The Instruction-Set Extension Problem: A Survey

Carlo Galuzzi and Koen Bertels*

Computer Engineering, EEMCS
Delft University of Technology, The Netherlands
{C.Galuzzi,K.L.M.Bertels}@ewi.tudelft.nl

Abstract. Over the last years, we have witnessed the increased use of Application-Specific Instruction-Set Processors (ASIPs). These ASIPs are processors that have a customizable instruction-set, which can be tuned towards specific requirements. The identification, definition and implementation of those operations that provide the largest performance improvement and that should be hardwired, extending in this way the Instruction-Set, constitutes a major challenge. The purpose of this paper is to investigate and study the issues regarding the customization of an Instruction-Set in function of the specific requirements of an application. Additionally, the paper provides an overview of all relevant aspects of the problem and compensates the lack of a general view of the problem in the existing literature.

1 Motivation

Electronic devices are very common in everyday life. It's enough to think about mobile phones, digital cameras, 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 in contrast to ASICs which 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 usually leads to more power consumption due to the inefficient units consumption. On the other hand, ASICs are able to satisfy specific constraints such as size, performance and power consumption using an optimal architecture for the application, but today designing and manufacturing an ASIC is a long and expensive process [1]. This design complexity grows exponentially due to shrinking geometries and the high mask and testing costs constitute a significant part of the manufacturing cost.

* This work was supported by the European Union in the context of the MORPHEUS project Num. 027342.

R. Woods et al. (Eds.): ARC 2008, LNCS 4943, pp. 209–220, 2008.

Over the last years, we have witnessed the increased use of GPPs that are combined with ASIPs. These ASIPs are processors situated in between GPPs and ASICs that have a customizable instruction-set, which can be tuned towards specific requirements. Time-to-market and reduced development costs have became increasingly important and have paved the way for reconfigurable architectures. These combine the flexibility of SW with the performance of HW. The higher cost/performance ratio for ASIPs have led researchers to look for methods and properties to maximize the performance of these processors. Each particular configuration can then be seen as an extension of the instruction-set. The identification, definition and implementation of those operations that provide the largest performance improvement and that should be hardwired, constitutes a major challenge.

The issues involved at each step are various and range from the isomorphism problem and the covering problem, well known computationally complex problems, to the function's study necessary for the guide function and the cost function, involved in the generation step and in the selection step respectively. Beside these, all the issues involved in this problem will be analyzed and studied in detail.

The customization of an instruction-set can be categorized in two main approaches. As the name suggests, complete customization involves the whole instruction-set which is tuned towards the requirements of an application [2,3,4,5], while partial customization involves the extension of an existing instruction-set by means of a limited number of instructions [6,7,8,9,10,11,12,13,14]. In both cases the goal is to design an Instruction-Set containing the most important operations needed by the application to maximize the performance. Besides providing an overall account, we also address considerations such as scalability, how to deal with overlapping instructions and how to address the complexity of the problem at hand.

The instruction-set customization problem represents a well specified topic where results and concepts from many different fields, such as engineering and graph theory are required. Especially the latter is the dominant approach and seems to provide the right analytical framework. Every application is thus represented by a directed graph and the required new complex instructions are seen as subgraphs having particular properties. The problem then translates into recognizing isomorphic subgraphs. Equally important are the covering and the selection problem. These are addressed by different techniques such as branch-and-bound, dynamic programming, etc. The proposed solutions are either exact, mathematical models whenever appropriate and possible or, given that the problem involved is known to be computationally complex, heuristics that are used in those cases where the mathematical solution is not computable.

The purpose of this paper is to investigate and study the issues regarding the customization of an instruction-set in function of the specific requirements of an application. The main goal of the paper is to provide a critical and detailed overview of all the aspects involved in instruction-set customization. The contribution of the paper is twofold: firstly, it provides an overview of **all relevant aspects** of the problem. Secondly, it compensates for the **lack** of a general view of the problem in the existing literature which only consists of **sporadic comparison** limited to isolated issues involved.

The paper is structured as follows. In Section 2, an introduction to the problem is presented. Section 3, 4 and 5 present the subproblems involved in the instruction-set customization, namely instruction generation and selection and the guide/cost function respectively. Section 6 presents an analysis of the type of instructions which is possible to generate. Concluding remarks and an outline of research conducted are given in Section 7.

