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 Set 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 Set Processor (ASIP) is a processor designed for a particular application or for a set of specific applications. An ASIP exploits special characteristics of application(s) to meet the desired performance, cost and power requirements. ASIPs can be very efficient when applied to a specific application such as digital signal processing, servo motor control, automatic control systems, avionics, cellular phones etc [35, 5].

According to Liem et al [31], ASIPs are a balance between two extremes : Application Specific Integrated Circuits (ASICs) and general programmable processors. ASIPs offer the availability of custom sections for time critical tasks (e.g. a multiply-add for real time DSP), and offer flexibility through an instruction set. Hoon et al [15] observed that for the complex applications more flexibility is required to accommodate design errors and specification changes, which may happen at the later design stages. Since an ASIC is specially designed for one behaviour, 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.

Several issues in ASIP design process are addressed in various approaches. We have identified the important steps involved in the process. An attempt is made to classify the approaches for each step on the basis of the technique used, highlighting their important contributions. A comparison table is generated wherever possible. At least one approach is explained in detail for each class during classification. The paper notes the need to broaden the architecture space being explored and tightly couple the various subtasks.

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 for architectural exploration and instruction set generation respectively. Section

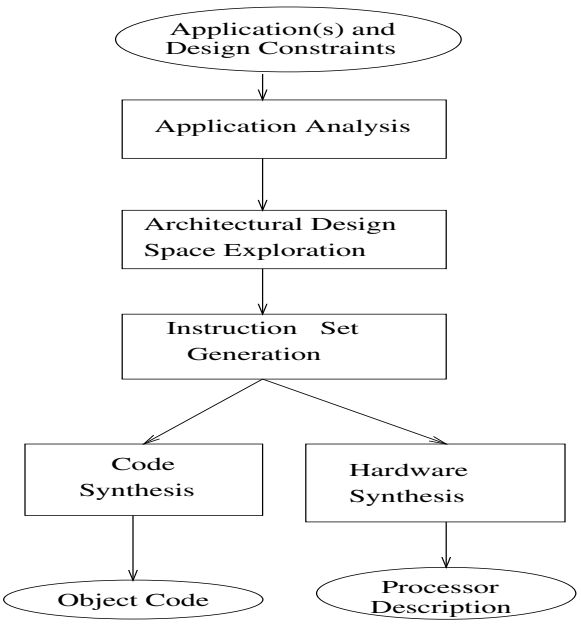

Figure 1: Flow diagram of ASIP design methodology

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 [9] 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.

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 architectures satisfying performance and power constraints and having minimum hardware cost is selected.

3. **Instruction Set Generation :** Instruction set is to be synthesized or selected from a provided superset of instructions 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 :** Code is synthesized in two different ways. Either a compiler generator is used to generate a compiler which is used to synthesize code or a 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. [11], [15] and [29], 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. [13], [17], [32] and [36].

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 [35] 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 [12] 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 [8].

Saghir et al [34] 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

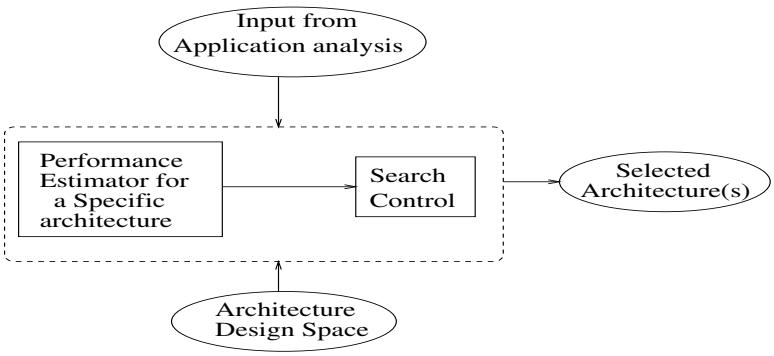

Figure 2: Block diagram of an architecture explorer

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 techniques employed.

4.1 Architecture design space

Almost all the techniques suggested by various researchers emphasized the need for a good parameterized model for the architecture. Different values can be assigned to the parameters (of course keeping design constraints into consideration), which constitute the design space. So the size of 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 [10] consider storage units and interconnect resources also in their architectural model. Binh et al [2, 3] emphasize on incorporating pipelined functional units in the model. Kienhuis et al [19] consider available element types (like buffer, controller, router and functional unit) and their composition rules in their architectural model. Composition rules generate alternative implementations of elements. All parameters together support a large design space.

Kin et al [20] 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 [12] 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 [8] 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 [3] and [19] do not support instruction level parallelism, whereas [10] and [12] support VLIW architecture and [8] and [20] support VLIW as well as super scalar architecture.

Most of these approaches consider only a flat memory. Only [20] 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 [3] and [9]. Table 1 compares various architecture models used in major approaches for design space exploration.

