ASIP Design Methodologies : Survey and Issues

Manoj Kumar Jain, M. Balakrishnan, Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
{manoj,mbala,anshul}@cse.iitd.ernet.in

Abstract

Interest in synthesis of Application Specific Instruction Processors or ASIPs has increased considerably and a number of methodologies have been proposed in the last decade. This paper attempts to survey the state of the art in this area and identifies some issues which need to be addressed. We have identified the five key steps in ASIP design as application analysis, architectural design space exploration, instruction set generation, code synthesis and hardware synthesis. A broad classification of the approaches reported in the literature is done. The paper notes the need to broaden the architectural space being explored and to tightly couple the various subtasks in ASIP synthesis.

1 Introduction

An Application Specific Instruction Processor (ASIP) is a processor designed for a particular application or for a set of applications. An ASIP exploits special characteristics of application(s) to meet the desired performance, cost and power requirements.

According to Liem et al [17], ASIPs are a balance between two extremes : ASICs and general programmable processors. Since an ASIC is specially designed for one behavior, it is difficult to make any changes at a later stage. In such a situation, the ASIPs offer the required flexibility at lower cost than general programmable processors.

In this paper we present a survey of work done on ASIP design. Important steps involved in the ASIP design process are identified. Approaches employed for each major step are classified and important contributions are highlighted.

In section 2 we have described the steps involved in the ASIP design process, whereas, in section 3 we have discussed the approaches to application analysis which is crucial for synthesizing a suitable ASIP architecture. Sections 4 and 5 describe the various techniques to architectural exploration and instruction set generation respectively. Section 6 describes and classifies the approaches to code generation, and finally, section 7 presents the conclusion and future work.

2 Steps in ASIP Synthesis

Gloria et al [4] defined some main requirements of the design of application-specific architectures. Important among these are as follows.

- Design starts with the application behaviour.
- Evaluate several architectural options.
- Identify hardware functionalities to speed up the application.
- Introduce hardware resources for frequently used operations only if it can be supported during compilation.

Various methodologies have been reported to meet these requirements. We have studied these methodologies and found that typically there are five main steps followed in the synthesis of ASIPs. The steps are shown in fig. 1.

1. **Application Analysis :** Input in the ASIP design process is an application or a set of applications, along with their test data and design constraints. It is essential to analyze the application to get the desired characteristics/ requirements which can guide the hardware synthesis as well as instruction set generation. An application written in a high level language is analyzed statically and dynamically and analyzed information is stored in some suitable intermediate format, which is used in the subsequent steps.

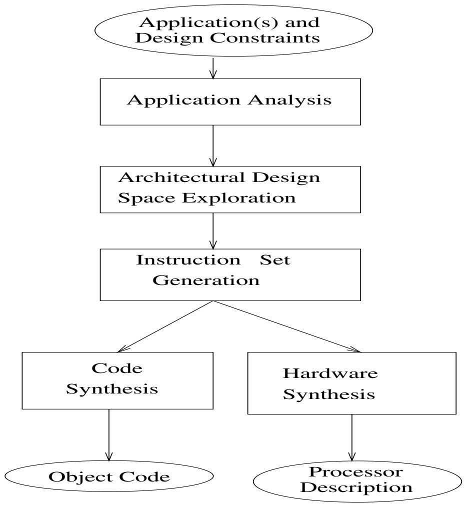

Figure 1: Flow diagram of ASIP design methodology

2. **Architectural Design Space Exploration :** First a set of possible architectures is identified for a specific application(s) using output of step 1 as well as the given design constraints. Performance of possible architectures is estimated and suitable architecture satisfying performance and power constraints and having minimum hardware cost is selected.

3. **Instruction Set Generation :** Instruction set is to be generated for that particular application and for the architecture selected. This instruction set is used during the code synthesis and hardware synthesis steps.

4. **Code Synthesis :** Compiler generator or retargetable code generator is used to synthesize code for the particular application or for a set of applications.