2 Introduction to the Problem

Typically we start with a high level code, like C, that specifies the application and we manually specialize the embedded processor in a way that performance and cost constraints are satisfied. Irrespective of the type of customization, complete or partial, we can distinguish two approaches related to the level of abstraction on which we operate, i.e. the granularity at which code is considered: *fine-grained* and *coarse-grained*. The first one works at the operation level and implements small clusters of operations in HW [7,10,12,13,14,43,59,44]; the second one operates at the loop or procedure level and identifies critical loops or procedures in the application, and displaces them from SW to HW as a whole [16,17,18,19,20,21]. The main differences are in terms of speedups and flexibility: although a coarse-grained approach could produce a large speedup, its flexibility is limited, given that this approach is often performed on a per application basis and it is difficult that other applications have the same loop or procedure as critical part. Consequently many authors prefer either a fine-grained approach, even if it limits the achievable speedup compared to the coarse-grained one, or a mix of coarse and fine-grained techniques, since they operate at different levels and do not interfere with each other.

Basically the target is the identification of the operations that should be implemented in HW and the ones that have to be left for SW execution to achieve the requirements of the application. For this reason many authors naturally define this problem as a *HW-SW codesign problem* or *HW-SW partitioning* [22,23,24,25,26] which consists of concurrently balance at design time, the presence of HW and SW. The operations implemented in HW are incorporated in the processor either as new instructions and processor capabilities, in the form of special functional units integrated on the processor or implemented as peripheral devices. The interface between these systems parts is usually in the form of special purpose instructions embedded in the instruction stream. These HW components are more or less tightly coupled to the processor and involve different synchronization costs. Thus it becomes necessary also to select an appropriate communication and synchronization method within the architecture. The implementation of clusters of operations in HW as new complex operations, whatever nature they have, will benefit the overall performance only if the time the HW platform takes to evaluate them is less than the time required to compute the same operations in SW. As a result, compilation time and initialization time of the reconfigurable resources have to be considered as well.

At first, we profile the application SW looking for computation intensive segments of the code which, if mapped on HW, increases performance. The processor is then manually tailored to include the new capabilities. Although human ingenuity in manual creation of custom capabilities creates high quality results, performance and time-to-market requirements as well as the growing complexity of the design space, can benefit from an automatic design flow for the use of these new capabilities [28,29,12,30,31,32,13,14,60]. Moreover the selection of multiple custom instructions from a large set of candidates involves complex tradeoff and can be difficult to be performed manually.

There is a huge number of different interpretations and possible solutions to the instruction-set extension problem. Many authors adopt a graph theoretical approach in their work. Graph theory has became the dominant approach and seems to provide the right analytical framework. In this context the code of the application is represented with a directed graph, called the subject graph, and the intensive segments of the code to map on HW are subgraphs of the subject graph [12,14]. Depending on the level of abstraction on which we operate, nodes represent basic operations as well as entire procedures, functions or loops; edges represent data dependencies.

The extension of an instruction-set with new complex instructions can formally be divided into instruction generation and instruction selection. Given the application code, instruction generation consists of clustering of basic operations (such as add, or, load, etc.) or of mixed operations into larger and more complex operations. These complex operations are identified by subgraphs which can cover entirely or partially the subject graph. Once the *subgraphs* are identified, these are considered as single complex operations and they pass through a selection process. Generation and selection are performed with the use of a *guide function* and a *cost function* respectively, which take into account constraints that the new instructions have to satisfy to be implemented in HW. We now analyze instruction generation and instruction selection in more detail.

3 Instruction Generation

Instruction generation is mainly based on the concept of template. We call **template** a set of program statements that is a candidate for implementation as a custom instruction. Therefore a template is equivalent to a subgraph representing the list of statements selected in the subject graph, where nodes represent the operations and edges represent the data dependencies.

Instruction generation can be performed in two non exclusive ways: *using existing templates* or *creating new templates*. A collection of templates constitutes a **library of templates**. Many authors assume the existence of templates which are given as an input and which are identified inside the subject graph [33,6,31], however this is not always the case and many authors develop their own templates [16,17,34,7,10,35,14,43,59].