Table 1: Comparison of various architecture models used for design space exploration

Approach → Attribute ↓	Gloria [9]	Sato [35]	Gong [10]	Binh [2, 3]	Kienhuis [19]	Kin [20]	Gupta [12]	Gazhal [8]
Number of FUs of different type	yes	yes	yes	yes	yes	yes	yes	yes
Register organization		yes	yes		yes		yes	
Storage units		yes	yes	yes			yes	yes
Interconnect resources	yes		yes					
Pipelining	yes			yes				
Controller					yes			
Variable Issue-width						yes		
No. of branch units						yes		
Memory hierarchy	yes					yes		
Instruction level parallelism	VLIW	no	VLIW	no	no	VLIW, super-scalar	VLIW	VLIW, super-scalar
Addressing support							yes	yes
Memory pack/ unpack								yes
Loop vectorization								yes
Complex arith. patterns							yes	yes
Distinguishing	unconstrained FU complex-ity		storage units interconnect resources consid-eration	pipelining	consideration of al-ternative implemen-tation of elements	memory hier-archy	optimizing features considered	optimizing features considered

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 [10], [12] and [8].

Gong et al [10] proposed a scheduler which takes into account not only functional and storage unit resources but also interconnect resources for performance evaluation. They use a customized GNU C compiler (GCC) to translate a C program into three address instructions. The three-address instruction format is based on a RISC like load/store architecture in which only load and store operations access memory and all other operations work on registers. A parallelizing compiler which is used to extract program parallelism is a fine-grain VLIW compiler which takes serial three-address instruction code produced by GCC as input and generates parallel three-address instruction code in which an instruction may contain more than one operation. The VLIW compiler schedules operations as soon as possible using unlimited resources. A ultra-fine-grain scheduler is developed to exploit hardware parallelism offered by various architectures. It takes either the serial three-address instruction code or the parallel three-address instruction code as input and schedules them on various application-specific architectures instantiated from the parameterized architecture model. The output is the parallel control code in which register transfers in each control step are performed concurrently. A simulator is used to mimic the execution of the serial three-address code, the parallel three-address code as well as parallel control code to obtain performance metrics.

Gupta et al [12] have developed a resource constrained scheduler that estimates the number of clock cycles, given the description file of the target processor architecture, application timing constraints and the profiling data. A key feature of this list scheduler is its capability to reflect the flexibility of the instruction set to handle concurrency. Processor selector rejects processors which do not meet the constraints. Ghazal et al [8] have developed a retargetable estimator for performance estimation required for architectural exploration using a better architectural model. A distinguishing feature of their approach is that they consider several optimizing transformations while mapping the application on to the target architecture. 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. The flow diagram of the estimation scheme followed in both ([8] and [12]) approaches is shown in fig. 3

One of the important features of the architectural models used in [8] and [12] is that it captures the differentiating capabilities 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 [8] also produces a trace of the application annotated with the suggested optimizations and ranked bottlenecks.

4.2.2 Simulator based

In the approaches suggested by [9], [19], [20], [18] [3] and [4], 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.

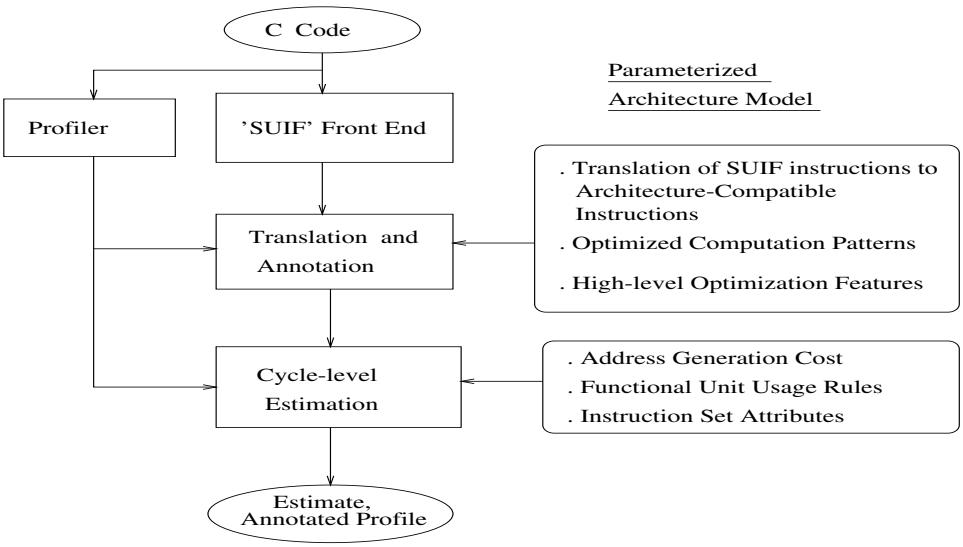

Figure 3: The Flow of Estimation Scheme : Ghazal [8] & Gupta [12]

A system for the evaluation of architectures has been presented by Gloria et al [9]. The simulator, by using predefined modules and the architecture description, reports performance and statistics about resource utilization. Its engine is an event-driven algorithm also supporting various memory hierarchy structures (data-cache, instruction-cache, bank interleaving, etc.). The application is translated into an intermediate code, composed of simple instructions directly expressing primitive hardware functionalities. Optimization techniques such as loop unrolling, data-dependency analysis, variable renaming and loop interchanging, are further applied in order to increase the degree of parallelism. A sequential code simulation is then performed, in order to validate the code, and to extract some early features which can perform preliminary architectural choices. For instance, frequently executed sequence of instructions can be substituted by the introduction of new instructions.

