The Instruction-Set Extension Problem: A Survey

CARLO GALUZZI and KOEN BERTELS, Delft University of Technology

The extension of a given instruction-set with specialized instructions has become a common technique used to speed up the execution of applications. By identifying computationally intensive portions of an application to be partitioned in segments of code to execute in software and segments of code to execute in hardware, the execution of an application can be considerably speeded up. Each segment of code implemented in hardware can then be seen as a specialized application-specific instruction extending a given instruction-set. Although a number of approaches exist in literature proposing different methodologies to customize an instruction-set, the description of the problem consists only of sporadic comparisons limited to isolated problems. This survey presents a unique detailed description of the problem and provides an exhaustive overview of the research in the past years in instruction-set extension. This article presents a thorough analysis of the issues involved during the customization of an instruction-set by means of a set of specialized application-specific instructions. The investigation of the problem covers both instruction generation and instruction selection and different kinds of customizations are analyzed in a great detail.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General—*Instruction set design (e.g., RISC, CISC, VLIW)*

General Terms: Design

Additional Key Words and Phrases: Instruction-set, customization, instruction-set extension, instruction generation, instruction selection, HW/SW codesign, reconfigurable architecture

ACM Reference Format:
Galuzzi, C. and Bertels, K. 2011. The instruction-set extension problem: A survey. ACM Trans. Reconfig. Technol. Syst. 4, 2, Article 18 (May 2011), 28 pages.
DOI = 10.1145/1968502.1968509 http://doi.acm.org/10.1145/1968502.1968509

1. INTRODUCTION

In the past years, electronic devices have been steadily penetrating the market, featuring not only an ubiquitous nature but also a plethora of functionalities. During the years, these functionalities have been implemented by using different kinds of computer architectures which can be categorized according to their degree of flexibility into two main groups: the general-purpose computing group and the application-specific computing group [Bobda 2007; Guo 2006].

1.1. General-Purpose Computing

General-purpose architectures have been widely used and studied in the past decades. This type of architectures provides a high degree of flexibility in terms of application

This work was supported by the European Union in the context of the Morpheus Project no. 027342, the Artemisia iFEST project (grant 100203), the Artemisia SMECY project (grant 100230), and the FP7 Reflect project (grant 248976).
Authors' address: C. Galuzzi (corresponding author) and K. Bertels, Delft University of Technology, The Netherlands; email: C.Galuzzi@tudelft.nl.

DOI 10.1145/1968502.1968509 http://doi.acm.org/10.1145/1968502.1968509

domains. “Additionally, many tools have become available on the market and have allowed programmers to map many different applications onto this type of architectures virtually effortlessly” [Guo 2006].

The general-purpose computing group is based on the Von Neumann computing model. The general structure of a Von Neumann machine consists of a memory for holding both program instructions and data (Harvard architectures contain two parallel accessible memories for holding the program instructions and the data separately), a control unit used to store the addresses of the instructions to execute, and an arithmetic and logic unit used to execute the instructions [Bobda 2007].

A program targeting a Von Neumann machine is coded as a set of instructions to be executed sequentially. The execution of an instruction is realized in five steps: (1) fetching the instruction from the program memory, (2) decoding the instruction to determine which operation has to be executed and which operands are required, (3) reading the operands from the memory, (4) executing the instruction, and (5) writing the result of the operation back to the data memory. This execution model results in a high performance overhead for each individual operation, which turns into energy overhead. In this sense, the general-purpose computing group is considered to be the most flexible hardware at the cost of a general high-energy consumption [Bobda 2007; Guo 2006].

Over the years, different techniques to increase the level of parallelism have been introduced at instruction level: for instance, techniques such as instruction pipelining, superscalar execution, out-of-order execution, and register renaming. Parallelism has also been exploited at other levels: bit-level, data-level, and loop-level parallelism. Although the level of parallelism has been increased over the years, it is still relatively limited for highly parallelizable applications, which become poor candidates for implementation on these architectures.

1.2. Application-Specific Computing

In the context of application-specific computing, three main categories can be identified: Application-Specific Integrated Circuits (ASICs), Application Domain-Specific Processors (ADSPs), and Application-Specific Instruction-set Processors (ASIPs).

ASICs are circuits designed for a specific application such as the processor in a TV set top box. Being designed for a specific use, ASICs are able to satisfy specific constraints and to reduce energy consumption, using an appropriate architecture designed for the targeted application, compared with general purpose architectures which are designed for a generic use. In an ASIC, the entire application has been hard-wired and the software component is usually represented by runtime configurable parameters. However, energy saving comes at the cost of low flexibility and programmability: for each new functionality or application, the hardware has to be redesigned and built. Today designing and manufacturing an *ASIC* is a time-consuming and expensive process [Keutzer et al. 2002]. The increasing NonRecurring Engineering (NRE) costs, due to the high mask and testing costs associated with manufacturing, together with factors such as Deep SubMicron effects (DSM), increased feature sets, and heterogeneous integration contribute to increase the production costs. Additionally, this long process has to deal with the shrinking time-to-market which sometimes makes the choice of an ASIC not suitable.

ADSPs and ASIPs are processors having a partially customizable instruction-set which can be tuned towards the specific requirements of an application (ASIPs) or a domain of applications (ADSPs) by extending the basic instruction-set with dedicated instructions. Digital signal processors are an example of ADSP. “These processors are specialized for accelerating computation of repetitive, numerically intensive tasks in

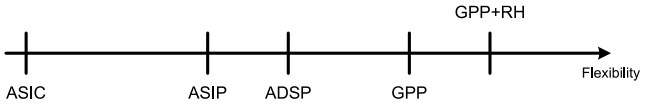

Fig. 1. Positioning of different computer architectures in terms of flexibility. GPP + RH represents a reconfigurable architecture composed by a GPP and a Reconfigurable Hardware (RH).

the digital-signal processing area such as, for example, multimedia and image processing" [Bobda 2007]. A typical application-specific instruction implemented on a DSP processor is the Multiply ACcumulate (MAC) instruction which can be performed on huge set of data concurrently. A MAC instruction performed on a common Von Neumann machine would have to access the memory to load/store the intermediate result. As a result, by using specialized hardware that directly perform addition after multiplication without having to access the memory a considerable amount of time can be saved. If the processor has to be used only for one application, ASIPs can be used instead of ADSPs. From an optimization point of view, ASIPs can be better optimized than ADSPs. This happens because modifications to the latter have to benefit all the applications in a specific domain, whereas in the former case only one application is taken into consideration [Arnold 2001].

The customizable instruction-set of ADSPs and ASIPs introduce more flexibility in the design even though the number of different instruction-set customizations is usually relatively limited and, therefore, the execution of different applications can be inefficient [Fornaciari et al. 1999].

The aforementioned architectures can then be positioned in terms of flexibility as depicted in Figure 1. The flexibility of a General-Purpose Processor (GPP) can be further extended by using a reconfigurable hardware, as described in the next section.

1.3. Reconfigurable Computing

Ideally, we would like to combine the flexibility of a general purpose system with the high performance of an application-specific system. The last two decades have seen a new emerging class of architectures, the so-called reconfigurable architectures. Time-to-market and reduced development costs have became increasingly important and have paved the way for reconfigurable architectures. Reconfigurable devices, including the most widely used Field-Programmable Gate Arrays (FPGAs)[1], consist of

> "arrays of programmable logic cells interconnected using a set of routing resources which are also reconfigurable. In this way, custom digital circuits can be mapped onto the reconfigurable hardware by computing the logic functions of the circuit within the logic blocks and the reconfigurable routing is used to connect the logic blocks together to form the necessary circuit" [Compton and Hauck 2002].

Reconfigurable architectures are typically formed with a combination of a conventional processor, like a General-Purpose Processor (GPP), and a reconfigurable device. Part of the operations are executed by the host processor while the rest of the operations are executed by the reconfigurable device[2]. A reconfigurable architecture is an architecture able *to adapt* to the application: the structure of the architecture can change at start-up time or even at runtime to match the applications.

[1]Xilinx (http://www.xilinx.com/) and Altera (http://www.altera.com/) are currently the main producers of FPGAs devices on the market.

[2]As described later in Section 5, it is also possible to embed the processor into the reconfigurable device either as a hard core or as a soft core implemented on resources of the reconfigurable hardware itself.

Reconfigurable architectures present three main advantages compared with the architectures previously described: first, changing an existing architecture, rather than defining a completely new one, allows to reuse its associated compiler which has to be partially modified and not redesigned from scratch [Pozzi 2000]. Second, reconfigurable architectures can serve a much wider range of applications, being an *extension* of GPP (or a processor, in the general case) (see Figure 1). Examples are data encryption, data compression, and genetic algorithms. Third, reconfigurable architectures can be used for rapid prototyping. "Rapid prototyping allows a device to be tested in real hardware before its final production" [Bobda 2007]. In this way, considerable amounts of development and debugging efforts can be eliminated and the time-to-market can be reduced. Additionally, the design remains flexible until the product enters the market and even after, allowing to ship a product that meets the minimum requirements and add features after deployment [Bobda 2007].

The higher cost/performance ratio for reconfigurable architectures has led researchers to look for methods and properties to maximize the performance. Each particular configuration can then be seen as an extension of the instruction-set of the host processor. The identification, definition, and implementation of those operations that provide the largest performance improvement constitutes a major challenge and represents the so-called *instruction-set extension problem*.

In this article, we present a survey of current research in instruction-set extension, investigating the issues regarding the customization of an instruction-set under specific requirements. The main objective is to provide a detailed overview of all the aspects involved in the customization of an instruction-set. It does not seek to cover every technique and research project in instruction-set extension. Instead, it provides an overview of all relevant aspects of the problem and it compensates for the lack of a general view of the problem in the existing literature, which only consists of sporadic comparisons limited to isolated issues involved.

2. INSTRUCTION-SET EXTENSIONS

The customization of an instruction-set presents, among others, many advantages: first,

> "the application code can be more densely encoded, resulting in a code size reduction; second, the total number of instructions that have to be executed may be reduced, which results in a lower power consumption and third, the execution of the application can be more efficient in terms of increased performance using the customized instruction" [Arnold 2001].

Although the focus of this article is on presenting in detail how to generate and select custom instructions for extending a given instruction-set, the issue concerning the efficient implementation of the selected instructions in hardware has to be addressed as well. Later in the article, we give an overview of different architectures that integrate custom instructions for application acceleration.

The identification process of new specialized instructions is usually subject to different constraints such as power consumption, area, code size, cycle count, operating frequency, etc. Additionally, not all the instructions suitable for a hardware implementation can be selected for being implemented in hardware, due to the ever-limited hardware resources in the general case. The issues involved are diverse and range from the isomorphism problem and the covering problem, well-known computationally complex problems, to the function's study necessary for the guide/cost function involved in the generation and selection of custom instructions. Equally important is the selection problem addressed by different techniques such as branch-and-bound and

dynamic programming. The proposed solutions are either exact, whenever appropriate and possible or, given that the problems involved are known to be computationally complex, heuristics that are used in those cases where the solution is not computable in a feasible time. In the next sections, we overview the current state-of-the-art in instruction-set customization, describing in detail all the issues involved.

The instruction-set customization problem represents a well-specified topic where results and concepts from many different research fields are required. Graph theory is one of the dominant approaches and it seems to provide the right analytical framework. Thinking about the data-flow or control-flow graphs of an application[3], it is easy to imagine an application represented by a directed graph, where nodes represent operations and edges represent data dependencies, and the required new complex instructions are represented by subgraphs having particular properties. Thus, the problem turns into the identification of methods for the recognition of certain types of subgraphs.