5. **Hardware Synthesis :** In this step the hardware is synthesized using the ASIP architectural template and instruction set architecture starting from a description in VHDL/ VERILOG using standard tools.

Every methodology does not emphasize all these steps. Some of them consider the processor micro-architecture to be fixed while only generating the instruction set within the flexibility provided by the micro-architecture, e.g. [6], [10] and [15], while others consider the process of instruction set generation only after the parallelism and functionality of the processor micro-architecture is finalized based on the application, e.g. [8], [11], [18] and [22].

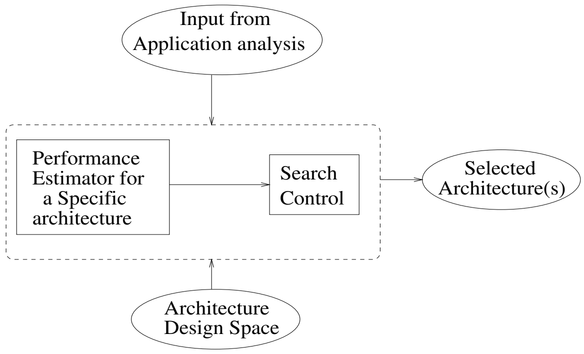

Figure 2: Block diagram of an architecture explorer

3 Application Analysis

Typically ASIP design starts with analysis of the applications. These applications with their test data should be analyzed statically and dynamically using some suitable profiler before proceeding further in the design process.

Sato et al [21] have developed an Application Program Analyzer (APA). The output of APA includes data types and their access methods, execution counts of operators and functions etc used in application program, the frequency of individual instructions and sequence of contiguous instructions.

The methodology suggested by Gupta et al [7] takes the application as well as the processor architecture as inputs. Using SUIF as intermediate format, a number of application parameters are extracted. These include the average basic block size, number of Multiply-Accumulate (MAC) operations, ratio of address computation instructions to data computation instructions, ratio of input/output instructions to the total instructions, average number of cycles between generation of a scalar and its consumption in the data flow graph etc. Similar analysis is performed in the scheme reported by Ghazal et al [3].

Saghir et al [20] have shown that dynamic behaviour of DSP kernels significantly differs from that of DSP application programs. This implies that kernels should play a more significant role in DSP application analysis especially for parameters related to operation concurrency.

4 Architectural Exploration

Figure 2 shows a typical architecture explorer block diagram. Inputs from the application analysis step are used along with the range of architecture design space to select a suitable architecture(s). The selection process typically can be viewed to consist of a search technique over the design space driven by a performance estimator. The methodologies considered differ both in the range and nature of architecture design space as well as the estimation technique employed.

4.1 Architecture design space

Almost all the techniques emphasize the need for a good parameterized model for the architecture. Different values can be assigned to the parameters (keeping design constraints into consideration), which constitute the design space. The design space will depend on the number of parameters and the range of values which can be assigned to these parameters.

The parameterized architecture model suggested by almost all the techniques includes the number of functional units of different types. Gong et al [5] consider storage units and interconnect resources also in their architectural model. Binh et al [2] emphasize on putting pipelined functional units in the model. Kienhuis et al [12] consider available element type (like buffer, controller, router and functional unit) and their composition rules which generate alternative implementations of elements in the model.

Kin et al [13] consider issue width, number of branch units, number of memory units, the size of instruction cache and size of data cache etc. in the model. Gupta et al [7] use a model which include parameters like number of registers, number of operation slots in each instruction, concurrent load/store operations and latency of functional units and operations. Ghazal [3] include optimizing features like addressing support, instruction packing, memory pack/ unpack support, loop vectorization, complex arithmetic patterns such as dual multiply-accumulate, complex multiplication etc in their architectural model. This results in a large design space. They have developed a retargetable estimator which takes advantage of such an architecture model.

Architectures considered by different researchers also differ in terms of the instruction level parallelism they support. For example [2] and [12] do not support instruction level parallelism, whereas [5] and [7] support VLIW architecture and [3] and [13] support VLIW as well as super scalar architecture.