Kienhuis et al [19] 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. Kin et al [20] propose an approach to design space exploration for media processors considering power consumption. Aggressively optimized code is generated using *IMPACT* compiler to increase instruction level parallelism. The optimized code is consumed by the *Lsim* simulator. Measured run-times of benchmarks through simulations are used to compute the energy based on the power model they have described. The power dissipation numbers are normalized with respect to a baseline machine power dissipation.

Most researchers have focused on performance and area but do not address the power consumption. Only Kin et al[20] and Imai et al[18] considered power consumption. Kin et al[20] computes power consumption of the generated ASIP from a very coarse model which is based on the total cycle count. Main approaches considering performance estimation are compared in table 2.

Table 2: Comparison of major performance measurement techniques

Approach → Attribute ↓	Gloria [9]	Gong [10]	Binh [2, 3]	Kienhuis [19]	Kin [20]	Gupta [12]	Gazhal [8]
Technique	simulation based	scheduler based	simulation based	simulation based	simulation based	scheduler based	scheduler based
Algorithm		extension of list scheduling	branch-and-bound			list scheduling	list scheduling
FU consideration		yes	multiple identical FUs			yes	yes
Interconnect resources		yes				yes	yes
Optimizations	loop-unrolling, data-dependency analysis, variable-renaming, loop interchange	no			instruction level parallelism	optimized multi-operation pattern matching, address modes, rescheduling, loop unrolling	as by Gupta [12] and loop vectorization, memory packing, software pipelining etc.
Measure of performance	cycle count and resource allocation statistics			performance number		cycle count	cycle count and trace of the application annotated with suggested optimizations
Memory hierarchy	yes				yes		
Other main features			RAM and ROM sizes consideration	metric collectors used to obtain performance number	power consumption and area constraints consideration	only differentiating features of instruction set is captured	only differentiating features of instruction set is captured

4.3 Search Control

Algorithms for search range from branch-and-bound ([3] and [20]) to exhaustive over a restricted domain ([12]).

Binh et al [3] suggested a HW/SW partitioning algorithm (branch-and-bound) for synthesizing the highest performance pipelined ASIPs with multiple identical functional units. Constraints considered are given gate count and power consumption. They have improvised their algorithm considering RAM and ROM sizes as well as chip area constraints [4]. Chip area includes the hardware cost of the register file for a given application program with the associated input data set. This optimization algorithm defines the best trade offs between the CPU core, RAM and ROM of an ASIP chip to achieve highest performance while satisfying design constraints on the chip area. Kin et al [20] 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 [12] 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 [16, 17], [15] and [11]. These approaches differ in the manner the instructions are formed, from a list of operations.

Huang and Despain [16, 17] integrated the problem of instruction formation with the scheduling problem. The 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. Pipelined machine with data stationary control model is assumed. They had considered goal of the instruction set design as to optimize and trade off the instruction set size, the program size and the number of cycles to execute a program. General multi-cycle instructions are not considered. However multi-cycle arithmetic/logic operations, memory access, and change of control flow (branch/jump/call) are supported by specifying the delay cycles as design parameters. The formulation takes the application, architecture template, objective function and design constraints in the input and generates as outputs the instruction set, resource allocation (which instantiates the architecture template) and assembly code for the application. This approach is shown in fig. 4.

Hoon et al [15] 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.

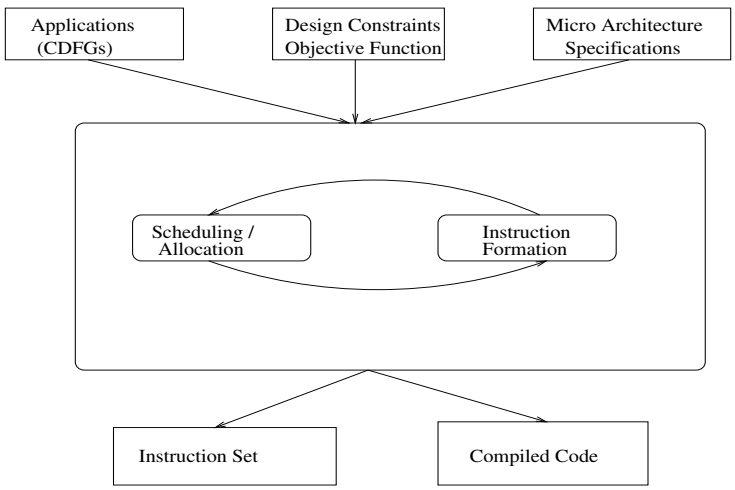

Figure 4: The integrated scheduling / instruction-formation process

If the performance is still unacceptable, special instructions which require special hardware units are included. Gshwind [11] 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], [18], [31], [29] and [36]. All the approaches in this technique differ in the algorithms they are using to select the instructions from the super set.

Imai et al [35, 18] assumes that the instruction set can be divided into two groups : operations and functions. It is relatively easy for a compiler to generate instructions corresponding to operators used in C language compared to those which corresponds to functions. It is because that the full set of operations is already described, but the full set of user-defined functions is not known a priori. They further divide the set of operations into two subgroups : primitive operators and basic operators. The set of primitive operators is chosen so that other basic operators can be substituted by a combination of primitive operators. Thus they have classified functionalities as follows.

- Primitive Functionalities (PF) : Can be realized by a minimal hardware component, such as ALU and shifter.
- Basic Functionalities (BF) : Set of operations used in C except those included in PF.
- Extended Functionalities (XF) : Library functions or user defined functions.

