CS812 Term Paper : ASIP Synthesis

Ravi Krishna
csd98422
Vipul Jain
csd98431

April 18, 2002

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 Processor (ASIP) is a processor designed for a particular application or for a set of applications. It has a custom instruction set and hardware which exploit the special characteristics of application(s) to meet the desired performance, cost and power requirements.

According to Liem, ASIPs are a balance between two extremes : ASICs and general programmable processors. Hoon 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 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 survey some of the work done on ASIP design and synthesis. Important steps involved in the ASIP design process are identified.

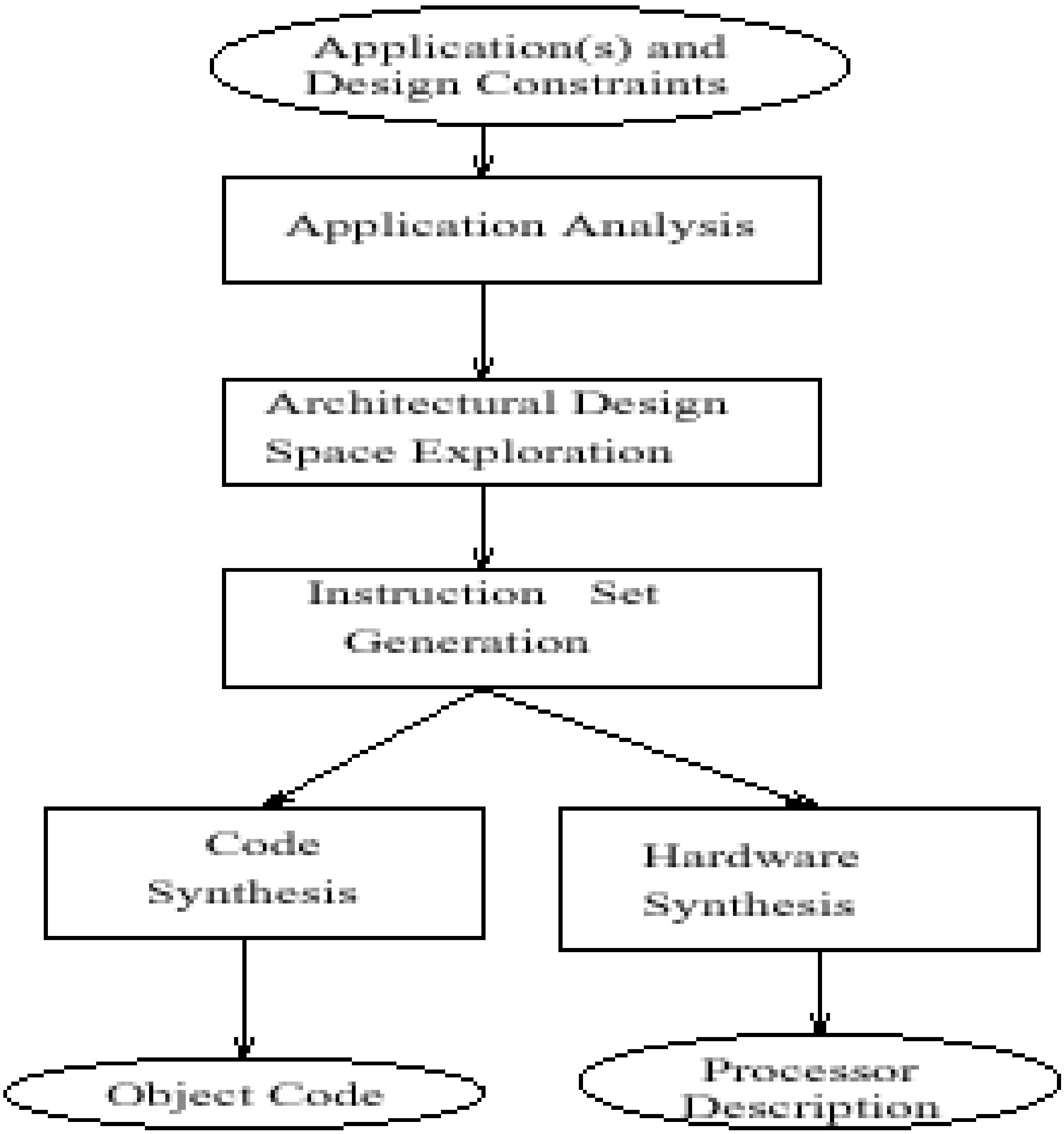

Figure 1: Steps in ASIP Synthesis

2 Steps in ASIP Synthesis

Gloria defined some main requirements of the design of application specific architectures. Important among these are as follows.