In the first case, instruction generation is nothing more than the identification of recurrences of specific templates from the library within the application. It is similar to the graph isomorphism problem [36,37,62]. In this case instruction generation can be considered as **template identification**. In the second case templates are identified inside the graph using a guide function. This function considers a certain number of parameters (often called constraints) and starting from a node taken as a seed, grows a template which respects all the parameters. Once a certain number of templates is identified the graph is usually reanalyzed to detect recurrences of the *built* templates.

The analysis of the application to identify instructions is often called design space exploration. We can detect a certain number of problems involved in instruction generation: (1) the complexity of the exploration, (2) the shape of the graph and (3) the overlapped templates.

A graph with n nodes contains 2^n subgraphs. Theoretically this means that there is an exponential number of possible new complex operations which can be selected inside a graph. This turns into an exponential complexity in the design space exploration. This problem can be avoided in two ways: reducing the design space explored, for example using heuristic instead of exact algorithms, or introducing more parameters into the guide function and introducing efficient bounding techniques. The use of heuristics, even though it reduces the design space explored, turns into the generation of non optimal solution or feasible ones, and they are often used with no theoretical guarantee. The introduction of additional parameters in the guide function can reduce the number of candidates for HW implementation, but has the drawback that every time a node is evaluated for a possible inclusion or not in the cluster, every parameter has to be satisfied and therefore the reduction of candidates turns into an increase of complexity of the approach due to the multiple analysis of the nodes.

A way to solve exactly covering problem is by using a **branch-and-bound** approach. This approach starts with a search space potentially exponential in size, and reduce step by step the search space using effective bounds and pruning techniques [38,39]. Other covering approaches use dynamic programming which is a way of decomposing certain hard to solve problems into equivalent formats that are more amenable to solution. 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. Other approaches, like [10], are based on dynamic programming, without the requirement that the subject graph and the patterns are trees.

The second difficulty concerns the shape of the graph. First of all graphs can be divided in cyclic and acyclic graphs. Usually only acyclic graphs are considered during the analysis. This follows from the fact that acyclic graph can be easily sorted, for example by a topological ordering, whereas cyclic graph cannot. Therefore the trouble of defining a one-to-one order of the nodes to the complexity of the problem is added. Moreover a cyclic graph can be transformed into an acyclic one if, for example, the cycles are unrolled. An other problem is given by the management of disconnected graphs. Even though the study of the problem including disconnected graphs in the analysis allows for exploiting the parallelism provided by considering each connected components at the same time [40,12,14], in many cases the authors have taken up only the study of connected graphs [10,28,25,31,32,41], shifting the study of disconnected graph in the study of k graphs, where k is the number of connected components.

The last problem is the management of overlapped templates[1] [32,42]. This problem is mainly related to the case when templates are provided. Usually, when a template is grown, the nodes included in the template are removed from the nodes subject to further analysis and therefore two disjointed templates can not overlap. This problem which, for instance, can be solved with the replications of the common nodes between the overlapped template, is very important. By the replication of few nodes, the cost of the replicated nodes can be paltry compared to the gain in performance which it is possible to get implementing in HW all the overlapped templates, especially under tight area constraint. Although mainly related to instruction generation, overlapped templates are a problem which affects also instruction selection.

4 The Guide Function and the Cost Function

Instruction generation as well as instruction selection make use of a function to identify or select the most profitable instructions to hardwire. These functions are called **guide function** and **cost function** respectively. They are strictly related one another and both are used to help the search of new instructions.

The aim of the guide function in template generation is to help the identification of a certain number of templates inside the graph. The output of the guide function is a set P defined as follows: $P = \{T_i \subseteq G, \text{ with } i \in \mathbb{N}\}$, where G is the subject graph and T_i are the templates identified in G.

Instruction selection makes use of a cost function. This function, similar to the guide function, is used to prune the set of candidates P generated during instruction generation. The main goal of the cost function is the identification of an optimal subset $P_{Opt} \subseteq P$ of templates. These templates satisfy a certain number of constraints. This is usually reflected into a reduction of the execution time of the application, and/or into a properly filling of the available area on the reconfigurable component, and/or into a minimization of the delay, and/or of reduction the power consumption, etc. Clearly the bigger is the size of P, i.e. the greater is the number of templates identified inside the subject graph, the harder is the selection of $P_{Opt} \subseteq P$. Although this can be seen as an additional problem, it is not always the case. A big size of P in terms of candidates for HW implementation becomes useful when the constraints are changed, shrunk or relaxed allowing different choices of the subset P_{Opt} satisfying the new constraints. As a consequence, the reconfigurability of the approach benefits.