The intermediate instructions are described as primary RTL, basic RTL, and extended RTL respectively for these classes. The generated ASIP will include hardware modules corresponding to all of the primary RTL, but only a part of the basic RTL and extended RTL. The 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 [1, 2].

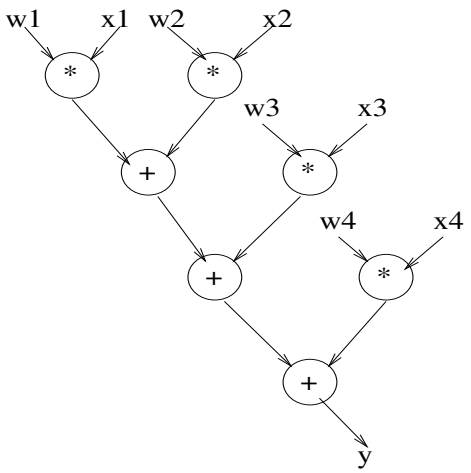

Figure 5: Expression tree

Some approaches ([31], [36] and [29]) 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, the matching process becomes significantly fast. For example there is an expression as follows.

$$y = w1 * x1 + w2 * x2 + w3 * x3 + w4 * x4$$

The corresponding expression tree is shown in fig. 5. This type of expression evaluation is very frequent in DSP domain. The hardware configuration required to implement it is shown in fig. 6 a. The instruction patterns are shown in fig. 6 b. Apart from intrinsic patterns MULT and ADD it has MAC also which performs multiplication and addition together. If only intrinsic patterns (ADD and MULT) are available for pattern matching then we get matching as shown in fig. 7 b. If we use the pattern MAC as well then we get optimal cover as shown in fig. 7 a. Using cost and delay of different patterns and the frequency count of this expression tree received from dynamic profiling the cost-performance trade offs can be evaluated, and it can be decided that such complex instructions should be included or not in the selected instruction set.

Liem et al [31] adopts a pattern matching approach to instruction set selection. A wide range of instruction types is represented as a pattern set, which is organized in a manner such that matching is extremely efficient and retargetting to architectures with new instruction set is well defined. Shu et al [36] have implemented the matching method based on a pattern tree structure of instructions. Two genetic algorithms (GA) 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 ([11] and [15]) as well as instruction selection ([1], [2] and [18]) categories which start from a basic instruction set and only synthesize/ select application-specific special or complex instructions. A comparison of various approaches for instruction set generation is shown in table 3.

There is an interesting approach given by Praet et al [32] 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. Directly coupled micro-operations are primitive processor activities