The remainder of this article is presented as the following. Section 3, after presenting a motivational example, overviews the different instruction-set customizations. Degree of customization, granularity of the instructions, and degree of automation of the process are presented in detail. Section 4 elaborates on the customization process and provides a detailed account of the problems involved in the customization. Instruction generation and selection, properties of the custom instructions, and existing solutions are presented to better understand the problem. Section 5 proposes a selected overview of the main architectural approaches that integrate custom logic for application acceleration. Finally, Section 6 presents concluding remarks and open issues worthy of further research and investigation in the context of instruction-set customization.

3. DIFFERENT TYPES OF CUSTOMIZATIONS

Instruction-set customization can be pursued by following different approaches in the type of customization, which can be complete or partial, and in the granularity of the instructions, which can be fine-grain or coarse-grain. We introduce a motivational example to informally outline the main idea of the instruction-set extension.

3.1. Motivational Example

In Figure 2(a), we present a data-flow subgraph extracted from the ADPCM application as implemented in the MediaBench benchmark suite [Lee et al. 1997]. Nodes represent the primitive operations, namely the instructions belonging to the instruction-set and the edges representing the data dependencies.

A custom instruction is represented by a subgraph of the data-flow graph. The main idea is to identify different clusters of basic operations within the graph, which can be implemented as single instructions to atomically execute in hardware. They become new specialized instructions extending the basic instruction-set and they allow to speed up the execution of an application. Although different criteria can be used to identify custom instruction, all instructions, as we will see in the next sections, can be divided in *single-output* and *multiple-output*. Additionally, an instruction can perform one or more parallel independent calculations at the same time, which means that all instructions can be divided into two sets: *connected* and *disconnected* instructions (see Section 4.3.1). Figure 2 presents an example of different custom instructions generated as in Alippi et al. [1999] and Galuzzi et al. [2006, 2007a].

[3]These are the directed graphs that show, respectively, the data dependencies and the control dependencies among a number of functions.

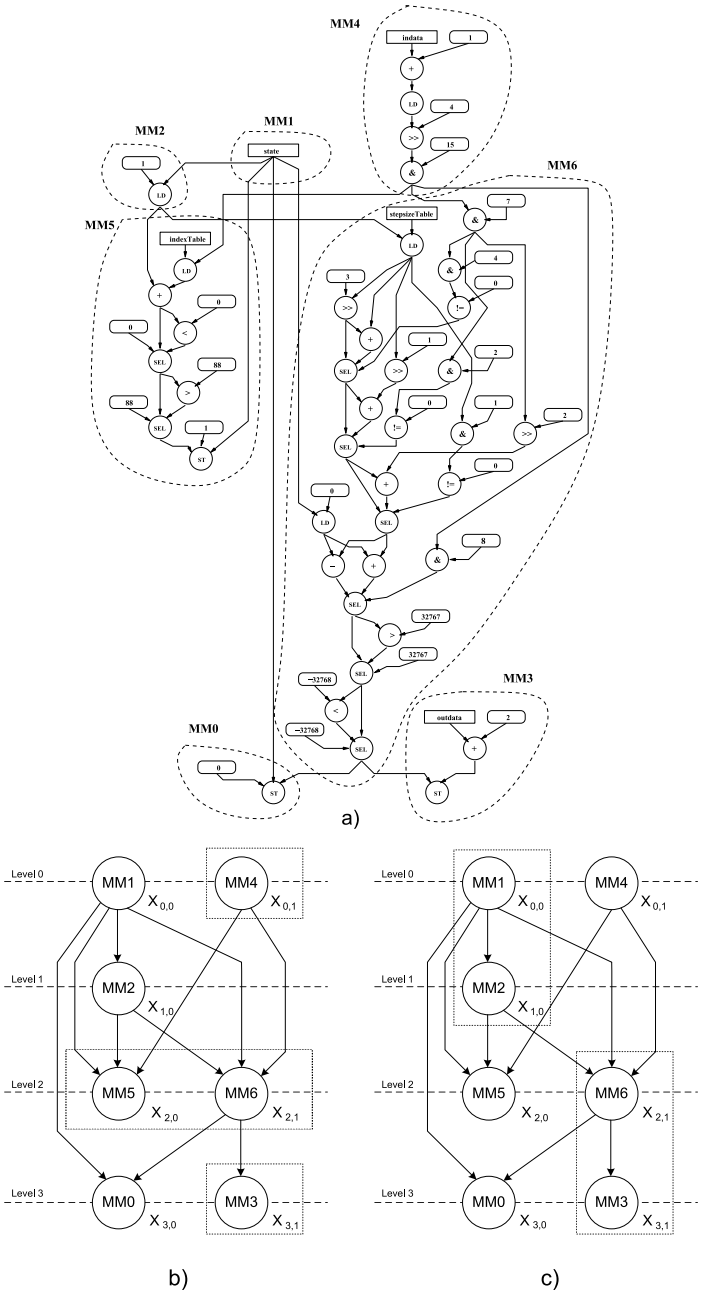

Fig. 2. Motivational example. The data-flow subgraph extracted from *ADPCM* decoder and different custom instructions: *(a)* maximal connected single-output instructions [Alippi et al. 1999], *(b)* disconnected multiple-input multiple-output instructions [Galuzzi et al. 2006], and *(c)* connected multiple-input multiple-output instructions [Galuzzi et al. 2007a].

In Figure 2(a), the nodes of the graph are partitioned in maximal single-output subgraphs (the dashed boxes) as described in Alippi et al. [1999]. Each cluster is a connected single-output subgraph. Considering each of these subgraphs fused as a single complex multiple-input and single-output instruction, it is possible to draw the graphs in Figure 2(b) and 2(c), where each node represents one of the clusters identified in Figure 2(a). Following the clustering methodology proposed in Galuzzi et al. [2006], the nodes of the graph are further combined in multiple-input multiple-output disconnected subgraphs, the dashed boxes in Figure 2(b). Following the method proposed in Galuzzi et al. [2007a], the nodes of the graph are further combined in multiple-input and multiple-output connected subgraphs, the dashed boxes in Figure 2(c).

Additionally, we can roughly calculate the performance gain for these instructions[4]. Let's now assume that the hardware latency for a node n_i in Figure 2(a) to be l_i. When k nodes at the same level are combined together, the execution time of the cluster in hardware is $\max_{i=1..k} l_i$. The performance gain in this case is $\sum_{i=1..k}(l_i) - \max_{i=1..k}(l_i)$. If, successively, we combine nodes through the levels of the graph, the overall performance gain increases. Let's assume that $\alpha_1, ..., \alpha_h$ are the levels of the nodes belonging to a cluster. The overall performance gain in this case is

$$\sum_{j=\alpha_1}^{\alpha_h}\left(\sum_{i_j} l_{i_j} - \max_{i_j}(l_{i_j})\right). \tag{1}$$

This means, for example, that using the custom instruction in Figure 2(b) there is a performance gain of $l_5 + l_6 - \max(l_5, l_6)$.

Roughly speaking, the identification of custom instructions partitions an application in segments of code which are implemented in software and segments of code which are implemented in hardware. For this reason, many authors naturally associate this problem to the *hardware-software codesign problem* or *hardware-software partitioning problem* [Bình et al. 1995; Niemann and Marwedel 1996, 1997; De Micheli and Gupta 1997; Baleani et al. 2002; Arató et al. 2003; Huynh et al. 2007], which consists of concurrently balancing, at design time, the presence of hardware and software.

3.2. Types of Customizations

The identification of custom instructions for instruction-set extension can be categorized according to the following.

Complete Customization vs. Partial Customization. The previous example shows three different clustering methods which *extend* a given instruction-set with different kinds of specialized instructions: single-output instructions and connected or disconnected multiple-output instructions. 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 or a domain of applications [Holmer 1993; Huang and Despain 1994a, 1994b; Van Praet et al. 1994]. Partial customization involves the extension of an existing instruction-set by means of a limited number of instructions [Alomary 1996; Arnold and Corporaal 2001; Atasu et al. 2003a; Choi et al. 1998, 1999; Faraboschi et al. 2000; Kastner et al. 2002; Liem et al. 1994; Wang et al. 2001]. In both cases, the goal is to design an instruction-set that contains the most important operations needed by one or more applications to maximize the performance of execution. By extending an instruction-set rather than designing a completely new one, it is possible, for example, to reuse its

[4]In this example, we consider a performance gain over a single issue, in-order CPU.

associated compiler which has to be partially modified and not redesigned from scratch [Pozzi 2000].

Fine Granularity vs. Coarse Granularity. Irrespective of the type of customization, complete or partial, we can distinguish two approaches related to the granularity at which code is considered: *fine-grain* and *coarse-grain*[5]. The first one works at operation level and implements small clusters of operations in hardware [Arnold and Corporaal 2001; Atasu et al. 2003a, 2005; Choi et al. 1999]. The second one operates at loop or procedure level and identifies critical loops or procedures in the application, and displaces them from software to hardware as a whole [Athanas and Silverman 1993; Geurts 1995, 1997; Hauser and Wawrzynek 1997; Razdan et al. 1994; Wirthlin and Hutchings 1995]. The main differences are in terms of speedup and flexibility: although a coarse-grain approach can produce a large speedup, its flexibility is limited. This appears 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 researchers prefer either a fine-grain approach, even if it limits the achievable speedup compared to the coarse-grain one, or a mix of coarse- and fine-grain techniques, when these do not interfere with each other [Arnold 2001]. For example, in Figure 2, the custom instructions have a fine granularity.

Automatic Extension vs. Maual Extension. An important issue related to the extension of an instruction-set is the degree of human effort required to identify and implement the instruction-set extensions. 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, can benefit from an automatic design flow for the use of these new capabilities [Atasu et al. 2003a, 2005; Bonzini and Pozzi 2007b; Borin et al. 2004; Clark and Zhong 2005; Clark et al. 2002, 2003; Cong et al. 2004; Huynh et al. 2007; Peymandoust et al. 2003; Sun et al. 2004]. Moreover, the selection of multiple custom instructions from a large set of candidates involves complex trade-offs and can be difficult to be performed manually, making often "the design efforts more time consuming and expensive than the design of an ASIC" [Clark 2007]. There also are commercial products available for automatic instruction-set customization. Examples are Tensilica's Xtensa $LX2$ processor and the $MIPS$ Pro series.

Up to now, we described the different types of instruction-set customizations. Then, an important issue arises: How can we extend a given set of instructions with custom instructions? Given an application, the design process involves first the identification of segments of code to speed up. Second, the segments are analyzed for the generation of custom instructions and, then, a subset of the most profitable instructions is selected for hardware implementation based on hardware limitations. Thus, the customization process can mainly be divided in two phases: instruction generation and instruction selection. Given an application or part of an application code, instruction generation consists of clustering of basic operations (such as add, or, load, etc.) (the ones belonging to the instruction-set) or of mixed operations into larger and more complex operations. The custom instructions can cover the application entirely or partially. Once the new instructions are identified, they pass through a selection process which selects a subset of the most profitable ones. Instruction generation and selection are performed with the use of a function called *cost function* or *guide function*, which takes into account different constraints and guides the identification and selection of the new instructions.

[5]The word granularity in this context does not have to be confused with the granularity of a reconfigurable device, which refers to the size of the reconfigurable blocks.