The guide function usually includes physical constraints as parameters like the number of inputs and outputs. Apart from that, the guide function can include more generally constraints which, if respected, allows the implementation

[1] For example two subgraphs with set of nodes $\{1, 2, 4\}$ and $\{1, 3, 5\}$ respectively overlap at node 2 and then only one of them is enumerated.

in HW. A cost function, however, reduces a big set and leaves those elements which increase performance. The two functions are often considered together since they have a similar use. When the functions are considered independently, a right division of the parameters taken into account by the functions can reduce the complexity of the approach limiting the number of checks. For example [12] describes an approach for the generation of convex MIMO operations. The new operations are grown from a single operation/node taken as a seed and the adjacent nodes are evaluated for inclusion in the cluster. Every time a node is analyzed for inclusion in the cluster, the node passes through a triple check: inputs, outputs and convexity. Using this approach to identify convex MISO operations the complexity can be reduced. This because a MISO operation is naturally a convex operation [14,43,44] and therefore a check on single outputs of the final cluster naturally implies that the cluster is convex.

The main metrics which usually are all or part of the parameters used by the guide and cost functions are the following :

- *number of inputs and outputs*, usually related to the type of architecture used. Although limitations on input and output result in reduced performance, many architectures impose severe limitations on the characteristic of the final cluster to implement in HW.
- *area*, although it is hardly related to the single instructions, each instruction occupies a certain area, hence the total area of the cluster is an important factor;
- *execution time*, even though not possible to obtain accurate estimates of the system's cycle time in all cases. Therefore *cycle count* is often used as a substitute for the execution time;
- *power consumption.*

Usually a subset of the above metrics is used to identify and select an optimal set of new instructions. An exhaustive outline of metrics can be seen in [2, Chap.4]. One of the main goals when designing an instruction-set is to make the design appropriate for an implementation on many different technologies.

The coming of new technologies, and especially the increased use of reconfigurable technologies in the last decade can therefore lead researchers to think about the design of an instruction-set technology independent and suitable for multiple reuses. Theoretically exact, this concept has to deal with the effective implementation of an instruction-set which includes compilation time, initialization time as well as time for loading and reading parameters from memory or registers. Since these metrics are strictly dependent on the effective implementation, the design of an optimal instruction-set cannot be completely independent of the effective architecture on which it is implemented. Additional metrics can be identified in specific properties that the final cluster has to satisfy, as graph properties (like convexity, a property which guarantees a proper scheduling, etc.). Additional properties can be seen as a metric but in this survey we make a distinction between metrics and graph properties like connection, convexity, etc.

5 Instruction Selection

The main goal of instruction selection is the identification of a set of optimal new instructions to hardwire from a superset of candidates generated by the instruction generation step. One of the main problem during the selection of the best candidates is the covering of the design space: exact algorithms can be too expensive in terms of computational cost. Heuristics alone do not guarantee optimality, or even feasibility of the solution. The selection can follow different policies. The elements of P_{Opt}[2] can be selected attempting to minimize the number of distinct templates that are used [7], or the number of instances of each template, or the number of nodes left uncovered in the graph [45,46], or in such a way that the longest path through the graph should have minimal delay. Other approaches select instructions based on regularity, i.e. the repeated occurrence of certain templates [47,48,49,40], or resource sharing [50,51], or considering the frequency of execution, or the occurrence of specific nodes [11,52]. Instruction selection, guided by the cost function, can take one or more of these targets as parameters for an optimal choice of the instructions.

A way to address instruction selection is by using Integer Linear Programming (ILP) and more generally Linear Programming (LP) in combination with efficient LP solver. Basically each instruction is associated to a variable which can have integer value (Integer Linear Programming, ILP), non integer value (Linear Programming, LP), or boolean value (0-1 Linear Programming). The instructions, and then the variables, have to satisfy a certain number of constraints which are expressed with a system of linear inequalities and the optimal solution is the one that maximize or minimize the, so called, objective function. Example of instruction selection by using LP can be seen in [53,22,13,14].