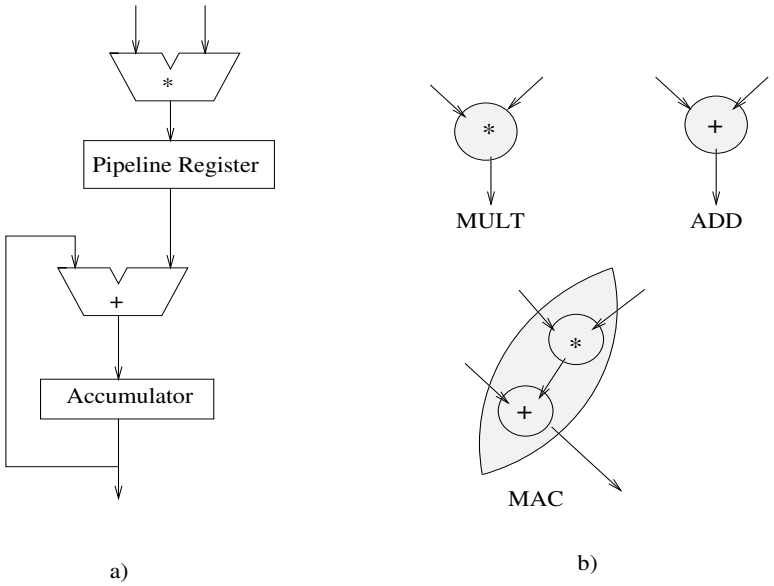

Figure 6: a) Multiplier-Accumulator , b) Instruction Patterns

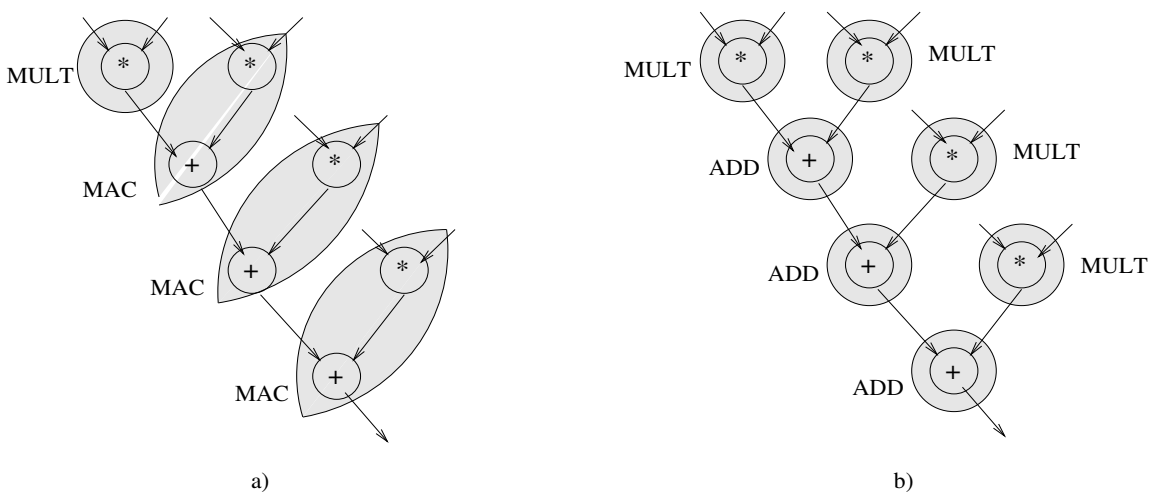

Figure 7: a) Optimal cover with MAC , b) Cover with intrinsic patterns only

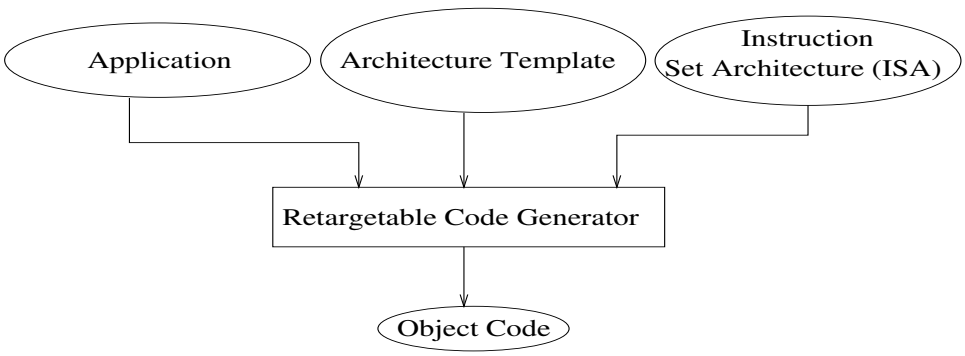

Figure 8: Retargetable Code Generator

which pass data to each other through a wire or latch. A graph structure was used for pattern matching rather than a 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 work reported 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 (ISA) and application as inputs, object code is generated (fig. 8). [6], [38], [24, 25, 29], [21], [33], [13], [37] and [39] follow such an approach.

All these approaches try to address following sub-problems while synthesizing instructions

1. Instruction mapping: DFG of a given application is mapped to the instruction patterns using tree or graph pattern matching technique in an attempt to get a good cover.
2. Resource allocation and binding: Resources like registers, memory, FUs, buses etc are allocated and corresponding bindings are performed.
3. Scheduling: Instructions are scheduled for execution.

Some approaches address the issue of code compaction also while generating code. Examples are [21], [24], [25] and [29].

Cheng and Lin [6] address the issue of reducing the memory access and register usage conflicts. The goal is to maximize the possibility of scheduling more instructions into a control step. Wilson et al [38] formulated an integer linear programming model to solve the three subproblems concurrently. Their model considers spill over to memory as well. Leupers et al [24, 25, 29] suggest a two phase scheme using pattern matching for instruction mapping. 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.

Table 3: Comparison of major instruction generation approaches

Approach → Attribute ↓	Imai [35, 18, 1, 2]	Liem [31]	Praet [32]	Huang [16, 17]	Shu [36]	Leupers [29]	Hoon [15]	Gshwind [11]
Generation mechanism	selection	selection	selection and synthesis	synthesis	selection	selection	synthesis	synthesis
Technique/ Algorithm	ILP, branch-and-bound	tree pattern matching, dynamic programming	graph pattern matching	modified scheduling problem, simulated annealing	tree pattern matching, genetic algorithms	ILP, dynamic programming	sub-set-sum problem	hw/sw co-evaluation
Instruction types	primary, basic and special			single-cycle			single-cycle, multi-cycle complex	intrinsic, complex
Other main issues and distinguishing features	primary, basic and special RTL for basic, primary and special functionalities	capture behavioural representation of instructions		instruction utilization, instruction operand encoding, delay load/ store and delay branches. integrated instruction formation with scheduling problem		better optimization than tree pattern matching	generated single and multi-cycle complex instructions then sub-set-sum technique	extendible core technique, application specific instruction set extensions