1. Design starts with the application behavior.
2. Evaluate several architectural options.
3. Identify hardware functionalities to speed up the application.
4. 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.

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.

Static analysis reports frequency of instructions and instruction sequences.Dynamic anlysis is runtime profiling of the program. Sato et al 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 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 .

Saghir 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 Archtectural Exploration

This step is more like a search where given the application parameters and a number of architectures one searches for the best possible.

As clear from the diagram its the performance estimator that drives the process. As we assign different values to the parameters the estimater evaluates the dependent parameters like delay,cost etc so that the explorer is driven in right direction.

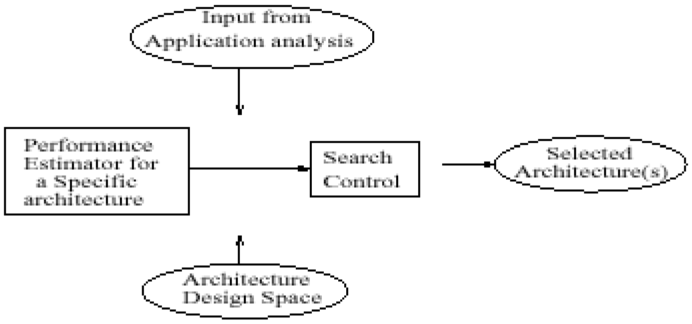

Figure 2: Architecture Exploration

4.1 Architectural Design Space

What we start with, is an architecture. The model should be parameterized and representative. Now the size of the architectural design space depends on number of parameters concerned and the range of values that they can take.

The parameterized architecture model suggested by almost all the techniques includes the number of functional units of different types.Gong consider storage units and interconnect resources also in their architectural model. Binh emphasize on incorporating pipelined functional units in the model. Kienhuis 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.

The other issues that have been looked at are issue width, number of branch units, number of memory units, the size of instruction cache and size of data cache etc by Kin.Gupta 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 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.All this leads to very large design spaces. Retargetable estimater has been developed for this.

Some researchers have also considered VLIW and Superscalar cases. The VLIW is very natural to the ASIP's because we can use each unit in each instruction.

Only very basic memory model has been considered.Only a few have been found to consider instruction and data caches.

4.2 Performance Estimation

Performance estimation is like evaluating the objective function. There are two main methods for this.

4.2.1 Schedular Based

Here estimation is looked upon as a resource constrained scheduling problem. We try to schedule the operation as soon as possible and thus get the cycle count needed for the whole operation.The profiling data is used to get the frequency of various operations. Gong in their approach also consider interconnects as a constrained resourced.They use the following three step process.

1. Conversion of C program to three address code They have customized gcc to generate RISC like three address code.All work is using registers only memory instructions being load/store.

2. Extraction of a fine grained parallelism A parallelizing compiler which is used to extract program parallelism is a finegrain 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.

3. Scheduling It starts with the result of the last step.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.

There are several other appoaches also.

- Gupta use resource constrained scheduling 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.

- Ghazal in their work have considered several optimizing transformations like optimized multi-operation pattern matching (e.g. multiply-accumulate), address mode optimization (e.g. autoupdate, circular buffer), loop optimization (e.g. prefetched sequential loads), loop vectorization/ packing, simple if else conversion (by use of predicated instructions), rescheduling within basic blocks, loop unrolling, software pipelining etc.

Gupta and Ghazal make their performance estimation fast by considering the 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.