In the next sections, instruction generation and instruction selection are analyzed in detail.

4. THE CUSTOMIZATION PROCESS

The generation of new instructions relies on the concept of template. A *template* is a set of program statements that is a candidate for implementation as a custom instruction. As mentioned before, an application can be described with graphs, such as the data-flow graph and the control-flow graph. In this context, a template is equivalent to a subgraph where nodes represent operations and edges represent dependencies. A collection of different templates constitutes a *library of templates*.

4.1. Custom Templates vs. Predefined Templates

A template can be, for example, the multiply accumulate (MAC) operation, a very common operation in signal processing areas. An approach which looks at methods to automatically identify parts of the code to move from software to hardware can make use of templates from preexisting libraries, as in the case of the MAC operation, or it can build a custom library of templates for the application or domain of applications under consideration.

When preexisting templates are used, the used templates represent the instruction-set extensions. The general two-step process, instruction generation and instruction selection for the identification of custom instructions, is reduced to a single step in which the application is analyzed to find recurrences of the given templates. It is similar to the graph isomorphism problem [Chen 1996; Fortin 1996; Messmer and Bunke 1995]. Many approaches assume the existence of predefined libraries of templates [Cheung et al. 2003a; Clark et al. 2003; Liem et al. 1994; Sreenivasa Rao and Kurdahi 1992]. However, this is not always the case and many authors develop their own templates [Arnold and Corporaal 2001; Atasu et al. 2003a; Athanas and Silverman 1993; Choi et al. 1999; Kastner et al. 2001; Pozzi et al. 2006a; Razdan et al. 1994]. In the general case, custom templates are generated through an incremental clustering. A node is selected as a seed and, iteratively, nodes are merged together following different policies.

One of the main goals in designing a method to extend a given instruction-set with dedicated instructions is to make the method, in a certain way, suitable to be applied on different architectures. Unfortunately, this concept has to deal with the effective implementation of the instruction-set on the architecture, which can have specific hardware limitations. For example, if the architecture allows operations with no more than one output, a custom instruction with multiple outputs cannot be implemented in hardware, making unusable the custom instruction identified. For this reason, the generation of custom instructions is subject to specific constraints.

4.2. The Cost Function

The generation of custom instructions makes use of a function, called *cost function* (or *guide function*). The cost function guides the search for the identification of the custom instructions which satisfy specific metrics (or constraints). The main metrics are listed here.

(1) *Number of inputs and outputs*. The size of a custom instruction can be limited by imposing limitations on the total number of inputs and/or outputs. This constraint is generally architecture dependent;
(2) *Area*. Depending on the architecture and on the implementation choices, each operation requires a certain amount of area when implemented in hardware. The cost function considers the area of a cluster as the sum of the operations included

in the cluster. When hardware resources are limited, the cost function continues or stops clustering based on the available hardware resources;

(3) *Power or energy consumption*. Power consumption is an important parameter for the design of efficient custom instructions. Based on the power consumption, the cost function can include or exclude a node from the custom instruction. One of the large power consumers is the memory system. For this reason, many time limitations to the number of memory accesses are introduced to limit the total power consumption of the custom instruction.

Additionally, the cost function can take into consideration.

(1) *Latency*. A custom instruction speeds up the execution of an application if, when moved to hardware, it reduces the total latency. The combination of different operations, as described in Section 3.1, can lead to fewer cycles to execute the operations in conjunction than they do individually;

(2) *Instruction scheduling*. "If all inputs of an instruction are supposed to be available at issue time and all results are produced at the end of the instruction execution" [Ienne and Leupers 2006], it is required that a feasible scheduling exists for the custom operation when it is fused into a single instruction that is atomically executed in hardware. This constraint is usually identified with the convexity of the instruction, a topic that is explained in more detail in the next sections.

Additional metrics can be introduced to guide the generation of custom instructions. The five aforementioned metrics are general and common to the majority of the approaches for custom instruction generation. Additional specific constraints related to the targeted architecture can also be considered. An exhaustive outline of different metrics used for the generation of custom instruction is presented in Holmer [1993, Chapter 4].

4.3. Instruction Generation

The analysis of the application for the generation of custom instructions is a *design space exploration* which aims at identifying instructions that can be selected for hardware implementation. We can detect two problems involved in instruction generation: the complexity of the exploration and the shape of the graph.

The Complexity of the Exploration. Given a graph that represents an application, in the most general case, each node of the graph can either be included or excluded from a candidate instruction. This means that there is an exponential number of potential candidates which turns into an exponential complexity of the design space exploration. The cost function, taking into consideration different constraints, reduces the number of candidates to a limited number. Several techniques have been proposed to handle the high computational complexity of the exploration. This can mainly be tackled in two ways: (1) by reducing the design space to explore (for example, by either using heuristics instead of exact algorithms or by limiting the size of the problem) or (2) by introducing specific constraints.

Many efficient heuristics have been proposed with very good runtimes when compared to exact solutions. The use of heuristics, even though it can efficiently reduce the design space explored, also turns into the generation of nonoptimal or even feasible solutions. Heuristics are often used with no theoretical guarantee.

An alternative way is to limit the size of the problem. For example, the approach presented in Atasu et al. [2003a] generates optimal sets of custom instructions. Even though the approach still has a worst exponential case runtime, for graphs of limited size, the solution is provided in a timely manner.

The introduction of additional constraints can reduce the number of candidates for hardware implementation, but has the drawback that every time a node is evaluated for inclusion or exclusion from a candidate instruction, all constraints have to be verified. Therefore, a reduction of candidates turns into a growth of the computational time due to the multiple analyses.

A way to optimally solve covering problems is by using a *branch-and-bound* approach. This approach starts with a search space potentially exponential in size, and reducing step-by-step the search space. The essence of this approach may be summarized in this way: if it is possible to show at any node in the total enumeration that the optimal solution cannot occur in any of its descendants, there is no need to consider those descendant nodes. Then, the search can be pruned at that node and the more we prune in the search space the more computationally manageable the problem becomes. A limitation on the analysis of unsuccessful branches relies on two aspects [Coudert 1996; Coudert and Madre 1995]: effective bounds and pruning techniques. Their combination can significantly improve the efficiency of the covering technique used to identify the candidate instructions for hardware implementation.

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. Basically a dynamic programming approach solves a multivariable problem by solving a series of single-variable problems. A drawback of dynamic programming is that it can only operate on tree-shaped graphs. Thus, the non-tree-shaped graph has to be decomposed into sets of disjoint trees. Other covering approaches, like Arnold and Corporaal [2001], use methods based on dynamic programming modified to deal with non-tree-shaped graphs.

The Shape of the Graph. The subject graph, the directed graph representing the given application, can be an acyclic or a cyclic graph. Usually, acyclic graphs are considered during the analysis. This follows from the fact that acyclic graphs can be easily sorted, for example, by a topological ordering, whereas cyclic graphs cannot. Therefore, for cyclic graphs, the issue of defining a one-to-one order of the nodes is added to the problem. Additionally, a cyclic graph can be transformed into an acyclic one if, for example, the cycles are unrolled.

Alternatively for dealing with cyclic graphs, one can consider the complete loops as single nodes in the graph. In this way, the graph can be topologically sorted but it presents two drawbacks: first, the number of custom instructions which is possible to generate is drastically reduced[6]. Second, it is difficult for different applications to share the same loops. This means that the custom instructions generated will speed up the execution of the given application and will hardly be used to speed up the execution of other applications.

Given a subject graph, the custom instructions can be generated following different criteria. When the generation is concluded, a subset of instructions, which maximizes the performance gain, is usually selected based on the available hardware resources. In the next sections, we present an overview of the different types of custom instructions. After that, Section 4.4 continues the analysis of the problem describing the different methods used to select which instructions are the most suitable to be implemented in hardware within the set of custom instructions generated.

[6]As mentioned in the previous section, the number of potential instructions is exponential in the number of nodes of the graph under analysis. By considering complete loops as single nodes, the total number of nodes in the graph is reduced, which in turn reduces the number of potential instructions.

4.3.1. Connected Instructions vs. Disconnected Instructions. The custom instructions can make use of the parallelism provided by the hardware implementation. This can be realized by looking for instructions which perform parallel independent operations at the same time. As previously described, in general, when n operations are performed in parallel, the total execution time is the maximum of the execution times of the considered operations. This means that a considerable speedup can be gained by identifying disconnected operations which can be clustered together in a custom instruction [Atasu et al. 2003a; Galuzzi et al. 2006; Yu and Mitra 2007]. Even though disconnected instructions can provide a high speedup, the majority of the authors look only for connected instructions [Arnold and Corporaal 2001; Baleani et al. 2002; Clark et al. 2003; Cong et al. 2004; Pozzi et al. 2001, 2002; Yu and Mitra 2004] due to the lower computational complexity of the algorithms. In literature only three works exhaustively enumerate all feasible (see the next paragraph) connected and disconnected subgraphs of a given data-flow graph: Yu and Mitra [2004, 2007] list all feasible connected and disconnected patterns, respectively, while Pozzi et al. [2006b] generate both connected and disconnected patterns.

4.3.2. Convexity and Schedulable Instructions. When a cluster of operations is fused into a single custom instruction that is atomically executed in hardware, the instruction has to be functionally executable. For example, in Figure 2(b), let G^* be the subgraph consisting of nodes $MM0$ and $MM1$. If G^* is fused into a single instruction, assuming that all inputs are available at issue time and all results are produced at the end of the instruction execution, there exists no feasible scheduling for G^*. This basically means that there exists a path between the nodes of G^* which includes nodes not belonging to G^* ($MM6$, in this case). The convexity of a graph is the property that guarantees that this eventuality does not occur. In this way it is possible to guarantee a feasible scheduling of the new instructions. Many works in literature generate convex custom instructions. Examples can be found in Atasu et al. [2003a], Yu and Mitra [2007], Gutin et al. [2007], and Zhao et al. [2008].

4.3.3. Single-Output Instructions vs. Multiple-Output Instructions. Depending on the target architecture, limitations on the maximum number of inputs and/or outputs can be introduced during the generation of the custom instructions. This is mainly due to the length of the instruction encoding and/or the number of ports in the register file [Yu and Mitra 2007]. 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). Accordingly, there are two types of algorithms for the identification of custom instructions: algorithms for the generation of MISO instructions and algorithms for the generation of MIMO instructions.

Multiple-Input Single-Output (MISO). "A single-output constraint allows for simplifying the architecture design by considering only one write port and it allows for avoiding conflicts in writing" [Pozzi 2000]. A representative example for the generation of single-output instructions is introduced in Alippi et al. [1999] and Pozzi et al. [2001] which address the generation of MISO instructions of maximal size, called MAXMISOs. Figure 2(a) shows an example of an application partitioned in MAXMISOs. The proposed algorithm exhaustively enumerates all MAXMISOs with a computational complexity linear with the number of processed elements. Access to memory, that is, load/store instructions, is not considered.