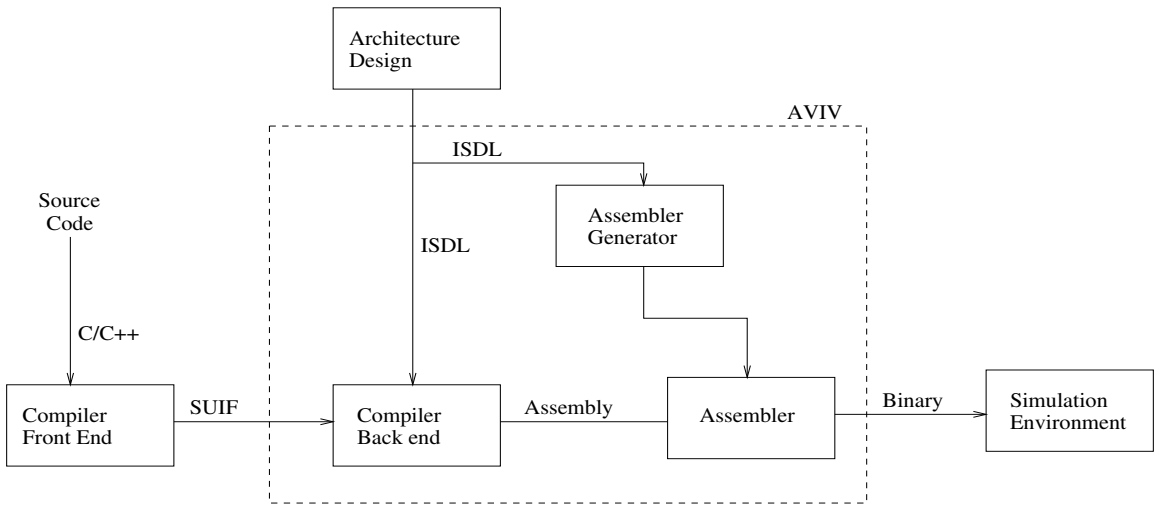

Figure 9: Compiler frame work used by *AVIV*

Kreuzer et al [21] 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 [33] have used graph based processor model in their approach which includes the processor connectivity and parallelism. They have defined the sub-problems of code generation in terms of this processor model. Yamaguchi et al [39] presented new binding and scheduling algorithms for a retargetable compiler which can deal with diverse architectures.

Hanono and Devadas [13] address all three subproblems concurrently using branch-and-bound approach. It accomplishes this by converting the input application to a graphical representation that specifies all possible ways of implementing the application on target processor. A detailed register allocation is carried out in the second step, whereas its estimates are generated in the first step considering insertion of loads and spill over to memory if the available resources are exceeded. Their approach is shown in Fig. 9. The compiler front end receives a source program written in C/C++. It performs machine independent optimizations and generates an intermediate format description in *SUIF*. The instruction set description language (ISDL) description of the target processor is generated either by hand or by a high-level CAD tool. The compiler back end takes the *SUIF code* and the *ISDL description* as inputs and produces assembly code specific to, and optimized for, the target processor. The ISDL description is also used to create an assembler which transforms the code produced by the compiler to a binary file that is used as input to an instruction-level simulator. Visser [37] used simulated annealing technique for code generation. This technique focuses especially on highly irregular DSP architectures. It fully tackles the compiler phase-ordering problem. Phase-ordering problem is how to order various phases of code generation, because when one phase is performed earlier to other we loose flexibility for other phase, which might produce better code. For example if code selection is performed before register allocation then we have limited flexibility for register allocation which may not give the best results. The above approach consider all the subproblems together to resolve phase-ordering problem.

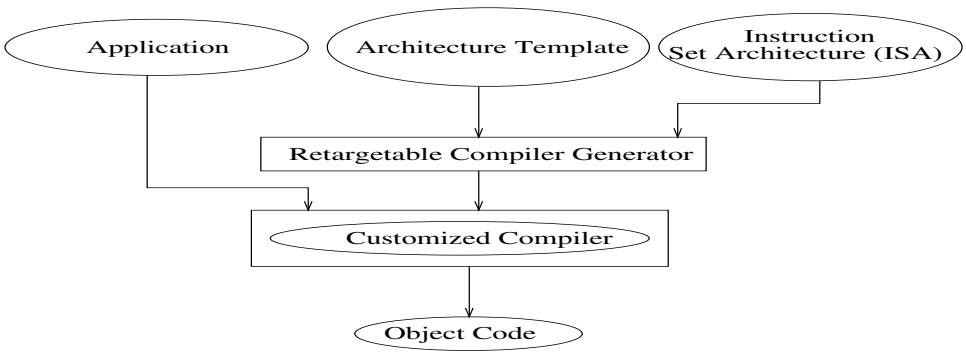

Figure 10: Retargetable Compiler

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. 10). Examples of such approaches are [14], [23] and [30]. 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 [14] 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. Its important features of their system include compact machine description via operator factoring and leaf predicates, fast production of time and efficient code generators. The System supports arbitrary evaluation orders and goal oriented code generation. Kuroda et al [23] proposed a knowledge-based compiler for application specific signal processors. Code optimization knowledge employed by expert programmer is implemented in the knowledge-base and is applied in various phases of the compiler. The generated retargetable compiler *RECORD* by Leupers at al [30] does not require tool-specific modeling formalism, but starts from general HDL processor models. All processor aspects needed for code generation are automatically derived. Table 4 compares various code synthesis approaches.

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 in the synthesis process. Performance estimation is based either on scheduler based or simulation based technique. Instruction set is generated either through synthesis or a selection process, whereas, 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 still 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.