Most of these approaches consider only a flat memory. Only [13] addresses consideration of instruction and data cache sizes during design space exploration, but the range of architectures explored is rather limited. Similarly no approach considers flexibility in terms of number of stages in a pipeline, though a pipelined architecture is considered in [2] and [4].

4.2 Performance Estimation

In the literature two major techniques have been used for performance estimation. They are scheduler based and simulator based.

4.2.1 Scheduler based

In this type of approach the problem is formulated as a resource constrained scheduling problem with the selected architectural components as the resources and the application is scheduled to generate an estimate of the cycle count. Profile data is used to obtain the frequency of each operation. Examples of such approaches are [7], [3] and [5].

One of the important feature of their architectural model used in [3] and [7] is that it captures the differentiating features of the instruction set and special functional resources, rather than the complete specification required for code synthesis or simulation. This helps in fast performance estimation. Further estimator used in [3] also produces a trace of the application annotated with the suggested optimizations and ranked bottlenecks. The optimizations considered include : optimized multi-operation pattern matching (e.g. multiply-accumulate), address mode optimization (e.g. auto-update, circular buffer), loop optimization (e.g. pre-fetched sequential loads), loop vectorization/ packing, simple if-else conversion (by use of predicated instructions), rescheduling within basic blocks, loop unrolling, software pipelining etc.

4.2.2 Simulator based

In the approaches suggested by [12], [13], [4] and [2], a simulation model of the architecture based on the selected features is generated and the application is simulated on this model to compute the performance.

Kienhuis et al [12] constructed a retargetable simulator for an architecture template. For each architecture instance of the architecture template, a specific simulator is derived in three steps. The architecture instance is constructed, an execution model is added and the executable architecture is instrumented with metric collectors to obtain performance numbers. Object oriented principles together with a high-level simulation mechanism are used to ensure retargetability and efficient simulation speed.

A system for the evaluation of architectures has been presented by Gloria et al [4]. The simulator, by using pre-defined modules and the architecture description, reports performance and statistics about resource utilization. Its engine is an event-driven algorithm supporting various memory hierarchy structures (data-cache, instruction-cache, bank interleaving, etc.).

Most researchers have focused on performance and area but do not address the power consumption. Only Kin et al[13] and Imai et al[2] considered power consumption. Kin et al[13] computes power consumption of the generated ASIP from a very coarse model which is based on the total cycle count.

4.3 Search Control

Algorithms for search range from branch-and-bound ([2] and [13]) to exhaustive over a restricted domain ([7]).

Binh et al [2] suggested a HW/SW partitioning algorithm (branch-and-bound) for synthesizing the highest performance pipelined ASIPs with multiple identical functional units. Kin et al [13] first reduced the search space by eliminating machine configurations not satisfying given area constraint and those that are dominated by at least one another machine configuration and then they used a branch-and-bound algorithm for searching for an optimum solution.

Gupta et al [7] developed a processor selector which first reduces the domain of search by restricting the various architectural parameter ranges based on certain gross characteristics of the processors, such as branch-delay, presence/ absence of multiply-accumulate operation, depth of pipeline and memory bandwidths. The output is a smaller subset of processors. Processor description files of this restricted range of processors are supplied to the performance estimator which exhaustively explores this set.

5 Instruction Set Generation

We have found that almost all approaches to instruction set generation can be classified as either instruction set synthesis approach or instruction set selection approach on the basis of how they are generating instructions.

5.1 Instruction set synthesis

In this class of techniques, instruction set is synthesized for a particular application based on the application requirements, quantified in terms of the required micro-operations and their frequencies. Examples of such approaches are in [11], [10] and [6]. These approaches differ in the manner the instructions are formed, from a list of operations.