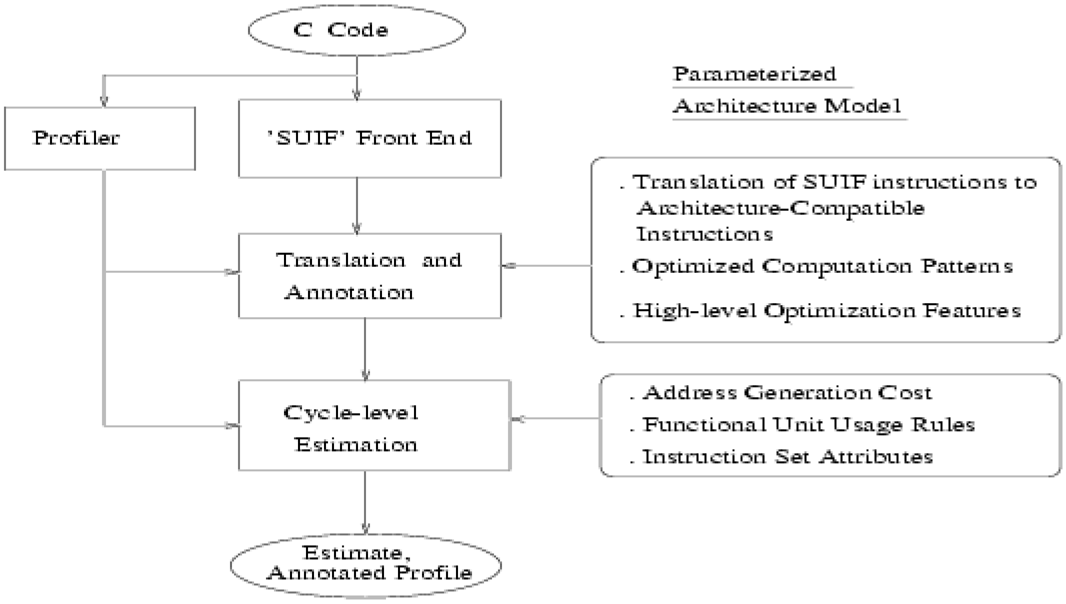

Figure 3: Performance Estimation

4.2.2 Simulator Based

Gloria have developed an event driven simulator.It uses predefined modules and architectural description to carry out the simulation and report the resource utilization statistics.The saliant features of his approach are the following

- translation into intermediate code with instructions representing simple HW functionalities
- Optimization techniques such as loop unrolling, datadependency 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.
- Optimization like replacing the frequently used instruction sequences with a single large instrucion

Another apporach due to Kienhuis considers a retargetable simulator based on architectural template.The architecture instance is constructed, an execution model is added and the executable architecture is instrumented with metric collectors to obtain performance numbers.

The next approach by Kin differs from the above as it considers the power consumption also. They use IMPACT compiler to create high instruction level parallelism in application. This is run over LSim to get performance numbers. The power estimation is done by a power model due to the same authors.

4.3 Search Control

Binh have suggested a branch and bound HW/SW partitioning algorithm based on branch and bound method.The consider gate count and power cosumption as the constraints.Later RAM/ROM sizes and Chip ares were added.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 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 have developed a processor selector. It takes several architectures and a small number of then is selected considering certain gross characteristics of the processors, such as branch-delay, presence/ absence of multiply-accumulate operation, depth of pipeline and memory bandwidths to get a smaller set.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.

Huang and Despain 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

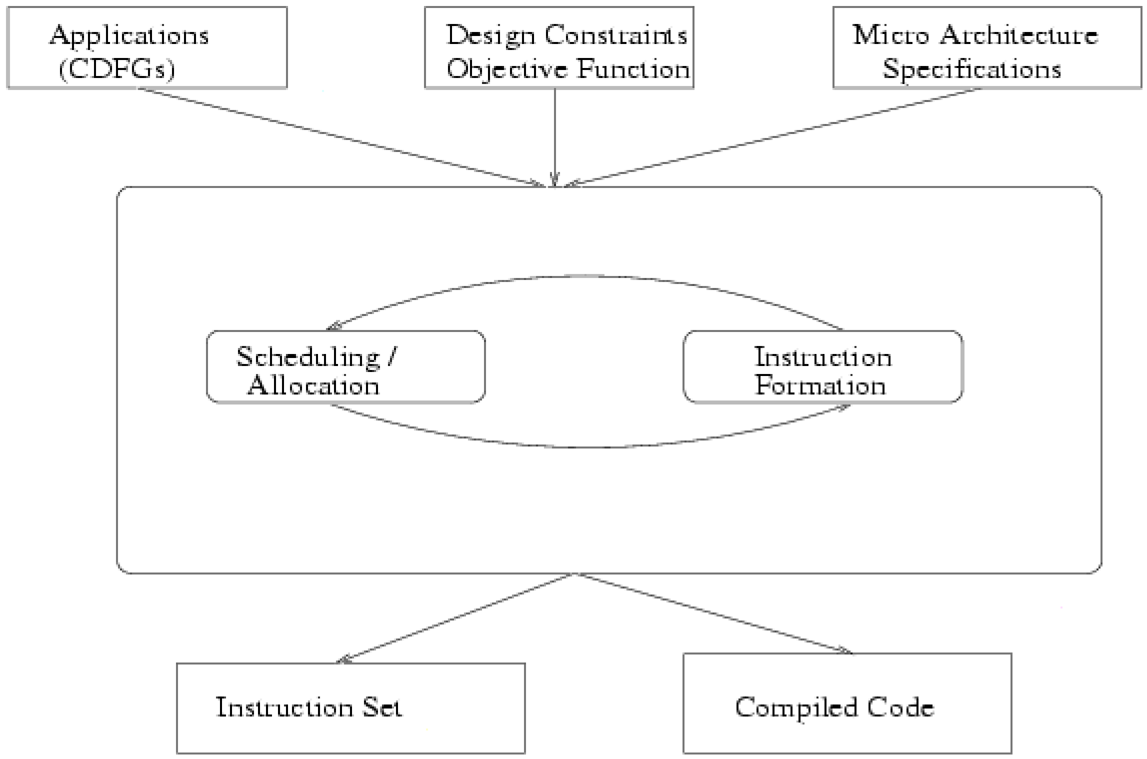

Figure 4: Instruction Generation

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 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.