Table 4: Comparison of major code synthesis approaches

Approach → Attribute ↓	Hatcher [14]	Kuroda [23]	Saghir [34]	Cheng [6]	Wilson [38]	Leupers [25, 26, 27, 28]	Kreuzer [21, 22]	Praet [33]	Gebotys [7]	Yamaguchi [39]	Hanono [13]	Visser [37]
Code generator/ Compiler generator	compiler generator	compiler generator	code generator	code generator	code generator	code generator	code generator	code generator	code generator	code generator	code generator	code generator
Instruction mapping	tree pattern matching					tree pattern matching	tree pattern matching	graph pattern matching	yes		yes	yes
Resource allocation and binding	yes		yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
Scheduling	yes	yes		yes	yes		yes	yes	yes	yes	yes	yes
Code optimization	yes	yes	yes	yes	yes	yes	yes	yes	yes		yes	yes
Register spill over					yes				yes	yes		
Technique/ Algorithm	tree pattern matching				ILP						branch-and-bound	simulated annealing
Retargetability	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
Instruction level parallelism				yes			yes				yes	
Phase-ordering							yes					yes
Distinguishing features	arbitrary evaluation order	uses experts knowledge	need for kernel synthesis			compaction based on a heuristic scheduler, behavioural/ structural target arch., address assignment			scheduling followed by instruction mapping		mapping, binding and scheduling concurrent, detailed reg. allocation at second step	phase-coupling

References

[1] Alomary, A.; Nakata, T.; Honma, Y.; Imai, M.; Hikichi, N. : "An ASIP instruction set optimization algorithm with functional module sharing constraint.", Proceedings of the IEEE/ACM International Conference on Computer Aided Design 1993. ICCAD-93. Digest of Technical Papers 1993 , 7-11 Nov. 1993, Pages: 526-532.

[2] Binh, N.N.; Imai, M.; Shiomi, A.; Hikichi, N. : "A hardware/software partitioning algorithm for pipelined instruction set processor.", Proceedings of the Design Automation Conference, 1995, with EURO-VHDL, EURO-DAC '95, 18-22 Sept. 1995, Pages: 176-181.

[3] Binh, N.N.; Imai, M.; Shiomi, A. : "A new HW/SW partitioning algorithm for synthesizing the highest performance pipelined ASIPs with multiple identical FUs.", Proceedings of the Design Automation Conference, 1996, with EURO-VHDL '96 , EURO-DAC '96, 16-20 Sept. 1996, Pages: 126-131.

[4] Binh, N.N.; Imai, M.; Takeuchi, Y. : "A performance maximization algorithm to design ASIPs under the constraint of chip area including RAM and ROM sizes.", Proceedings of the Asia and South Pacific Design Automation Conference 1998 (ASP-DAC '98), 10-13 Feb. 1998, Pages: 367-372.

[5] Childers, B.R.; Davidson J.W. : "Application Specific Pipelines for Exploiting Instruction-Level Parallelism.", University of Virginia Technical Report No. CS-98-14, May 1, 1998.

[6] Cheng W.; Lin Y. : "Code generation for a DSP processor.", Proceedings of the Seventh International Symposium on High-Level Synthesis 1994, 18-20 May 1994, Pages: 82-87.

[7] Gebotys, C.H. : "An efficient model for DSP code generation: performance, code size, estimated energy.", Proceedings of the Tenth International Symposium on System Synthesis 1997, 17-19 Sept. 1997, Pages: 41-47.

[8] Ghazal, N.; Newton, R.; Jan Rabaey. : "Retargetable estimation scheme for DSP architecture selection.", Proceedings of the Asia and South Pacific Design Automation Conference 2000 (ASP-DAC 2000), 25-28 Jan. 2000, Pages: 485-489.

[9] Gloria A. D.; Faraboschi, P. : "An evaluation system for application specific architectures.", Proceedings of the 23rd Annual Workshop and Symposium on Microprogramming and Microarchitecture. (Micro 23), 27-29 Nov. 1990, Pages: 80-89.

[10] Gong J.; Gajski, D.D.; Nicolau, A. : "Performance evaluation for application-specific architectures.", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Dec. 1995 Vol. 3 Issue: 4, Pages: 483-490.

[11] Gschwind, M. : "Instruction set selection for ASIP design.", Proceedings of the Seventh International Workshop on Hardware/Software Codesign 1999 (CODES '99), 3-5 May 1999, Pages: 7-11.

[12] Gupta, T.V.K.; Sharma, P.; Balakrishnan, M.; Malik, S. : "Processor evaluation in an embedded systems design environment.", Proceedings of Thirteenth International Conference on VLSI Design 2000, 3-7 Jan. 2000, Pages: 98-103.

[13] Hanono, S.; Devadas, S. : "Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator.", Proceedings of the Design Automation Conference 1998, 15-19 June 1998, Pages: 510-515.

[14] Hatcher, P.J.; Tuller, J.W. : "Efficient retargetable compiler code generation.", Proceedings of the International Conference on Computer Languages 1988, 9-13 Oct. 1988, Pages: 25-30.

[15] Hoon Choi; In-Cheol Park; Seung Ho Hwang; Chong-Min Kyung : "Synthesis of application specific instructions for embedded DSP software.", Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers 1998, 8-12 Nov. 1998, Pages: 665 - 671.

[16] Huang, I.J.; Despain, A.M. : "Generating Instruction Sets And Microarchitectures From Applications.", Proceedings of the IEEE/ACM International Conference on Computer-Aided Design 1994, 6-10 Nov. 1994, Pages: 391-396.

[17] Huang I.J.; Despain, A.M. : "Synthesis of application specific instruction sets.", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, June 1995 Vol. 14 Issue: 6 Pages: 663-675.

[18] Imai, M.; Alomary, A.; Sato, J.; Hikichi, N. : "An integer programming approach to instruction implementation method selection problem.", Proceedings of the European Design Automation Conference, 1992., EURO-VHDL '92, EURO-DAC '92, 7-10 Sept. 1992, Pages: 106 - 111.