Huang and Despain [11] integrated the problem of instruction formation with the scheduling problem. Simulated annealing technique is used to solve the scheduling problem. Instructions are generated from time steps in the schedule. Each time step corresponds to one instruction. This approach was able to handle only single cycle instructions. Hoon et al [10] proposed another approach which supports multi-cycle complex instructions as well. First the list of micro-operations is matched to the primary instructions in order to estimate the execution time if only these instructions are employed. If the estimated performance is unacceptable then the complex instructions are considered. If the performance is still unacceptable, special instructions which require special hardware units are included. Gshwind [6] describe an approach for application specific design based on extendible microprocessor core. Critical code portions are customized using the application-specific instruction set extensions.

5.2 Instruction selection

In this class of techniques a superset of instructions is available and a subset of them is selected to satisfy the performance requirements within the architectural constraints. Examples of such approaches are in [1], [2], [17] and [22]. All the approaches in this technique differ in the algorithms they are using to select the instructions from the super set.

Imai et al [1] classified functionalities as primary, basic and extended, and the intermediate instructions are described as primary RTL, basic RTL, and extended RTL respectively for these classes. Generated ASIP will include hardware modules corresponding to all of the primary RTL, but only a part of the basic RTL and extended RTL. Selection problem is formulated as an integer linear programming problem with the objective to maximize the performance of the CPU under the constraints of the chip area and power consumption. The branch-and-bound algorithm is used to solve the problem. Later, they have revised their formulation considering functional module sharing constraints and pipelining [2].

Some approaches ([17], [22] and [15]) use pattern matching for instruction selection. The set of template patterns is extracted from the instruction set and the graph representing the intermediate code of the application is covered by these patterns. By arranging instruction patterns in a tree structure, matching process becomes significantly fast.

Liem et al [17] adopts a pattern matching approach to instruction set selection. A wide range of instruction types is represented as a pattern set, which is organized

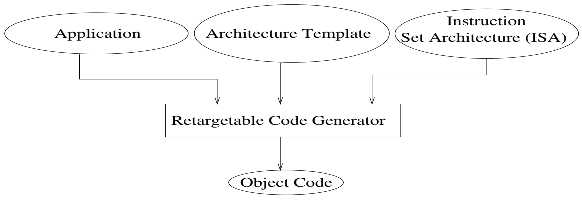

Figure 3: Retargetable Code Generator

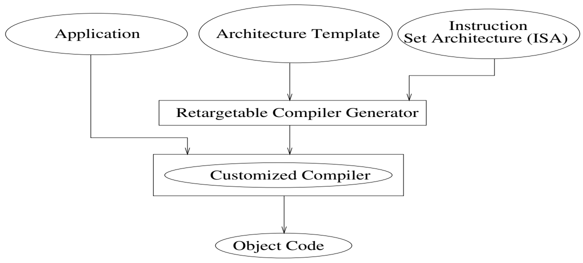

Figure 4: Retargetable Compiler

in a manner such that matching is extremely efficient and retargetting to architectures with a new instruction set is well defined. Shu et al [22] have implemented the matching method based on a pattern tree structure of instructions. Two genetic algorithms are implemented for pattern selection: a pure GA which uses standard GA operators and a GA with backtracking which employs variable-length chromosomes.

Our analysis also shows that there are design methodologies in both the instruction synthesis ([6], [10]) as well as instruction selection ([1], [2]) categories which start from a basic instruction set and only synthesize/ select application-specific special or complex instructions.

There is an interesting approach given by Praet et al [18] which emphasizes instruction generation by a mix of instruction selection and instruction synthesis. The main steps involved in this technique are instruction selection by bundling and instruction set definition. A bundle is a maximal sequence of micro-operations in which each micro-operation is directly coupled to its neighbours. Graph structure was used for pattern matching rather than tree structure. Instruction set definition includes selecting the representive application parts, selecting initial data part out of a library based on statistics obtained from the analysis tool, iteratively updating these parts considering area/ performance trade offs and defining instruction encodings.

6 Code Synthesis

The researches have followed two different approaches for code synthesis. They are retargetable code generator and compiler generator.