A way to solve exactly covering problem is by using dynamic programming or branch-and-bound methods. Exact solutions are proposed in [54,55]. Clearly a method is efficient if it is possible to prevent the exploration of unsuccessful branches at earlier stages of the search, and this relies on efficient bounding techniques [38,39,56,57].

6 The Type of Instructions

Basically, there are two types of clusters that can be identified, based on the number of output values: Multiple Input Single Output (MISO) and Multiple Input Multiple Output (MIMO). Clearly the set of MIMO graphs includes the subset of MISO graphs. We identify these two types of graphs for a specific reason: the sequence of instructions to shift from SW to HW can be seen as a multivalued function: given $n \geq 1$ input the function produces $m \geq 1$ outputs: $(Out_1, ..., Out_m) = f(In_1, ..., In_n)$, which can be written in a short way using a vector notation as $\underline{Out} = f(\underline{In})$.

Accordingly, there are two types of algorithms for instruction set extensions which are briefly presented in this section.

[2] In case an optimal solution is not feasible, P_{Opt} contains elements which are *close-to-optimal.*

For the first one, a representative example is introduced in [58,28] which addresses the generation of MISO instructions of maximal size, called MAXMISO. The proposed algorithm exhaustively enumerates all MAXMISOs. Its complexity is linear with the number of nodes. The reported performance improvement is of some processor cycles per newly added instruction. Access to memory, i.e. load/store instructions, are not considered.

The approach presented in [32] targets the generation of general MISO instructions. The exponential number of candidate instructions turns into an exponential complexity of the solution in the general case. In consequence, heuristic and additional area constraints are introduced to allow an efficient generation. The difference between the complexity of the two approaches in [32,58] is due to the properties of MISOs and MAXMISOs: while the enumeration of the first is similar to the subgraph enumeration problem (which is exponential) the intersection of MAXMISOs is empty and then once a MAXMISO is identified, it is removed generating a linear enumeration of them. A different approach is presented in [44] where, with an iterative application of the MAXMISO clustering presented in [58], MISO instructions called SUBMAXMISOs are generated with linear complexity in the number of processed elements. The iterative application of this algorithm allows the generation of MISO instructions of smaller size at each iteration when, for instance, tight limitations on the total number of inputs are applied.

The algorithms of second type are more general and provide more significant performance improvements. However they also have exponential complexity. For example, in [12] the identification algorithm detects optimal convex MIMO subgraphs based on Input/Output constraints but the computational complexity is exponential. A similar approach described in [41] proposes the enumeration of all the instructions based on the number of inputs, outputs, area and convexity. The selection problem is not addressed. Contrary to [12] which has scalability issues if the data-flow graph is very large or the micro-architectural constraints are too fine, this approach is quite scalable and can be applied on large data-flow graphs with relaxed micro-architectural constraints. The limitation to only connected instructions has been removed in [61], where the authors address the enumeration of the disconnected instructions.

In [13] the authors target the identification of convex clusters of operations given input and output constraints. The clusters are identified with a ILP based methodology. The main characteristic is that they iteratively solve ILP problems for each basic block. Additionally, the convexity is verified at each iteration increasing in this way the overall complexity of the approach.

In [14] the authors address the generation of convex MIMO operations in a manner similar to [13] although the identification of the new instructions is rather different. The authors construct convex MIMO based on MAXMISOs clustering in order to maximally exploit the MAXMISO level parallelism. The main difference between this approach and [13] is that the latter iteratively solves ILP problems for each basic block, while the former has one global ILP problem for the entire procedure. Additionally the convexity is addressed differently: in [13] the convexity is verified at each iteration, while in [14] it is guaranteed by construction.

An extension of the work in [14] is presented in [43]. In [43], the authors present a heuristic of linear complexity which address the generation of convex MIMO instruction. The key difference between the two solutions presented in [14] and [43] is the combination per levels. Since single MAXMISO execution in HW does not provide huge improvements in performance, the main idea is to combine, per levels, MAXMISOs available at the same level in the reduced graph, into a convex MIMO that is executed as a single instruction in HW where convexity is theoretically guaranteed. The idea of combining MAXMISO per level(s) has been further extended in [44,59] where linear complexity algorithms based on the notion of MAXMISO are presented.