The approach presented in Cong et al. [2004] targets the generation of MISO instructions. As previously described, in the most general case, each node of the graph can either be included or excluded from a candidate instruction turning into an

exponential number of potential candidates. As a consequence, a heuristic limiting the total number of input operands and area constraints is introduced to allow an efficient generation. The difference between the complexities of the two approaches in Cong et al. [2004] and Alippi et al. [1999] is represented by the properties of MISOs and MAXMISOs: while the enumeration of the first is similar to the subgraph enumeration problem, the intersection of MAXMISOs is empty and then MAXMISOs can be enumerated with linear complexity. A different approach is presented in Galuzzi et al. [2007b] where, with an iterative application of the MAXMISO clustering presented in Alippi et al. [1999], MISO instructions with variable number of inputs are generated with a heuristic of linear complexity in the number of processed elements. The approaches in Cong et al. [2004] and Galuzzi et al. [2007b] can be very effective when tight limitations on the total number of inputs are applied. An other approach presented in Lee et al. [2003b] groups the operations of a given loop body into single output clusters for an efficient implementation of the operations onto an ALU array. In Peymandoust et al. [2003], the authors propose polynomial manipulation-based techniques for the automatic extension of a given instruction-set with complex single-output instructions.

Multiple-Input Multiple-Output (MIMO). Multiple-output instructions can provide significant performance improvements compared with single-output instructions, as shown in Ienne and Leupers [2006, Chaper 7]. There exists an exponential potential number of candidate MIMO clusters. A number of approaches proposed in literature identify optimal solutions or use efficient heuristics to reduce the complexity of the solution generated. In Verma et al. [2002] and Atasu et al. [2003a] the identification algorithm detects an optimal number of convex MIMO subgraphs based on input/output constraints, area, and convexity, but the computational complexity is exponential and it has problems of scalability. A similar approach described in Yu and Mitra [2004] proposes the enumeration of all feasible instructions (MISO and MIMO) based on the number of inputs, outputs, area, and convexity. The selection problem is not addressed. Contrary to Atasu et al. [2003a] which has scalability issues if the data-flow graph is very large or the micro-architectural constraints are too fine, the approach presented in Yu and Mitra [2004] 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 Yu and Mitra [2007], where the authors address the exhaustive enumeration of connected and disconnected clusters based on the number of inputs, outputs, and convexity. In Biswas et al. [2004b], the authors present an approach similar to the one described in Atasu et al. [2003a] with the inclusion of memory accesses in the generation of custom instructions.

In Atasu et al. [2008] a similar problem is addressed but the authors enumerate only maximal convex subgraphs within an application. Additionally, they do not impose limitations on the number of input and output operands for the custom instructions. A similar target is presented in Pothineni et al. [2007] and Verma et al. [2007] where the authors propose similar methods to enumerate maximal convex subgraphs.

In Atasu et al. [2005] the authors target the identification of convex clusters of operations under input and output constraints. The clusters are identified with a Integer Linear Programming (ILP)-based methodology. In Galuzzi et al. [2006], the authors address the generation of convex MIMO operations in a manner similar to Atasu et al. [2005], although the identification of the new instructions is rather different and based on the MAXMISO clustering proposed in Alippi et al. [1999]. While Atasu et al. [2005] iteratively solve ILP problems for each basic block, Galuzzi et al. [2006] have one global ILP problem for the entire procedure. Additionally, the convexity is

addressed differently: in Atasu et al. [2005] the convexity is verified at each iteration, while in Galuzzi et al. [2006] the convexity is guaranteed by construction.

In Arnold [2001], the author proposes a method to generate instructions with an arbitrary number of inputs and outputs for VLIW processors. This approach is based on dynamic programming and removes the requirement of a tree-shaped graph during the generation of generally small clusters of instructions. In Choi et al. [1999], the authors observe that the number of operations per cluster is typically small and propose a clustering method which generates custom instructions limited to pair of instructions without constraining inputs and outputs. In Baleani et al. [2002], the authors propose a greedy algorithm, called *clubbing*, which identifies custom instructions with limited inputs and outputs ($3-2$ in the examples). In Biswas et al. [2004a, 2005], the authors use the Kernighan-Lin ($K-L$) min-cut algorithm (see [Lin and Kernighan 1973]), a well-known graph partitioning heuristic, to automatically generate custom instructions again imposing inputs and outputs constraints.

In Seto and Fujita [2008], "an approach which generates custom instructions with any numbers of inputs and outputs is presented. Unlike other approaches that generate a custom instruction from each subgraph, the authors generate a sequence of multiple custom instructions with high-level synthesis techniques and use resource sharing among the custom instructions in order to reduce the area usage" [Seto and Fujita 2008].

As mentioned at the beginning of this section, limitations on inputs and outputs are architecture dependent. Although a considerable speedup can be achieved by increasing the total number of inputs and/or outputs for the custom instructions, "additional ports result in increased register file size, power consumption and cycle time" [Atasu 2007]. To overcome the limitations on the operands, a number of techniques has been proposed which allow for relaxation of the limitations.

In Pozzi and Ienne [2005], the authors propose a solution to the limitation of actual register-file ports by serializing the register-file accesses and therefore addressing multicycle read and write. The technique combines register file access serialization with pipelining in order to obtain the best global solution. In Jayaseelan et al. [2006] the authors show that, by forwarding paths of the base processor, up to two additional inputs per custom instruction can be considered without incurring additional costs.

In the following section, the main approaches for the selection of a subset of candidates for hardware implementation are presented.

4.4. Instruction Selection

The main goal of instruction selection is the identification of a subset of custom instructions suitable to be implemented in hardware, based on the available hardware resources. The selection of the instruction can be optimal [Alippi et al. 2001; Atasu et al. 2003a, 2003b, 2005; Sang et al. 2005] or nonoptimal (heuristic) [Brisk et al. 2004; Cheung et al. 2003b; Pozzi et al. 2006a; Sun et al. 2003] depending on the used approach.

One of the main problems during the selection of the best candidates is the covering of the design space: optimal algorithms can be too expensive in terms of computational cost. Heuristics alone cannot guarantee either optimality or feasibility of the solution. The selection can follow different policies. The elements can be selected attempting to minimize the number of distinct templates that are used [Aho et al. 1989; Choi et al. 1999; Guo et al. 2003; Kavvadias and Nikolaidis 2005, 2006; Lam and Srikanthan 2009; Lam et al. 2006], attempting to maximize the number of instances of each template [Scharwaechter et al. 2007], or to minimize the number of nodes left uncovered in the graph [Liao et al. 1995, 1998], or in such a way that the longest path through the

graph should have minimal delay. Other approaches select instructions based on regularity or frequency of execution, that is, the repeated occurrence of certain templates [Arnold and Corporaal 1999; Brisk et al. 2002; Janssen et al. 1996; Peymandoust et al. 2003; Sreenivasa Rao and Kurdahi 1993b,a], or resource sharing [Huang and Malik 2001; Moreano et al. 2002], or the occurrence of specific nodes [Clark et al. 2003; Kastner et al. 2002; Sun et al. 2002; Wolinski and Kuchcinski 2007, 2008] or hardware reuse through similarity of the clusters that are implemented [Alomary et al. 1993; Geurts 1995, 1997]. Other appraches try to minimize the power dissipation or consumption [Cheung et al. 2005; Lee et al. 2003a; Strozek and Brooks 2006] or the code size [Biswas and Dutt 2003a, 2003b; 2005] or the memory accesses [Biswas et al. 2006].

One way to address instruction selection is by using Integer Linear Programming (ILP) and more generally Linear Programming (LP) in combination with an efficient LP solver. Linear programming addresses the problem of maximizing or minimizing a linear function over a convex polyhedron specified by linear and nonnegativity constraints. In essence, each instruction is associated with a variable which can have an integer value (ILP), noninteger value (LP), or a boolean value ($0-1$ LP). The instructions, and then the variables, have to satisfy a certain number of constraints which are expressed with a system of linear inequalities. The optimal solution is the one that maximizes or minimizes the so-called objective function. Examples of instruction selection by using ILP and LP can be seen in Atasu et al. [2005], Atasu et al. [2007], Galuzzi et al. [2006], Imai et al. [1992], Lee et al. [2002, 2007], Leupers et al. [2006], Niemann and Marwedel [1996], Yu and Mitra [2005], and Wong et al. [2007].

One way to optimally solve covering problems is by using dynamic programming or branch-and-bound methods. Exact solutions are proposed in Grasselli and Luccio [1965] and Brayton and Somenzi [1989]. A method is efficient when it prevents the exploration of unsuccessful branches at earlier stages of the search. This relies on efficient bounding techniques [Coudert and Madre 1995; Coudert 1996; Liao and Devadas 1997; Li et al. 2005]. In Liao and Devadas [1997] it has been shown that Linear-Programming Relaxation (LPR)[7] can be used to obtain tighter lower bounds than previous approaches [Coudert and Madre 1995; Coudert 1996]. "Their techniques, derived from computing a maximal independent set, are based on the idea of solving the LPR-equivalent of the ILP form of the binate-covering problem for lower-bounding purposes, and of applying traditional covering-matrix reduction techniques during branch-and-bound. These new lower bounds require more computation but they allow for early termination of suboptimal branches" [Liao and Devadas 1997]. In Bình et al. [1996a, 1996b] the authors propose a branch-and-bound-based algorithm to minimize the area cost under constraints of schedule length and power consumption.

An additional problem during the selection of the instructions is template overlapping [Cong et al. 2004; Aletà et al. 2004]. For example, in Figure 2(b), the two subgraphs containing nodes $MM1$ and $MM6$ and nodes $MM1$ and $MM2$, respectively, overlap at node $MM1$. This is a typical problem when a set of predefined templates is used. There are two ways of selecting instructions when we deal with overlapping templates: either by selecting a subset of nonoverlapping templates that maximizes performance or by first replicating the common nodes between the overlapping templates and then selecting a subset of templates that maximize performance. In this way, at an additional cost of the replicated nodes, performance can be increased through a greater number of candidates suitable for hardware implementation. In the general case, after the generation of a custom instruction, the nodes belonging to the cluster are removed

[7]The Linear-Programming Relaxation (LPR) of an ILP is the linear program obtained by disregarding the integrality constraints.

from the nodes subject to further analysis. Therefore, two disjoint templates do not overlap [Baleani et al. 2002; Galuzzi et al. 2007a].

Instruction selection, similarly to instruction generation, makes use of a cost function to guide the selection. Many approaches combine instruction generation and selection and use a unique cost function to generate and select custom instructions. As mentioned in Section 4.2, the cost function considers a certain number of metrics (constraints) to guide the generation. When generation and selection are considered independently, it is possible to split the constraints between the two functions and reduce the complexity of the generation/selection process of the custom instruction. For example, Atasu et al. [2003a] describe 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. Each node considered for inclusion or exclusion in/from a cluster needs to satisfy constraints on the total number of inputs, outputs, and convexity. Testing the convexity of a cluster involves multiple analyses of the nodes in the cluster to verify that for each pair of nodes in the cluster there is no path connecting the nodes that involves nodes not belonging to the cluster itself. If the output limit is set to one, each time a node is evaluated for inclusion or exclusion in a cluster, the convexity constraint is automatically satisfied by the single output of the cluster. This follows by the single-output property: if the cluster has a single output, for each pair of nodes in the cluster, all the paths connecting the two nodes belong to the cluster. As a consequence, by reducing the number of constraints to test from 3 to 2, a considerable amount of the execution time can be saved.

5. CUSTOM INSTRUCTION INTEGRATION

In this section, we present an overview of the main approaches which integrate custom instructions. There are several ways in which a processor and a reconfigurable logic can be coupled. "The tighter the integration, the more frequently the custom logic can be used within an application. This is mainly due to lower communication overhead" [Compton and Hauck 2002].

The main methods to couple a processor and a reconfigurable logic are [Atasu 2007; Pozzi 2000]:

— functional units,
— coprocessors,
— attached or external processing units,
— embedded cores.