6.1 Retargetable Code Generator

Taking architecture template, instruction set architecture and the application as inputs, object code is generated (fig. 3). [23], [15], [14], [19] and [8] follow such an approach. All these approaches try to address following code generation sub-problems.

1. Instruction mapping.
2. Resource allocation and binding.
3. Scheduling.

Some approaches ([14], [15]) address issue of code compaction also while generating code.

Wilson et al [23] formulated an integer linear programming model to solve the three subproblems concurrently. Their model considers spill over to memory as well. Leupers et al [15] suggest a two phase scheme using pattern matching for instruction selection. In the first phase the instructions are modeled as tree patterns and an optimal cover is computed, while in the second phase a code compaction considering instruction level parallelism is done.

Kreuzer et al [14] suggest a method for code compaction and register optimization to exploit instruction level parallelism based on *trellis tree straight-line code generation* algorithm. Retargetability is provided by the use of target architecture description file. Algorithms for intermediate code generation, compaction and optimizations are processor independent. Praet et al [19] have used graph based processor model in their approach which include the processor connectivity and parallelism. They have defined the sub-problems of code generation in terms of this processor model. Hanono and Devadas [8] address all three subproblems concurrently using branch-and-bound approach. A detailed register allocation is carried out in the second step, whereas its estimates are generated in the first step.

6.2 Compiler Generator

Taking architecture template and instruction set architecture as the inputs, a customized compiler is generated. This is used to generate object code for the given application written in a high level language (fig. 4). Examples of such approaches are in [9] and [16]. A compiler generated in this manner generally has phases similar to general compilers, namely, program analysis, intermediate code optimization and code generation. The difference is that the optimizations are tailored to the specific architecture-application combination.

Hatcher et al [9] developed a *code generator generator* that accepts a machine description in a YACC-like format and a set of C structure definitions for valid tree nodes and produces C source (both functions and initialized data) for a code generator which is time and space efficient. The technique suggested by Leupers et al [16] does not require tool-specific modeling formalism, but starts from general HDL processor models. All processor aspects needed for code generation are automatically derived.

7 Conclusion and future work

We have identified five key steps in the process of designing ASIPs. We have presented a survey on the work done while attempting classification of the approaches for each step. Performance estimation is based either on scheduler based or simulation based technique. Instruction set is generated either through synthesis or a selection process. Code is synthesized either by a retargetable code generator or by a custom generated compiler.

Though, a variety of approaches have evolved in addressing each of the key steps, the target architecture space being explored by these methodologies is limited. With the increase in integration, it should be possible to support memory hierarchies on the chip and the same has not been addressed in an integrated manner. Similarly issues of pipelined ASIP design as well as low power ASIP design is in its infancy. We also observe that the problems of processor synthesis and retargetable code generation have been considered in isolation.

References

[1] Alomary, A.; Nakata, T.; Honma, Y.; Imai, M.; Hikichi, N. : "An ASIP instruction set optimization algorithm with functional module sharing constraint.", Proc. ICCAD-93, 7-11 Nov. 1993, pp. 526-532.

[2] Binh, N.N.; Imai, M.; Shiomi, A. : "A new HW/SW partitioning algorithm for synthesizing the highest performance pipelined ASIPs with multiple identical FUs.", Proc. DAC-96, 16-20 Sept. 1996, pp. 126-131.

[3] Ghazal, N.; Newton, R.; Jan Rabaey. : "Retargetable estimation scheme for DSP architecture selection.", Proc. ASP-DAC-2000, 25-28 Jan. 2000, pp. 485-489.

[4] Gloria A. D.; Faraboschi, P. : "An evaluation system for application specific architectures.", Proc. Micro-23, 27-29 Nov. 1990, pp. 80-89.

[5] Gong J.; Gajski, D.D.; Nicolau, A. : "Performance evaluation for application-specific architectures.", IEEE Trans. on VLSI Systems, Dec. 1995 Vol. 3 Issue: 4, pp. 483-490.