7 Conclusions

In this paper, we presented an overview of the Instruction-Set extension problem providing an analysis of all relevant aspects involved in the problem. It compensates the lack of a general view of the problem in the existing literature which only consists of sporadic comparisons that address only a limited number of the issues involved. Additionally, we provided an in-depth analysis of all the subproblems involved. Therefore, our study benefits different kinds of readers ranging from the one simply interested in the issues involved in the problem, to the one interested in advancing the state-of-the-art and needs to know in detail the existing approaches and the open issues.

References

1. Keutzer,: From ASIC to ASIP: The next design discontinuity. In: ICCD 2002 (2002)
2. Holmer: Automatic design of computer instruction sets. PhD thesis (1993)
3. Huang: Generating instruction sets and microarchitectures from applications. In: ICCAD 1994, (1994)
4. Huang: Synthesis of instruction sets for pipelined microprocessors. In: DAC 1994, (1994)
5. Van Praet: Instruction set definition and instruction selection for ASIPs. In: ISSS 1994, (1994)
6. Liem: Instruction-set matching and selection for DSP and ASIP code generation. In: ED & TC 1994, (1994)
7. Choi,: Synthesis of application specific instructions for embedded DSP software. IEEE Trans. on Comp. 48(6), 603–614 (1999)
8. Faraboschi,: LX: a technology platform for customizable VLIW embedded processing. ACM SIGARCH Computer Architecture News, Special Issue. In: Proceedings of the 27th annual international symposium on Computer architecture (ISCA 2000) 28(2), 203–213 (2003)
9. Wang: Hardware/software instruction set configurability for System-on-Chip processors. In: DAC 2001 (2001)
10. Arnold: Designing domain-specific processors. In: CODES 2001(2001)
11. Kastner,: Instruction generation for hybrid reconfigurable systems. ACM TODAES 7(4), 605–627 (2002)
12. Atasu: Automatic application-specific instruction-set extensions under microarchitectural constraints. In: DAC 2003 (2003)

13. Atasu: An integer linear programming approach for identifying instruction-set extensions. In: CODES+ISSS 2005 (2005)
14. Galuzzi: Automatic selection of application-specific instruction-set extensions. In: CODES+ISSS 2006 (2006)
15. Alomary: A hardware/software codesign partitioner for ASIP design. In: ICECS 1996 (1996)
16. Athanas,: Processor reconfiguration through instruction-set metamorphosis. IEEE Computer 26(3), 11–18 (1993)
17. Razdan: PRISC software acceleration techniques. In: ICCS 1994 (1994)
18. Wirthlin: DISC: The dynamic instruction set computer. In: FPGAs for Fast Board Devel. and Reconf. Comp. vol. 2607, pp. 92–103 (1995)
19. Geurts: Synthesis of Accelerator Data Paths for High-Throughput Signal Processing Applications. PhD thesis (1995)
20. Geurts,: Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications. Kluwer Academic Publishers, Norwell (1997)
21. Hauser: GARP: a mips processor with a reconfigurable coprocessor. In: FCCM 1997 (1997)
22. Niemann: Hardware/software partitioning using integer programming. In: EDTC 1996 (1996)
23. Niemann,: An algorithm for hardware/software partitioning using mixed integer linear programming. ACM TODAES, Special Issue: Partitioning Methods for Embedded Systems 2(2), 165–193 (1997)
24. De Micheli,: Hardware/software co-design. Proc. of IEEE 85(3), 349–365 (1997)
25. Baleani, Sangiovanni-Vincentelli, A.: HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In: CODES 2002 (2002)
26. Arató: Hardware-software partitioning in embedded system design. In: WISP 2003 (2003)
27. Gschwind: Instruction set selection for ASIP design. In: CCODES 1999 (1999)
28. Pozzi: Automatic topology-based identification of instruction-set extensions for embedded processors. Technical Report CS 01/377, EPFL, DI-LAP, Lausanne (December 2001)
29. Clark: Automatically generating custom instruction set extensions. In: WASP 2002 (2002)
30. Peymandoust: Automatic instruction set extension and utilization for embedded processors. In: ASAP (2003)
31. Clark,: Processor acceleration through automated instruction set customization. In: MICRO 36
32. Cong: Application-specific instruction generation for configurable processor architectures. In: FPGA 2004 (2004)
33. Rao, S.: Partitioning by regularity extraction. In: DAC 1992 (1992)
34. Arnold: Automatic detection of recurring operation patterns. In: CODES 1999 (1999)
35. Kastner: Instruction generation for hybrid reconfigurable systems. In: ICCAD 2001 (2001)
36. Fortin: The graph isomorphism problem. Technical Report TR 96-20, Department of Computing Science, University of Alberta, Canada (July 1996)
37. Chen,: Graph isomorphism and identification matrices: Parallel algorithms. IEEE Trans. on Paral. and Distr. Systems 7(3), 308–319 (1996)
38. Coudert: New ideas for solving covering problems. In: DAC 1995 (1995)