In the following sections, these approaches are analyzed in more detail. Additionally, a representative overview of the main approaches proposed in the last years is presented.

5.1. Functional Units

In this scenario, processor and reconfigurable logic are tightly coupled. The custom instructions are integrated into the host processor data path in parallel to the basic execution unit. In this way, "it is possible to make use of the traditional programming environment extended with the custom instructions" [Compton and Hauck 2002].

Representative examples are the *OneChip* architecture [Wittig 1995; Wittig and Chow 1996] which combines an MIPS-like host processor with reconfigurable logic resources to accelerate speed-critical applications. The reconfigurable functional unit works in parallel with the normal units and no limitations are imposed on the kind of functions implemented in the reconfigurable logic. The architecture allows dynamic

scheduling and partial dynamic reconfiguration. Additionally, the functions to be implemented in hardware are manually selected.

An other example is the *Chimaera* architecture [Hauck et al. 1997, 2004; Ye et al. 2000], in which a reconfigurable functional unit works in parallel with the normal execution unit, has access to shadow registers (registers which duplicate a subset of the registers of the base processor in the custom logic area), and is mapped onto an on-chip FPGA which implements different multioperand functions utilizing partial runtime reconfiguration to reduce reconfiguration time.

The *PRogrammable Instruction-Set Computer* (PRISC) architecture [Razdan and Smith 1994] integrates combinational reconfigurable logic as reconfigurable functional units with limited inputs and outputs. The system automatically detects sequences of logic operations which can be implemented as single new instructions. The search is limited to sequences of operations with two inputs and one output which are executed in a single cycle and the reconfigurable functional units are partially dynamically reconfigurable.

In Vassiliadis et al. [2006, 2007] an embedded single issue RISC processor tightly coupled with a coarse-grain Reconfigurable Functional Unit (RFU) is presented.

> "Two architectural enhancements are presented: partial predicated execution, used to remove control dependencies and expose larger clusters of operations as candidates for execution in the RFU, and virtual opcode, used to alleviate the opcode space explosion and increase the number of candidate for execution in the RFU. The main characteristic of this architecture is that the communication overhead between the control unit and the datapath is eliminated. The elimination is achieved by an efficient integration of the reconfigurable functional unit, which optimally exploits the processor's pipeline structure. The reconfigurable functional unit executes a set of instructions with no data dependencies in parallel, increasing in this way the overall speed up"

An other architecture, *Processor Reconfiguration through Instruction-Set Metamorphosis* ($PRISM-I$) is presented in Athanas and Silverman [1993]. In this system, entire functions inside the application can be mapped onto reconfigurable hardware. Special instructions, embedded in the object code, control the interaction between what is executed in hardware and what is executed in software. The system starting from generic *C* code generates $FPGA$ configurations in a semiautomatic process. Due to the limitations of the $FPGA$ technology at that time, processor and $FPGA$ were located into separate chips making the interface between them relatively slow. This, together with an initialization overhead for the reconfigurable component, considerably limited the class of applications addressable by the system, which moreover is more suitable for a coarse-grained customization.

5.2. Coprocessors

In this scenario, the custom instructions are integrated as a coprocessor "which directly access to the main processor through a local bus or dedicated pins of the main processor" [Atasu 2007]. Coprocessors are, in general, able to perform many computations without constantly communicating with the main processor: the processor sends the data directly to the coprocessor or it provides information on where the data are located in the memory. Usually processor and coprocessor can work simultaneously. Additionally, the low-latency, high-bandwidth connection between processor and coprocessor allows accessing the custom logic more frequently. In literature, many

approaches follow this type of integration. Coprocessors can be divided into fine- and coarse-grain category [Atasu 2007].

The *Garp* architecture [Hauser and Wawrzynek 1997] belongs to the first category and is used to accelerate specific loops or subroutines. This system integrates on the same die a standard MIPS-II-like host processor with a reconfigurable coprocessor. When a reconfigurable function is called, the main processor activates the coprocessor to execute the operation. The coprocessor accesses both the processor main memory and cache memory and does not require processor intervention during the execution of the operations. For this reason, the processor is suspended when the coprocessor is activated. The reconfigurable array can be partially reconfigured as it is organized in rows.

The *Molen* architecture presented in Vassiliadis et al. [2001, 2004] is composed by a GPP, the core processor, which controls the execution and the (re)configuration of a reconfigurable coprocessor, tuning the latter for specific applications by implementing application-specific instructions. The instructions are decoded by an arbiter determining which unit is targeted. The instructions are partitioned in basic instructions executed by the core processor, and application-specific instructions implemented on the reconfigurable processor. The communication overhead is comparable to Athanas and Silverman [1993] but the configurations are defined as part of the processor design itself instead of being determined by compilation. Moreover, Molen has a high degree of freedom in the definition of the programmable array structure and can exploit commercial FPGAs, taking advantage of the technology development in this field while maintaining the basic architectural framework unchanged.

An other architecture is presented in Iseli and Sanchez [1995] and Iseli [1996], *Spyder*, a coprocessor with several reconfigurable execution units working in parallel, based on a VLIW processor architecture. Other examples are the PRISM-II [Wazlowski et al. 1993] and the NAPA architecture [Rupp et al. 1998].

The *REconfigurable Multimedia ARray Coprocessor* (REMARC) [Miyamori and Olukotun 1998] is part of the coarse-grain category. A reconfigurable coprocessor that consists of a global control unit and 64 programmable logic blocks called nano-processors is designed to accelerate multimedia applications, such as video compression, decompression, and image processing. Each 16-bit unit has an entry instruction RAM, ALUs, data RAM, instruction, and several other registers. The reconfigurable array operates on the coprocessor data registers and a control unit transfers data between these registers and the processor. The architecture allows dynamic reconfiguration.

In Lu et al. [1999], the authors present MORPHOSYS, a system which integrates a reconfigurable array of processing cells, a MIPS-like host processor, and an efficient memory interface unit designed to speed up video compression, data encryption, and target recognition.

The ADRES architecture [Mei et al. 2003] tightly couples a VLIW processor with a coarse-grain reconfigurable matrix into one single architecture. Processor and reconfigurable matrix cannot execute concurrently and this allows sharing of resources between them. The reconfigurable cells composing the reconfigurable matrix include ALU-like configurable functional units and local register files. Other examples are the *Reconfigurable Pipelined Datapath* (RaPiD) architecture [Ebeling et al. 1996] which aims at speeding up highly regular, computation-intensive tasks using deep pipelines, and the *Pleiades* Architecture [Rabaey 1997] which is designed for speeding up communication, speech coding, and video coding.

Coarse-grain reconfigurable logic usually has the advantage of providing faster reconfiguration times, fewer configuration bits, and faster clock speed in the reconfigurable logic. Coarse-grain configurable architectures are more suitable for

data-intensive applications in the multimedia and communication domains, while fine-grain architectures are better for bit-level computation [Huang et al. 2004].

Commercial products include, for example, *Cascade* by Criticalblue[8], an automated coprocessor synthesis solution used to accelerate the execution of compiled binary executable software code offloaded from the Central Processing Unit (CPU) by creating a loosely coupled programmable coprocessor.

5.3. Attached or External Processing Units

When custom instructions are integrated as attached or external processing units, communications between host processor and processing units is achieved through a general-purpose bus interface. In this case,

> "performance is affected by the high communication overhead due to the bandwidth and latency limitations of the general purpose bus. For this reason, this type of organization is used for applications which have a high computation to communication ratio, such as stream-based applications" [Atasu 2007]. This means that "a significant amount of processing can be done by the processing unit without the intervention of the main processor" [Compton and Hauck 2002].

An example is *PipeRench* [Goldstein et al. 1999], a reconfigurable fabric used as an attached processor designed to accelerate pipelined applications. The architecture, partially dynamically reconfigurable, consists of an interconnected network of processing elements organized in pipeline stages. Each processing element consists of registers and ALUs. An intermediate language is used to generate the fabric configurations.

The SONIC architecture [Haynes et al. 1999, 2000] consists of a set of processing elements, called Plug-In Processing Elements (PIPEs), interconnected by a bus. Each PIPE contains a reconfigurable processor, a scalable router that also formats video data, and a frame-buffer memory. The architecture is designed to exploit parallelism in video image processing algorithms.

Splash [Gokhale et al. 1991] and *Splash*2 [Buell et al. 1996] are attached processors using FPGAs as their processing elements (32 and 17, respectively). The FPGAs, each coupled with a RAM, are connected as a linear array through a crossbar switch that introduces larger flexibility than that of a simple linear array.

5.4. Embedded Cores

In this case, the processor is embedded in the reconfigurable hardware [Atasu 2007; Todman et al. 2005]. The processor is embedded either as a hard core or as a soft core implemented on resources of the reconfigurable hardware itself which can be used to extend the core with specialized instructions. In the former category, there are commercial products as the Altera's Excalibur and the Xilinx Virtex II which embed an ARM922T core and a PowerPC 405 core, respectively, and the Atmel FPSLIC which embeds a 20 MIPS AVR 8-bit RISC core. The Altera Nios and Nios II and the Xilinx MicroBlaze and PicoBlaze belong to the latter category. When hard cores are compared with soft cores, they present advantages and drawbacks. First, hard cores are more area efficient, leaving additional logic for other uses, and second they are usually faster. Third, hard cores are less flexible and fourth, hard cores do not allow for an arbitrary choice of the number of cores.

Many other architectures have been proposed and a number of surveys exist. Exhaustive reviews are presented in Barat and Lauwereins [2000], Barat et al. [2002],

[8]http://www.criticalblue.com/criticalblue_products/cascade.shtml

Compton and Hauck [2002], Hartenstein [2001a, 2001b], Radunovic and Milutinovic [1998], and Vassiliadis and Soudris [2007]. We refer the interested reader to the aforementioned surveys, where the classification of the architectures is also presented in terms of granularity of the reconfigurable logic blocks and in terms of different coupling approaches.

6. CONCLUSIONS

In this article, we presented an overview of the issues involved in the customization of an instruction-set by means of a set of specialized instructions for a given application or domain of applications. The problems, analyzed in detail, consider different types of customizations and instructions and both instruction generation and selection.

The problems involved, as described in the article, are computational complex problems. Hardware/software partitioning, equivalent to instruction-set customization under certain assumptions, is proven NP-hard in the general case [Arató et al. 2003]. Optimal solutions have been proposed by many authors and a plethora of efficient heuristics have been proposed to find near-optimal solutions when the computational complexity of the problem becomes unmanageable and exact solutions cannot be found in a timely manner.

As things stand, one of the major issues in the generation of custom instructions is represented by the degree of human effort required to identify and implement the instruction-set extensions. As described in the article, human ingenuity in manual creation of custom capabilities creates high-quality results. In spite of that, the complexity of the problem as well as the time-to-market requirements led researchers to look for automatic or partially automatic methods for identifying custom instructions. As a result, quality results are produced through a balance of human intervention and automatic methods in the generation of the instructions. However, future approaches will substantially minimize the amount of human effort due to the increasing complexity of the designs.

An additional limitation in the current state-of-the-art in instruction-set extension is the limited number of input and output operands of the custom instructions. This limitation, which is architecture dependent, has been relaxed in the last years by using methods proposed to overcome severe limitations on the number of operands, as mentioned in Section 4.3.3. As a result, new methodologies, by making use of these techniques, will be able to generate and select many more instructions, which in turn will allow a better customization of the instruction-set.