6 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.All the approaches in this technique differ in the algorithms they are using to select the instructions from the super set. Imai 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.

Some approaches 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.

7 Code Synthesis

There are two different approaches for code synthesis. They are retargetable code generator and compiler generator.

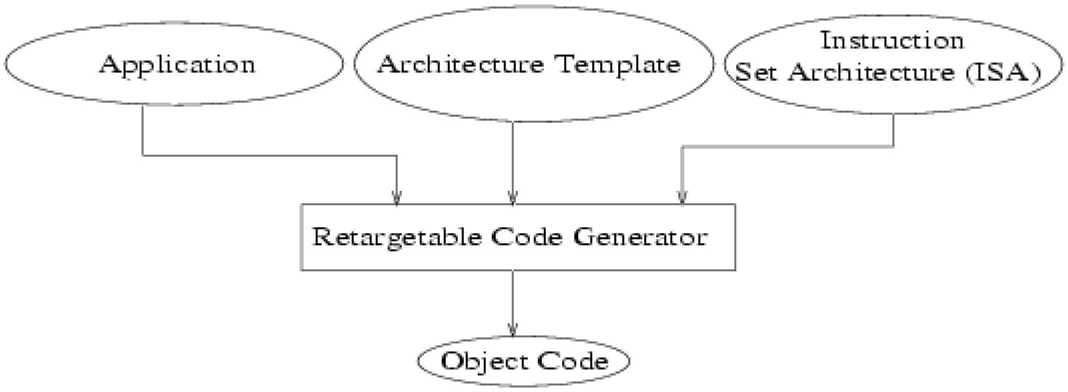

Figure 5: Retargetable Code Generation

7.1 retargetable code generator

Taking architecture template, instruction set architecture (ISA) and application as inputs, object code is generated.

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.

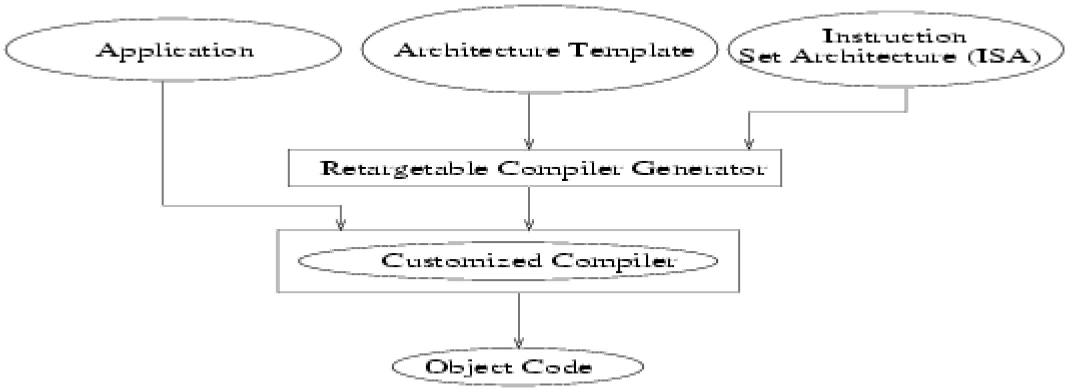

Figure 6: Compiler Generation

7.2 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.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 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 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 does not require tool-specific modeling formalism, but starts from general HDL processor models. All processor aspects needed for code generation are automatically derived.

8 Conclusion and future work

We have identified the five key steps in the ASIP synthesis process and tried to present the overview of the state of the art in the area.

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.

Besides the problems of processor synthesis and retargetable code generation have been considered in isolation.With developement of algorithms and processing capability we expect a greater integration of these steps to produce better results.

9 Acknowledgments

We acknowledge the guidance provided by Anshul Sir and Bala Sir. Besides Mr. Manoj Kr. Jain was of immense help in providing us with materials and explaining the details.

10 References

1. ASIP Design Methodologies : Survey And Issues by Manoj Kumar Jain, M. Balakrishnan and Anshul Kumar

2. ASIP Design Methodologies : Survey And Issues, brief paper by Manoj Kumar Jain, M. Balakrishnan and Anshul Kumar

3. Integrating Communication Cost Estimation in Embeded Systems Design : A PCI Case Study, by Anupam Rastogi,M. Balakrishnan and Anshul Kumar