39. Coudert: On solving covering problems. In: DAC 1996 (1996)
40. Brisk: Instruction generation and regularity extraction for reconfigurable processors. In: CASES 2002 (2002)
41. Yu: Scalable custom instructions identification for instruction-set extensible processors. In: CASES 2004 (2004)
42. Aletà,: Removing communications in clustered microarchitectures through instruction replication. ACM TACO 1(2), 127–151 (2004)
43. Vassiliadis, S., Bertels, K., Galuzzi, C.: A Linear Complexity Algorithm for the Automatic Generation of Convex Multiple Input Multiple Output Instructions. In: Diniz, P.C., Marques, E., Bertels, K., Fernandes, M.M., Cardoso, J.M.P. (eds.) ARCS 2007. LNCS, vol. 4419, pp. 130–141. Springer, Heidelberg (2007)
44. Galuzzi: A linear complexity algorithm for the generation of multiple input single output instructions of variable size. In: SAMOS VII Works
45. Liao: Instruction selection using binate covering for code size optimization. In: ICCAD 1995 (1995)
46. Liao,: A new viewpoint on code generation for directed acyclic graphs. ACM TODAES 3(1), 51–75 (1998)
47. Rao, S.: On clustering for maximal regularity extraction. IEEE Trans, on CAD 12(8), 1198–1208 (1993)
48. Rao, S.: Hierarchical design space exploration for a class of digital systems. IEEE Trans. on VLSI Systems 1(3), 282–295 (1993)
49. Janssen: A specification invariant technique for regularity improvement between flow-graph clusters. In: EDTC 1996 (1996)
50. Huang: Managing dynamic reconfiguration overhead in system-on-a-chip design using reconfigurable datapaths and optimized interconnection networks. In: DATE 2001 (2001)
51. Moreano: Datapath merging and interconnection sharing for reconfigurable architectures. In: ISSS 2002 (2002)
52. Sun: Synthesis of custom processors based on extensible platforms. In: ICCAD 2002 (2002)
53. Imai: An integer programming approach to instruction implementation method selection problem. In: EURO-DAC 1992 (1992)
54. Grasselli,: A method for minimizing the number of internal states in incompletely specified sequential networks. IEEE Trans. Electron. Comp. EC-14, 350–359 (1965)
55. Brayton: Boolean relations and the incomplete specification of logic networks. In: ICCAD 1989 (1989)
56. Liao: Solving covering problems using LPR-based lower bounds. In: DAC 1997 (1997)
57. Li: Effective bounding techniques for solving unate and binate covering problems. In: DAC 2005 (2005)
58. Alippi: A DAG-based design approach for reconfigurable VLIW processors. In: DATE 1999 (1999)
59. Galuzzi: The spiral search: A linear complexity algorithm for the generation of convex multiple input multiple output instruction-set extensions. In: ICFPT 2007 (2007)
60. Huynh: An Efficient Framework for Dynamic Reconfiguration of Instruction-Set Customizations. In: CASES 2007 (2007)
61. Yu: Disjoint pattern enumeration for custom instructions identification. In: FPL 2007 (2007)
62. Bonzini: A retargetable framework for automated discovery of custom instructions. In: ASAP 2007 (2007)