In the last years, many low-power and power-aware architectures have been proposed. While the former minimize power consumption while satisfying performance constraints, the latter maximize performance parameters while satisfying power constraints. The current state-of-the-art in instruction-set customization shows that very few methods exist which take into consideration power issues during the generation of custom instruction. As power consumption/reduction/optimization have become one of the main topics of research, we will see more and more methods appearing for generating custom low-power or power-aware instructions which will be trade off between their size (by limiting the size of the instruction, the power consumption is limited as well) and their frequency of execution (a limited number of executions reduces the power consumption).

One of the main issues in instruction-set customization is also represented by the degree of specialization of the custom instructions. If the instructions are too specialized, instruction reuse becomes hard. This is experienced because it is uncommon that applications from different domains perform the same complex calculations. Vice versa, if a custom instruction is used by many different applications, the requirement to speed up different applications from different domains turns into the generation

of custom instructions of limited size. Therefore, the custom instructions are limited to few operations per instruction, which, in turn, can reduce performance. This is a practical issue which is common to every approach and which will be always present: a method for the generation of custom instructions will always be a trade-off between the level of specialization of the instructions and the speed up that they can provide.

Finally, in the last years, multicore systems have become ubiquitous. Many architectures integrate two or more cores in the same hardware to increase performance of execution exploiting the available parallelism. Multicore architectures can provide high performance, run at lower clock speed than single-core architectures, and can reduce power consumption. Multicore systems can be homogeneous or heterogeneous. The former implement identical copies of the same core: same frequencies, cache sizes, functions, etc. Examples are the Intel Core 2 Duo and the Advanced Micro Devices Athlon 64 $X2$. Heterogeneous systems integrate different cores which can have different functions, frequencies, memory models, etc. Examples are the CELL Processor used in Sony's PlayStation 3 game console and the Tilera TILE64. Existing methods for the customization of an instruction-set typically consider a single core and a single instruction-set. Nevertheless, in the future, we envision that instruction-set customization will take advantage of the multicore architecture by extending each core with a set of specialized instructions. In this way, a considerable amount of applications from different domains such as graphics, audio, cryptography, communications, mathematics, or biology and more, will be efficiently executed on the architecture.

ACKNOWLEDGMENTS

The authors would like to thank Niki Frantzeskaki, Sebastian Isaza, Daniele Ludovici, Dimitris Theodoropoulos, and Christos Strydis for their help.

REFERENCES

AHO, A. V., GANAPATHI, M., AND TJIANG, S. W. K. 1989. Code generation using tree matching and dynamic programming. *ACM Trans. Programm. Lang. Syst. 11,* 4, 491–516.

ALETÀ, A., CODINA, J. M., GONZÁLEZ, A., AND KAELI, D. 2004. Removing communications in clustered microarchitectures through instruction replication. *ACM Trans. Archit. Code Optimiz. 1,* 2, 127–151.

ALIPPI, C., FORNACIARI, W., POZZI, L., AND SAMI, M. March 1999. A dag-based design approach for reconfigurable vliw processors. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'99)*. 778–779.

ALIPPI, C., FORNACIARI, W., POZZI, L., AND SAMI, M. 2001. Determining the optimum extended instruction-set architecture for application specific reconfigurable vliw cpus. In *Proceedings of the 12th International Workshop on Rapid System Prototyping (RSP'01)*. 50–56.

ALOMARY, A., NAKATA, T., HONMA, Y., IMAI, M., AND HIKICHI, N. 1993. An asip instruction set optimization algorithm with functional module sharing constraint. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'93)*. 526–532.

ALOMARY, A. Y. 1996. A hardware/software codesign partitioner for asip design. In *Proceedings of the 3rd IEEE International Conference on Electronics, Circuits, and Systems (ICECS'96)*. 251–254.

ARATÓ, P., JUHÁSZ, S., ÁDÁM MANN, Z., ORBÁN, A., AND PAPP, D. 2003. Hardware-Software partitioning in embedded system design. In *Proceedings of the IEEE International Symposium on Intelligent Signal Processing (WISP'03)*. 197–202.

ARNOLD, M. 2001. Instruction set extension for embedded processors. Ph.D. thesis, University of Delft, The Netherlands.

ARNOLD, M. AND CORPORAAL, H. 1999. Automatic detection of recurring operation patterns. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES'99)*. 22–26.

ARNOLD, M. AND CORPORAAL, H. 2001. Designing domain-specific processors. In *Proceedings of the 9th International Symposium on Hardware/Software Codesign (CODES'01)*. 61–66.

ATASU, K. 2007. Hardware/software partitioning for custom instruction processors. Ph.D. thesis, Boğaziçi University, Turkey. December.

ATASU, K., POZZI, L., AND IENNE, P. 2003a. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 40th Conference on Design Automation (DAC'03)*. 256–261.

ATASU, K., POZZI, L., AND IENNE, P. 2003b. Automatic application-specific instruction-set extensions under microarchitectural constraints. *Int. J. Parall. Programm. 31,* 6, Special issue: Workshop on application specific processors (WASP), 411–428.

ATASU, K., DÜNDAR, G., AND ÖZTURAN, C. 2005. An integer linear programming approach for identifying instruction-set extensions. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'05)*. 172–177.

ATASU, K., DIMOND, R. G., MENCER, O., LUK, W., ÖZTURAN, C., AND DÜNDAR, G. 2007. Optimizing instruction-set extensible processors under data bandwidth constraints. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'07)*. 588–593.

ATASU, K., MENCER, O., LUK, W., ÖZTURAN, C., AND DÜNDAR, G. 2008. Fast custom instruction identification by convex subgraph enumeration. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP'08)*. 1–6.

ATHANAS, P. M. AND SILVERMAN, H. F. 1993. Processor reconfiguration through instruction-set metamorphosis. *Comput. 26,* 3, 11–18.

BALEANI, M., GENNARI, F., JIANG, Y., PATEL, Y., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. 2002. Hw/sw partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES'02)*. 151–156.

BARAT, F. AND LAUWEREINS, R. 2000. Reconfigurable instruction set processors: A survey. In *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP'00)*. IEEE Computer Society, 168.

BARAT, F., LAUWEREINS, R., AND DECONINCK, G. 2002. Reconfigurable instruction set processors from a hardware/software perspective. *IEEE Trans. Softw. Engin. 28,* 9, 847–862.

BÌNH, N. N., IMAI, M., AND HIKICHI, N. 1995. A hardware/software partitioning algorithm for pipelined instruction set processor. In *Proceedings of the Conference on European Design Automation (EURO-DAC'95/EURO-VHDL'95)*. 176–181.

BÌNH, N. N., IMAI, M., AND SHIOMI, A. 1996a. A new hw/sw partitioning algorithm for synthesizing the highest performance pipelined asips with multiple identical fus. In *Proceedings of the Conference on European Design Automation (EURO-DAC'96/EURO-VHDL'96)*. 126–131.

BÌNH, N. N., IMAI, M., SHIOMI, A., AND HIKICHI, N. 1996b. A hardware/software partitioning algorithm for designing pipelined asips with least gate counts. In *Proceedings of the 33rd Annual Conference on Design Automation (DAC'96)*. 527–532.

BISWAS, P. AND DUTT, N. 2003a. Greedy and heuristic-based algorithms for synthesis of complex instructions in heterogeneous-connectivity-based DSPs. Tech. rep. 03-16, UCI-ISR.

BISWAS, P. AND DUTT, N. 2003b. Reducing code size for heterogeneous-connectivity-based vliw dsps through syntheis of instruction set extensions. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'03)*. 104–112.

BISWAS, P. AND DUTT, N. D. 2005. Code size reduction in heterogeneous-connectivity-based dsps using instruction set extensions. *IEEE Trans. Comput. 54,* 10, 1216–1226.

BISWAS, P., BANERJEE, S., DUTT, N., POZZI, L., AND IENNE, P. September 2004a. Fast automated generation of high-quality instruction set extensions for processor customization. In *Proceedings of the 3rd Workshop on Application Specific Processors (WASP'04)*.

BISWAS, P., CHOUDHARY, V., ATASU, K., POZZI, L., IENNE, P., AND DUTT, N. 2004b. Introduction of local memory elements in instruction set extensions. In *Proceedings of the 41st Annual Conference on Design Automation (DAC'04)*. 729–734.

BISWAS, P., BANERJEE, S., DUTT, N., POZZI, L., AND IENNE, P. 2005. Isegen: Generation of high-quality instruction set extensions by iterative improvement. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'05)*. 1246–1251.

BISWAS, P., DUTT, N., IENNE, P., AND POZZI, L. 2006. Automatic identification of application-specific functional units with architecturally visible storage. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'06)*. European Design and Automation Association, 212–217.

BOBDA, C. 2007. *Introduction to Reconfigurable Computing*. Springer.

BONZINI, P. AND POZZI, L. 2007a. Polynomial-Time subgraph enumeration for automated instruction set extension. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'07)*. 1331–1336.

BONZINI, P. AND POZZI, L. 2007b. A retargetable framework for automated discovery of custom instructions. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP07)*.

BORIN, E., KLEIN, F., MOREANO, N., AZEVEDO, R., AND ARAUJO, G. 2004. Fast instruction set customization. In *2nd Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia'04)*. 53–58.

BRAYTON, R. K. AND SOMENZI, F. 1989. Boolean relations and the incomplete specification of logic networks. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'89)*. 316–319.

BRISK, P., KAPLAN, A., KASTNER, R., AND SARRAFZADEH, M. 2002. Instruction generation and regularity extraction for reconfigurable processors. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Sfor Embedded Systems (CASES'02)*. 262–269.

BRISK, P., KAPLAN, A., AND SARRAFZADEH, M. 2004. Area-Efficient instruction set synthesis for reconfigurable system-on-chip designs. In *Proceedings of the 41st annual conference on Design automation (DAC'04)*. 395–400.

BUELL, D., KLEINFELDER, W., AND ARNOLD, J. 1996. *Splash 2: FPGAs in a Custom Computing Machine*.

CHEN, L. 1996. Graph isomorphism and identification matrices: Parallel algorithms. *IEEE Trans. Parall. Distrib. Syst. 7,* 3, 308–319.

CHEUNG, N., HENKEL, J., AND PARAMESWARAN, S. 2003a. Rapid configuration and instruction selection for an asip: A case study. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'03)*.

CHEUNG, N., PARAMESWARAN, S., AND HENKEL, J. 2003b. Inside: Instruction selection/identification and design exploration for extensible processors. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'03)*.

CHEUNG, N., PARAMESWARAN, S., AND HENKEL, J. 2005. Battery-Aware instruction generation for embedded processors. In *Proceedings of the Conference on Asia South Pacific Design Automation (ASP-DAC'05)*. 553–556.

CHOI, H., HWANG, S. H., KYUNG, C.-M., AND PARK, I.-C. 1998. Synthesis of application specific instructions for embedded dsp software. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'98)*. 665–671.

CHOI, H., KIM, J.-S., YOON, C.-W., PARK, I.-C., HWANG, S. H., AND KYUNG, C.-M. 1999. Synthesis of application specific instructions for embedded dsp software. *IEEE Trans. Comput. 48,* 6, 603–614.

CLARK, N. 2007. Customizing the computation capabilities of microprocessors. Ph.D. thesis, University of Michigan, Ann Arbor.