[6] Gschwind, M. : "Instruction set selection for ASIP design.", Proc. CODES-99, 3-5 May 1999, pp. 7-11.

[7] Gupta, T.V.K.; Sharma, P.; Balakrishnan, M.; Malik, S. : "Processor evaluation in an embedded systems design environment.", Proc. VLSI Design 2000, 3-7 Jan. 2000, pp. 98-103.

[8] Hanono, S.; Devadas, S. : "Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator.", Proc. DAC-98, 15-19 June 1998, pp. 510-515.

[9] Hatcher, P.J.; Tuller, J.W. : "Efficient retargetable compiler code generation.", Proc. Int. Conf. on Computer Languages 1988, 9-13 Oct. 1988, pp. 25-30.

[10] Hoon Choi; In-Cheol Park; Seung Ho Hwang; Chong-Min Kyung : "Synthesis of application specific instructions for embedded DSP software.", Proc. ICCAD-98, 8-12 Nov. 1998, pp. 665 - 671.

[11] Huang Ing-Jer; Despain, A.M. : "Synthesis of application specific instruction sets.", IEEE Trans. on CAD of IC & Systems, June 1995 Vol. 14 Issue: 6 pp. 663-675.

[12] Kienhuis, B.; Deprettere, E.; Vissers, K.; van der Wolf, P. : "The construction of a retargetable simulator for an architecture template.", Proc. CODES/CASHE-98), 15-18 March 1998, pp. 125-129.

[13] Kin, J.; Chunho Lee; Mangione-Smith, W.H.; Potkonjak, M. : "Power efficient mediaprocessors: design space exploration.", Proc. DAC-99, 21-25 June 1999, pp. 321-326.

[14] Kreuzer, W.; Gotschlich, M.; Wess, B. : "A retargetable optimizing code generator for digital signal processors.", Proc. ISCAS-96, Connecting the World., 1996, Vol. 2, pp. 257-260.

[15] Leupers, R.; Marwedel, P. : "Instruction selection for embedded DSPs with complex instructions.", Proc. EURO-DAC '96, 16-20 Sept. 1996, pp. 200-205.

[16] Leupers, R.; Marwedel, P. : "Retargetable generation of code selectors from HDL processor models.", Proc. ED&TC-97, 9-13 June 1997, pp. 534-539.

[17] Liem, C.; May, T.; Paulin, P. : "Instruction-set matching and selection for DSP and ASIP code generation.", Proc. EURODAC-94, 28 Feb.-3 March 1994, pp. 31-37.

[18] Praet J.V.; Goossens, G.; Lanneer, D.; De Man, H. : "Instruction set definition and instruction selection for ASIPs.", Proc. Int. Symp. on High-Level Synthesis 1994, 18-20 May 1994, pp. 11-16.

[19] Praet, J.V.; Lanneer, D.; Goossens, G.; Geurts, W.; De Man, H. : "A graph based processor model for retargetable code generation.", Proc. ED&TC-96, 11-14 March 1996, pp. 102-107.

[20] Saghir, M.A.R.; Chow, P.; Lee, C.G. : "Application-driven design of DSP architectures and compilers.", Proc. ICASSP-94, 19-22 April 1994, pp. II/437-II/440 Vol.2.

[21] Sato, J.; Imai, M.; Hakata, T.; Alomary, A.Y.; Hikichi, N. : "An integrated design environment for application specific integrated processor.", Proc. ICCD-91, 14-16 Oct. 1991, pp. 414-417.

[22] Shu, J.; Wilson, T.C.; Banerji, D.K. : "Instruction-set matching and GA-based selection for embedded-processor code generation.", Proc. VLSI-96, 3-6 Jan. 1996, pp. 73-76.

[23] Wilson, T.; Grewal, G.; Halley, B.; Banerji, D. : "An integrated approach to retargetable code generation.", Proc. Int. Symp. on High-Level Synthesis 1994, 18-20 May 1994, pp. 70-75.