Opt} can take some or all these metrics into account although more metrics, the harder is the process to provide a solution in terms of complexity.

Besides this type of metrics, the selection can be done following different policies: the element of P_{Opt} can be selected attempting to minimize the number of distinct templates that are used ([9]), or the number of instances of each template, or the number of nodes left uncovered in the graph, or in such a way that the longest path through the graph should have minimal delay, etc. ([23], [24]).

Finally the choice of the elements of P_{Opt} besides the metrics previously introduced has to manage also issues as template overlapping (e.g. $\{1, 2, 4\}$ and $\{1, 3, 5\}$ overlap at node 2 and then only one of them is enumerated) and node duplication, sometimes used to avoid template overlapping. Although the enumeration of overlapped templates can allows to produce a better result, especially under tight area constraint, this is not always considered ([6], [26]).

IV. Conclusions

The proposed paper has presented an overview of graph covering for generating instruction specific application instructions. This study inevitably has to address fundamental issues such as the graph isomorphism problem or the subset selection. Our aim has been to provide an overview of all relevant aspects of the problem giving a general view of the problem by aggregating the various approaches that only deal with different parts of the problem.

Future research will take the MISO-based analysis as a starting point to develop and propose powerful instruction set generation algorithms.

References

[1] FPGA - Introduction to FPGA processors. *http://www-li5.ti.uni-mannheim.de/fpga/?group/intro*.

[2] GREW A Scalable Frequent Subgraph Discovery Algorithm. M. kuramochi and g. karypis. Technical Report TR 04-024, Department of Computer Science and Engineering, University of Minnesota, June 2004.

[3] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami. A DAG-based design approach for reconfigurable VLIW processors. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference and Exhibition*, pages 778–779, Munich, Germany, 9-12 Mar. 1999.

[4] M. Arnold and H. Corporaal. Design domain specific processors. In *Proceedings of the 9th International Workshop on Hardware/Software CoDesign*, pages 778–779, Apr. 2001.

[5] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 40th Design Automation Conference*, pages 256–261, Anaheim, California, June 2003.

[6] M. Baleani, F. Gennari, Y. Jiang, Y. Pate, R. K. Brayton, and A. Sangiovanni-Vincentelli. HW/SW partitioning and code generation of embedded control application on a reconfigurable architecture platform. In *Proceedings of the 10th International Workshop on Hardware/Software Codesign*, pages 151–156, Estes Park, Colo., May 2002.

[7] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 262 – 269, Grenoble, France, 2002.

[8] L. Chen. Graph isomorphism and identification matrices: Parallel algorithms. In *IEEE Transactions on Parallel and Distributed Systems*, volume 7, no. 3, pages 308–319, March 1996.

[9] H. Choi, J. S. Kim, C. W. Yoon, I. C. Park, S. H. Hwang, and C. M. Kyung. Synthesis of application specific instructions for embedded DSP software. *IEEE Transactions on Computers*, 48(6):603–614, June 1999.

[10] N. Clark, W. Tang, and S. Mahlke. Automatically generating custom instruction set extension. In *Workshop on Application Specific Processors (WASP-1)*, Istanbul, Turkey, 19 Nov. 2002.

[11] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 129–140, 3-5 Dec. 2003.

[12] J. Cong, Y. Fan, G. Han, and Z. Zhang. Application specific instruction generation for configurable processor architectures. In *Proceeding of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pages 183–189, Monterey, California, 2004.

[13] O. Coudert and J. C. Madre. New ideas for solving covering problems. In *Proceedings of the 32nd ACM/IEEE Conference on Design Automation*, pages 641–646, San Francisco, California, June 1995.

[14] Olivier Coudert. On solving covering problems. In *Proceedings of the 33nd ACM/IEEE Conference on Design Automation*, pages 197–202, Las Vegas, Nevada, 3-7 June 1996.

[15] M. Anton Ertl. Optimal code selection in DAGs. *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 242–249, January 1999.

[16] Scott Fortin. The graph isomorphism problem. Technical Report TR 96-20, Department of Computing Science, University of Alberta, Canada, July 1996.

[17] D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *Proceedings of the 2003 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'03)*, pages 137–147, San Jose, California, 30 Oct. - 1 Nov. 2003.

[18] B. K. Holmer. *Automatic Design of Computer Instruction Sets.* PhD thesis, University of California, Berkeley, California, 1993.

[19] R. Kastner, A. Kaplan, S. Ogrenci Memik, and E. Bozorgzadeh. Instruction generation for hybrid reconfigurable system. *ACM Transactions on Design Automation of Embedded Systems (TODAES)*, 7(4):605–627, Oct. 2002.

[20] B. Kastrup, A. Bink, and J. Hoogerbrugge. ConCISe: A compile-driven CPLD-based instruction set accelerator. In *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, California, Apr. 1999.

[21] Y. Kukimoto, R. K. Brayton, and P. Sawkar. Delay-optimal technology mapping by dag covering. In *Proceedings of the 35th Annual Conference on Design Automation*, pages 348–351, San Francisco, California, 1998.

[22] M. Kuramochi and G. Karypis. An efficient algorithm for discovering frequent subgraphs. In *IEEE Trasactions on Knowledge and Data Engineering*, volume 16, no. 9, pages 1038–1051, Sept. 2004.

[23] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Instruction selection using binate covering for code size optimization. *IEEE/ACM International Conference on Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers*, pages 393–399, 5-9 Nov. 1995.

[24] S. Liao, K. Keutzer, S. Tjiang, and S. Devadas. A new viewpoint on code generation for directed acyclic graphs. In *ACM Transaction on Design Automation of Electronic Systems*, volume 3, no. 1, pages 51–75, Jan. 1998.

[25] Giovanni De Micheli and Rajesh K. Gupta. Hardware/Software co-design. *Readings in hardware/software co-design*, pages 30–44, 2001.

[26] L. Pozzi, M. Vuletić, and P. Ienne. Automatic topology-based identification of instruction-set extensions for embedded processors. Technical Report CS 01/377, EPFL, DI-LAP, Lausanne, Dec. 2001.

[27] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–180, 1994.

[28] Math World. *http://mathworld.wolfram.com/*.

[29] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 225–235, Vancouver, June 2000.

[30] P. Yu and T. Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 69–78, 2004.