CLARK, N. T. AND ZHONG, H. 2005. Automated custom instruction generation for domain-specific processor acceleration. *IEEE Trans. Comput. 54,* 10, 1258–1270.

CLARK, N., TANG, W., AND MAHLKE, S. 2002. Automatically generating custom instruction set extensions. In *Proceedings of 1st Workshop on Application Specific Processors (WASP)*. 94–101.

CLARK, N., ZHONG, H., AND MAHLKE, S. 2003. Processor acceleration through automated instruction set customization. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO'36)*.

CLARK, N., KUDLUR, M., PARK, H., MAHLKE, S., AND FLAUTNER, K. 2004. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'37)*. 30–40.

CLARK, N., BLOME, J., CHU, M., MAHLKE, S., BILES, S., AND FLAUTNER, K. 2005. An architecture framework for transparent instruction set customization in embedded processors. *SIGARCH Comput. Archit. News 33,* 2, 272–283.

CLARK, N., HORMATI, A., MAHLKE, S., AND YEHIA, S. 2006. Scalable subgraph mapping for acyclic computation accelerators. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*. 147–157.

COMPTON, K. AND HAUCK, S. 2002. Reconfigurable computing: A survey of systems and software. *ACM Comput. Surv. 34,* 2, 171–210.

CONG, J., FAN, Y., HAN, G., AND ZHANG, Z. 2004. Application-specific instruction generation for configurable processor architectures. In *Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA'04)*. 183–189.

COUDERT, O. 1996. On solving covering problems. In *Proceedings of the 33rd Annual Conference on Design Automation (DAC'96)*. 197–202.

COUDERT, O. AND MADRE, J. C. 1995. New ideas for solving covering problems. In *Proceedings of the 32nd ACM/IEEE Conference on Design Automation (DAC'95)*. 641–646.

DE MICHELI, G. AND GUPTA, R. K. 1997. Hardware/software co-design. *Proc. IEEE 85,* 3, 349–365.

EBELING, C., CRONQUIST, D., AND FRANKLIN, P. 1996. Rapid - reconfigurable pipelined datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers (FPL'96)*. Springer, 126–135.

FARABOSCHI, P., BROWN, G., FISHER, J. A., DESOLI, G., AND HOMEWOOD, F. 2000. Lx: a technology platform for customizable vliw embedded processing. *ACM SIGARCH Comput. Archit. News 28,* 2, 203–213.

FORNACIARI, W., POZZI, L., AND SAMI, M. 1999. Processori riconfigurabili: unalternativa flessibile per i sistemi dedicati. *Alta Frequenza - Rivista di Elettronica*, 22–28.

FORTIN, S. 1996. The graph isomorphism problem. Tech. rep. TR 96-20, Department of Computing Science, University of Alberta, Canada.

GALUZZI, C., BERTELS, K., AND VASSILIADIS, S. 2007a. A linear complexity algorithm for the automatic generation of convex multiple input multiple output instructions. In *Proceedings of the 3rd International Workshop Reconfigurable Computing: Architectures, Tools and Applications (ARC'07)*, P. C. Diniz, E. Marques, K. Bertels, M. M. Fernandes, and J. M. P. Cardoso Eds., Lecture Notes in Computer Science, vol. 4419. Springer, 130–141.

GALUZZI, C., BERTELS, K., AND VASSILIADIS, S. 2007b. A linear complexity algorithm for the generation of multiple input single output instructions of variable size. In *Proceedings of the Embedded Computer Systems: Architectures, Modeling, and Simulation, 7th International Workshop (SAMOS'07)*, S. Vassiliadis, M. Berekovic, and T. D. Hämäläinen, Eds. Lecture Notes in Computer Science, vol. 4599. Springer, 283–293.

GALUZZI, C., MOSCU PANAINTE, E., YANKOVA, Y., BERTELS, K., AND VASSILIADIS, S. 2006. Automatic selection of application-specific instruction-set extensions. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'06)*. 160–165.

GEURTS, W. 1995. Synthesis of accelerator data paths for high-throughput signal processing applications. Ph.D. thesis, Katholieke Universiteit Leuven.

GEURTS, W. 1997. *Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications*. Kluwer Academic Publishers, Norwell, MA.

GOKHALE, M., HOLMES, W., KOPSER, A., LUCAS, S., MINNICH, R., SWEELY, D., AND LOPRESTI, D. 1991. Building and using a highly parallel programmable logic array. *Comput. 24,* 1, 81–89.

GOLDSTEIN, S. C., SCHMIT, H., MOE, M., BUDIU, M., CADAMBI, S., TAYLOR, R. R., AND LAUFER, R. 1999. Piperench: A co-processor for streaming multimedia acceleration. *SIGARCH Comput. Archit. News 27,* 2, 28–39.

GRASSELLI, A. AND LUCCIO, F. 1965. A method for minimizing the number of internal states in incompletely specified sequential networks. *IEEE Trans. Electron. Comp. EC-14*, 350–359.

GUO, Y. 2006. Mapping applications to a coarse-grained reconfigurable architecture. Ph.D. thesis, University of Twente, The Netherlands.

GUO, Y., SMIT, G. J., BROERSMA, H., AND HEYSTERS, P. M. 2003. A graph covering algorithm for a coarse grain reconfigurable system. In *Proceedings of the ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems (LCTES'03)*. 199–208.

GUTIN, G., JOHNSTONE, A., REDDINGTON, J., SCOTT, E., SOLEIMANFALLAH, A., AND YEO, A. 2007. An algorithm for finding connected convex subgraphs of an acyclic digraph. In *Proceedings of the ACiD 2007*.

HARTENSTEIN, R. 2001a. Coarse grain reconfigurable architecture (embedded tutorial). In *Proceedings of the Conference on Asia South Pacific Design Automation (ASP-DAC'01)*. 564–570.

HARTENSTEIN, R. 2001b. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'01)*. 642–649.

HAUCK, S., FRY, T. W., HOSLER, M. M., AND KAO, J. P. 1997. The chimaera reconfigurable functional unit. In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM'97)*.

HAUCK, S., FRY, T. W., HOSLER, M. M., AND KAO, J. P. 2004. The chimaera reconfigurable functional unit. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 12,* 2, 206–217.

HAUSER, J. R. AND WAWRZYNEK, J. 1997. Garp: a mips processor with a reconfigurable coprocessor. In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM'97)*.

HAYNES, S. D., CHEUNG, P. Y. K., LUK, W., AND STONE, J. 1999. Sonic - A plug-in architecture for video processing. In *Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*.

HAYNES, S. D., STONE, J., CHEUNG, P. Y. K., AND LUK, W. 2000. Video image processing with the sonic architecture. *Comput. 33,* 4, 50–57.

HOLMER, B. 1993. Automatic design of computer instruction sets. Ph.D. thesis.

HUANG, I.-J. AND DESPAIN, A. M. 1994a. Generating instruction sets and microarchitectures from applications. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'94)*. 391–396.

HUANG, I.-J. AND DESPAIN, A. M. 1994b. Synthesis of instruction sets for pipelined microprocessors. In *Proceedings of the 31st Annual Conference on Design Automation (DAC'94)*. 5–11.

HUANG, Z. AND MALIK, S. 2001. Managing dynamic reconfiguration overhead in system-on-a-chip design using reconfigurable datapaths and optimized interconnection networks. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'01)*. 735–740.

HUANG, Z., MALIK, S., MOREANO, N., AND ARAUJO, G. 2004. The design of dynamically reconfigurable datapath coprocessors. *Trans. Embed. Comput. Syst. 3,* 2, 361–384.

HUYNH, H. P., SIM, J. E., AND MITRA, T. 2007. An efficient framework for dynamic reconfiguration of instruction-set customization. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'07)*. 135–144.

IENNE, P. AND LEUPERS, R. 2006. *Customizable Embedded Processors: Design Technologies and Applications (Systems on Silicon)*. Morgan Kaufmann Publishers, San Francisco, CA.

IMAI, M., SATO, J., ALOMARY, A., AND HIKICHI, N. 1992. An integer programming approach to instruction implementation method selection problem. In *Proceedings of the Conference on European Design Automation (EURO-DAC'92)*. 106–111.

ISELI, C. 1996. Spyder: A reconfigurable processor development system. Ph.D. thesis, Ecole Polytechnique Federale de Lausanne.

ISELI, C. AND SANCHEZ, E. 1995. Spyder: A sure (superscalar and reconfigurable) processor. *J. Supercomput. 9,* 3, 231–252.

JANSSEN, M., CATTHOOR, F., AND DE MAN, H. 1996. A specification invariant technique for regularity improvement between flow-graph clusters. In *Proceedings of the European Conference on Design and Test (EDTC'96)*.

JAYASEELAN, R., LIU, H., AND MITRA, T. 2006. Exploiting forwarding to improve data bandwidth of instruction-set extensions. In *Proceedings of the 43rd Annual Conference on Design Automation (DAC'06)*. 43–48.

KASTNER, R., OGRENCI-MEMIK, S., BOZORGZADEH, E., AND SARRAFZADEH, M. 2001. Instruction generation for hybrid reconfigurable systems. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'01)*. 127–130.

KASTNER, R., KAPLAN, A., MEMIK, S. O., AND BOZORGZADEH, E. 2002. Instruction generation for hybrid reconfigurable systems. *ACM Trans. Des. Automa. Electron. Syst. (TODAES) 7,* 4, 605–627.

KAVVADIAS, N. AND NIKOLAIDIS, S. 2005. Automated instruction-set extension of embedded processors with application to mpeg-4 video encoding. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*. 140–145.

KAVVADIAS, N. AND NIKOLAIDIS, S. May 16-19, 2006. A flexible instruction generation framework for extending embedded processors. In *Proceedings of the 13th IEEE Mediterranean Electrotechnical Conference (MELECON'06)*. 125–128.

KEUTZER, K., MALIK, S., AND NEWTON, A. R. 2002. From asic to asip: The next design discontinuity. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*. 84–90.

LAM, S.-K. AND SRIKANTHAN, T. 2009. Rapid design of area-efficient custom instructions for reconfigurable embedded processing. *J. Syst. Archit. 55,* 1, 1–14.

LAM, S. K., SRIKANTHAM, T., AND CLARKE, C. T. 2006. Rapid generation of custom instructions using predefined dataflow structures. *Microprocess. Microsyst. 30,* 6, (*Special Issue on FPGA's*), 355–366.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'30)*. 330–335.

LEE, J.-E., CHOI, K., AND DUTT, N. 2002. Efficient instruction encoding for automatic instruction set design of configurable asips. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'02)*. 649–654.

LEE, J.-E., CHOI, K., AND DUTT, N. D. 2003a. Energy-efficient instruction set synthesis for application-specific processors. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design (ISLPED'03)*. 330–333.

LEE, J.-E., CHOI, K., AND DUTT, N. D. 2003b. An algorithm for mapping loops onto coarse-grained reconfigurable architectures. In *Proceedings of the ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems (LCTES'03)*. 183–188.

LEE, J.-E., CHOI, K., AND DUTT, N. D. 2007. Instruction set synthesis with efficient instruction encoding for configurable processors. *ACM Trans. Des. Autom. Electron. Syst. 12,* 1, 8.

LEUPERS, R., KARURI, K., KRAEMER, S., AND PANDEY, M. 2006. A design flow for configurable embedded processors based on optimized instruction set extension synthesis. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'06)*. European Design and Automation Association, 581–586.