[19] Kienhuis, B.; Deprettere, E.; Vissers, K.; van der Wolf, P. : "The construction of a retargetable simulator for an architecture template.", Proceedings of the Sixth International Workshop on Hardware/Software Codesign 1998 (CODES/CASHE '98), 15-18 March 1998, Pages: 125-129.

[20] Kin, J.; Chunho Lee; Mangione-Smith, W.H.; Potkonjak, M. : "Power efficient mediaprocessors: design space exploration.", Proceedings of the 36th Design Automation Conference 1999, 21-25 June 1999, Pages: 321-326.

[21] Kreuzer, W.; Gotschlich, M.; Wess, B. : "A retargetable optimizing code generator for digital signal processors.", Proceedings of the IEEE International Symposium on Circuits and Systems 1996. ISCAS '96., Connecting the World., 1996, Vol. 2, Pages: 257-260.

[22] Kreuzer, W.; Wess, B. : "Cooperative register assignment and code compaction for digital signal processors with irregular datapaths.", Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, 1997. ICASSP-97, 21-24 April 1997, Vol. 1, Pages: 691-694.

[23] Kuroda, I.; Nishitani, T. : "A knowledge-based retargetable compiler for application specific signal processors.", Proceedings of the IEEE International Symposium on Circuits and Systems 1989, 8-11 May 1989, Vol. 1, Pages: 631-634.

[24] Leupers, R.; Schenk, W.; Marwedel, P. : "Retargetable assembly code generation by bootstrapping.", Proceedings of the Seventh International Symposium on High-Level Synthesis 1994, 18-20 May 1994, Pages: 88-93.

[25] Leupers, R.; Niemann, R.; Marwedel, P. : "Methods for retargetable DSP code generation.", Proceedings of the International Workshop on VLSI Signal Processing, VII, 1994, 26-28 Oct. 1994, Pages: 127 - 136.

[26] Leupers, R.; Marwedel, P. : "A BDD-based frontend for retargetable compilers.", Proceedings of the European Design and Test Conference, 1995. ED&TC 1995, 6-9 March 1995, Pages: 239-243.

[27] Leupers, R.; Marwedel, P. : "Time-constrained code compaction for DSPs.", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, March 1997, Vol. 5, Issue: 1, Proceedings of the International Conference on Very Large Scale Integration (VLSI) Systems 1995, 18-20 Jan. 1995, Pages: 112-122.

[28] Leupers, R.; Marwedel, P. : "Algorithms for address assignment in DSP code generation.", Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers 1996, 10-14 Nov. 1996, Pages: 109-112.

[29] Leupers, R.; Marwedel, P. : "Instruction selection for embedded DSPs with complex instructions.", Proceedings of the European Design Automation Conference, 1996, with EURO-VHDL '96, EURO-DAC '96, 16-20 Sept. 1996, Pages: 200-205.

[30] Leupers, R.; Marwedel, P. : "Retargetable generation of code selectors from HDL processor models.", Proceedings of the European Design and Test Conference, 1997. ED&TC 97, 9-13 June 1997, Pages: 534-539.

[31] Liem, C.; May, T.; Paulin, P. : "Instruction-set matching and selection for DSP and ASIP code generation.", Proceedings of the European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, 28 Feb.-3 March 1994, Pages: 31-37.

[32] Praet J.V.; Goossens, G.; Lanneer, D.; De Man, H. : "Instruction set definition and instruction selection for ASIPs.", Proceedings of the Seventh International Symposium on High-Level Synthesis 1994, 18-20 May 1994, Pages: 11-16.

[33] Praet, J.V.; Lanneer, D.; Goossens, G.; Geurts, W.; De Man, H. : "A graph based processor model for retargetable code generation.", Proceedings of the European Design and Test Conference 1996 (ED&TC 96), 11-14 March 1996, Pages: 102-107.

[34] Saghir, M.A.R.; Chow, P.; Lee, C.G. : “Application-driven design of DSP architectures and compilers.”, Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing 1994 (ICASSP-94), 19-22 April 1994, Pages: II/437-II/440 Vol.2.

[35] Sato, J.; Imai, M.; Hakata, T.; Alomary, A.Y.; Hikichi, N. : “An integrated design environment for application specific integrated processor.”, Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors 1991, ICCD ’91, 14-16 Oct. 1991, Pages: 414-417.

[36] Shu, J.; Wilson, T.C.; Banerji, D.K. : “Instruction-set matching and GA-based selection for embedded-processor code generation.”, Proceedings of the Ninth International Conference on VLSI Design 1996, 3-6 Jan. 1996, Pages: 73-76.

[37] Visser, B.-S. : “A framework for retargetable code generation using simulated annealing.”, Proceedings of the 25th EUROMICRO Conference 1999, 8-10 Sept. 1999, Pages: 458-462.

[38] Wilson, T.; Grewal, G.; Halley, B.; Banerji, D. : “An integrated approach to retargetable code generation.”, Proceedings of the Seventh International Symposium on High-Level Synthesis 1994, 18-20 May 1994, Pages: 70-75.

[39] Yamaguchi, M.; Ishiura, N.; Kambe, T. : “Binding and scheduling algorithms for highly retargetable compilation.”, Proceedings of the Asia and South Pacific Design Automation Conference 1998 (ASP-DAC ’98), 10-13 Feb. 1998, Pages: 93-98.