LI, X. Y., STALLMANN, M. F., AND BRGLEZ, F. 2005. Effective bounding techniques for solving unate and binate covering problems. In *Proceedings of the 42nd Annual Conference on Design Automation (DAC'05)*. 385–390.

LIAO, S. AND DEVADAS, S. 1997. Solving covering problems using lpr-based lower bounds. In *Proceedings of the 34th Annual Conference on Design Automation (DAC'97)*. 117–120.

LIAO, S., DEVADAS, S., KEUTZER, K., AND TJIANG, S. 1995. Instruction selection using binate covering for code size optimization. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'95)*. 393–399.

LIAO, S., KEUTZER, K., TJIANG, S., AND DEVADAS, S. 1998. A new viewpoint on code generation for directed acyclic graphs. *ACM Trans. Design Automat. Electron. Syst. (TODAES) 3,* 1, 51–75.

LIEM, C., MAY, T., AND PAULIN, P. 1994. Instruction-set matching and selection for DSP and ASIP code generation. In *Proceedings of the European Design and Test Conference (ED&TC)*. 31–37.

LIN, S. AND KERNIGHAN, B. 1973. An effective heuristic algorithm for the traveling-salesman problem. *Oper. Res. 21,* 2, 498–516.

LU, G., SINGH, H., LEE, M.-H., BAGHERZADEH, N., KURDAHI, F. J., AND FILHO, E. M. C. 1999. The morphosys parallel reconfigurable system. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing (Euro-Par'99)*. Springer, 727–734.

MEI, B., VERNALDE1, S., VERKEST, D., MAN, H. D., AND LAUWEREINS, R. 2003. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL'03)*. Springer, 61–70.

MESSMER, B. T. AND BUNKE, H. 1995. Subgraph isomorphism in polynomial time. Tech. rep. IAM 95-003, University of Bern, Switzerland.

MIYAMORI, T. AND OLUKOTUN, K. 1998. Remarc (abstract): Reconfigurable multimedia array coprocessor. In *Proceedings of the ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays (FPGA'98)*.

MOREANO, N., ARAUJO, G., HUANG, Z., AND MALIK, S. 2002. Datapath merging and interconnection sharing for reconfigurable architectures. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS'02)*. 38–43.

NIEMANN, R. AND MARWEDEL, P. 1996. Hardware/software partitioning using integer programming. In *Proceedings of the European Conference on Design and Test (EDTC9'6)*.

NIEMANN, R. AND MARWEDEL, P. 1997. An algorithm for hardware/software partitioning using mixed integer linear programming. *Des. Automat. Embedd. Syst. 2,* 2, Special Issue: Partitioning Methods for Embedded Systems, 165–193.

PEYMANDOUST, A., POZZIL, L., IENNE, P., AND MICHELI, G. D. 2003. Automatic instruction set extension and utilization for embedded processors. In *Proceedings of the 14th International Conference on Application-Specific Systems, Architectures and Processors (ASAP'03)*. 108–118.

POTHINENI, N., KUMAR, A., AND PAUL, K. 2007. Application specific datapath extension with distributed i/o functional units. In *Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference (VLSID'07)*. 551–558.

POZZI, L. 2000. Methodologies for the design of application-specific reconfigurable vliw processors. Ph.D. thesis, Politecnico di Milano, Milano, Italy.

POZZI, L. AND IENNE, P. 2005. Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'05)*. 2–10.

POZZI, L., VULETIĆ, M., AND IENNE, P. 2001. Automatic topology-based identification of instruction-set extensions for embedded processors. Tech. rep. CS 01/377, EPFL, DI-LAP, Lausanne.

POZZI, L., VULETIĆ, M., AND IENNE, P. 2002. Automatic topology-based identification of instruction-set extensions for embedded processors. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'02)*.

POZZI, L., ATASU, K., AND IENNE, P. 2006a. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Trans. Comput.-Aid. Desi. Integra. Circ. Syst. 25,* 7, 1209 –1229.

POZZI, L., ATASU, K., AND IENNE, P. 2006b. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Trans. Comput.-Aid. Des. Integra. Circ. Syst. 25,* 7, 1209–1229.

RABAEY, J. 1997. Reconfigurable processing: The solution to low-power programmable dsp. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'97)*. vol. 1.

RADUNOVIC, B. AND MILUTINOVIC, V. M. 1998. A survey of reconfigurable computing architectures. In *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm (FPL'98)*. Springer, 376–385.

RAZDAN, R., BRACE, K. S., AND SMITH, M. D. 1994. PRISC software acceleration techniques. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computer & Processors (ICCS'94)*. 145–149.

RAZDAN, R. AND SMITH, M. D. 1994. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO'27)*. 172–180.

RUPP, C. R., LANDGUTH, M., GARVERICK, T., GOMERSALL, E., HOLT, H., ARNOLD, J. M., AND GOKHALE, M. 1998. The napa adaptive processing architecture. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'98)*.

SANG, S., LI, X., AND YE, Y. 2005. Automatic instruction generation for application specific co-processor. In *6th International Conference On ASIC (ASICON'05)*. 934–938.

SCHARWAECHTER, H., YOUN, J. M., LEUPERS, R., PAEK, Y., ASCHEID, G., AND MEYR, H. 2007. A code-generator generator for multi-output instructions. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'07)*. 131–136.

SETO, K. AND FUJITA, M. 2008. Custom instruction generation with high-level synthesis. In *Proceedings of the 2008 Symposium on Application Specific Processors (SASP)*. Anaheim, California, 14–19.

SREENIVASA RAO, D. AND KURDAHI, F. J. 1992. Partitioning by regularity extraction. In *Proceedings of the 29th ACM/IEEE Conference on Design Automation (DAC'92)*. 235–238.

SREENIVASA RAO, D. AND KURDAHI, F. J. 1993a. Hierarchical design space exploration for a class of digital systems. *IEEE Trans. Very Large Scale Integra. (VLSI) Syst. 1,* 3, 282–295.

SREENIVASA RAO, D. AND KURDAHI, F. J. 1993b. On clustering for maximal regularity extraction. *IEEE Trans. Comput.-Aid. Des. 12,* 8, 1198–1208.

STROZEK, L. AND BROOKS, D. 2006. Efficient architectures through application clustering and architectural heterogeneity. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*. 190–200.

SUN, F., RAVI, S., RAGHUNATHAN, A., AND JHA, N. K. 2002. Synthesis of custom processors based on extensible platforms. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'02)*. 641–648.

SUN, F., RAVI, S., RAGHUNATHAN, A., AND JHA, N. K. 2003. A scalable application-specific processor synthesis methodology. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'03)*.

SUN, F., RAVI, S., RAGHUNATHAN, A., AND JHA, N. K. 2004. Custom-instruction synthesis for extensible processor platform. *IEEE Trans. Comput.-Aid. Des. Integra. Circ. 23,* 2, 216–228.

TODMAN, T., CONSTANTINIDES, G., WILTON, S., MENCER, O., LUK, W., AND CHEUNG, P. 2005. Reconfigurable computing: Architectures and design methods. *IEE Proc. - Comput. Digital Tech. 152,* 2, 193–207.

VAN PRAET, J., GOOSSENS, G., LANNEER, D., AND DE MAN, H. 1994. Instruction set definition and instruction selection for asips. In *Proceedings of the 7th International Symposium on High-level Synthesis (ISSS'94)*. 11–16.

VASSILIADIS, S. AND SOUDRIS, D., Eds. 2007. *Fine- and Coarse-Grain Reconfigurable Computing*. Springer.

VASSILIADIS, S., WONG, S., AND COTOFANA, S. 2001. The molen $\varrho\mu$-coded processor. In *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications (FPL'01)*. Springer-Verlag, London, UK, 275–285.

VASSILIADIS, S., WONG, S., GAYDADJIEV, G., BERTELS, K., KUZMANOV, G., AND MOSCU PANAINTE, E. 2004. The molen polymorphic processor. *IEEE Trans. Comput. 53,* 11, 1363–1375.

VASSILIADIS, N., KAVVADIAS, N., THEODORIDIS, G., AND NIKOLAIDIS, S. 2006. A risc architecture extended by an efficient tightly coupled reconfigurable unit. *Inte. J. Electron. 93,* 6, 421–438.

VASSILIADIS, N., THEODORIDIS, G., AND NIKOLAIDIS, S. 2007. Enhancing a reconfigurable instruction set processor with partial predication and virtual opcode support. In *Proceedings of the 2nd International*

Workshop on Applied Reconfigurable Computing (ARC'06). Lecture Notes in Computer Science, vol. 3985. Springer, 217–229.

VERMA, A. K., ATASU, K., VULETIĆ, M., POZZI, L., AND IENNE, P. Nov. 2002. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 1st Workshop on Application Specific Processors (WASP-1)*.

VERMA, A. K., BRISK, P., AND IENNE, P. 2007. Rethinking custom ise identification: A new processor-agnostic method. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'07)*. 125–134.

WANG, A., KILLIAN, E., MAYDAN, D., AND ROWEN, C. 2001. Hardware/software instruction set configurability for system-on-chip processors. In *Proceedings of the 38th Conference on Design Automation (DAC'01)*. 184–188.

WAZLOWSKI, M., AGARWAL, L., LEE, T., SMITH, A., LAM, E., ATHANAS, P., SILVERMAN, H., AND GHOSH, S. 1993. Prism-ii compiler and architecture. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*. 9–16.

WIRTHLIN, M. J. AND HUTCHINGS, B. L. 1995. Disc: The dynamic instruction set computer. In *Proceedings of the International Society of Optical Engineering SPIE. Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*. vol. 2607. 92–103.

WITTIG, R. AND CHOW, P. 1996. OneChip: An FPGA processor with reconfigurable logic. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*. 126–135.

WITTIG, R. D. 1995. Onechip: An fpga processor with reconfigurable logic. M.S. thesis, Department of Electrical and Computer Engineering, University of Toronto.

WOLINSKI, C. AND KUCHCINSKI, K. 2007. Identification of application specific instructions based on subgraph isomorphism constraints. In *Proceedings of the IEEE International Application -specific Systems, Architectures and Processors*. 328–333.

WOLINSKI, C. AND KUCHCINSKI, K. 2008. Automatic selection of application-specific reconfigurable processor extensions. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'08)*. 1214–1219.

WONG, S., VASSILIADIS, S., AND COTOFANA, S. 2007. Instruction set extension generation with considering physical constraints. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers*. 291–305.

YE, Z. A., MOSHOVOS, A., HAUCK, S., AND BANERJEE, P. 2000. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *ACM SIGARCH Comput. Archit. News (Special Issue: Proceedings of the 27th annual international symposium on Computer architecture ISCA)*, 225–235.

YU, P. AND MITRA, T. 2004. Scalable custom instructions identification for instruction-set extensible processors. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04)*. 69–78.

YU, P. AND MITRA, T. 2005. Satisfying real-time constraints with custom instructions. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'05)*. 166–171.

YU, P. AND MITRA, T. 2007. Disjoint pattern enumeration for custom instructions identification. In *Proceedings of the 17th IEEE International Conference on Field Programmable Logic and Applications (FPL'07)*. Amsterdam, The Netherlands, –.

ZHAO, K., BIAN, J., DONG, S., SONG, Y., AND GOTO, S. 2008. Fast custom instruction identification algorithm based on basic convex pattern model for supporting asip automated design. *IEICE Trans. Fundam. Electron. Comm. Comput. Sci. E91-A,* 6, 1478–1487.

Received May 2008; revised September 2009; accepted January 2010