設 計 自 動 化 60－1
（1991．12．12）

命令セット選択問題の整数計画法による解法

アラウディン　アロマリー*　　今井　正治*
佐藤　淳**　　引地　信之***

*）豊橋技術科学大学　情報工学系

）鶴岡工業高等専門学校　電気工学科　*）㈱ＳＲＡ

ＡＳＩＰ（Application Specific Integrated Processor:特定用途向き集積化プロセッサ）の命令セット・アーキテクチャおよびＣＰＵアーキテクチャを決定する場合、与えられたチップ面積（または最大ゲート数）および消費電力に関する制約条件のもとで、性能を最大化する命令セットを選択する必要がある。本論文では命令セット選択問題（Instruction Set Selection Problem:ISSP）を整数計画問題として定式化し、分枝限定法にもとづくアルゴリズムを提案する。このアリゴリズムをＣ言語で実現し実験を行なった結果、提案されたアリゴリズムはＩＳＳＰを効率良く解くことが知られた。

An Integer Programming Approach to Instruction Set Selection Problem in ASIP Design

Alauddin ALOMARY*, Masaharu IMAI*, Jun SATO**, and Nobuyoki HIKICHI***

* Toyohashi University of Technology
Department of Information and Computer Sciences

** Tsuruoka National College of Technology

*** Software Research Associates, Inc.

This paper proposes an algorithm for Instruction Set Selection Problem (ISSP) in ASIP design. This problem is to be solved in the instruction set architecture and CPU core architecture designs. First, the ISSP is formalized as an integer programming problem, which is to maximize the performance of the CPU under the constraints of chip area and power consumption. Then, a branch-and-bound algorithm to solve ISSP is described. Finally, some experimental results are illustrated. The experimental results show that the proposed algorithm is quite effective and efficient in solving the ISSP.

Keyphrases: ASIP, high-level synthesis, Instruction Set Selection Problem (ISSP), integer programming problem, branch-and-bound method.

1. Introduction

Due to the advance of VLSI technologies, it is now possible to design a very large scale ASIP (Application Specific Integrated Processor). Throughout this paper, we define the term "ASIP" as an application specific microprocessor which contains a CPU core, memory, and peripheral circuits. ASIPs can be very effective when applied to specific applications such as digital signal processing, servo motor control, etc. [SaKi90].

High-level synthesis will be one of the most effective methods to design ASIPs, as mentioned in the reference [RaWa91]. However, there are still some features to be enhanced to design a large scale ASIP. One is to assist the designs of instruction set architecture and CPU core architecture. And the other is to assist the realization of application program development environment, such as compiler and simulator [SaIm91].

This paper proposes an algorithm for Instruction Set Selection Problem (ISSP) in ASIP design. This problem is to be solved in the instruction set architecture and CPU core architecture designs. First, the ISSP is formalized as an integer programming problem, which is to maximize the performance of the CPU under the constraints of chip area and power consumption. Then, a branch-and-bound algorithm to solve ISSP is described. Finally, some experimental results are illustrated. The experimental results show that the proposed algorithm is quite effective and efficient to solve the ISSP.

2. Previous Work

A number of high-level synthesis systems have been proposed and are being used to design VLSI chips [DeNe87], [ThDi88], [CoRo89], [Phil89], [IkTa90], [RaWa91], [SaIm91]. High-level synthesis will be one of the most effective methods to design ASIPs. While high-level synthesis is quite effective, there are still some features to be enhanced. One is to assist the designs of instruction set architecture and CPU core architecture. And the other is to assist the realization of application program development environment, such as compiler and simulator.

In order to solve these problems, an integrated design environment for ASIPs named PEAS(Practical Environment for ASIP development) was proposed by the authors [SaIm91], which makes it possible to generate an ASIP hardware as well as application program development tools. The PEAS system consists of following four subsystems: **Application Program Analyzer, Architecture Information Generator, CPU Core Generator,** and **Application Program Development Tool Generator,** as shown in Figure 1. The outline of these subsystems are as follows:

(1) Application Program Analyzer

Application Program Analyzer (APA) statically and dynamically profiles a given set of application programs with corresponding data set. The output of APA includes: data types and their access methods, execution counts of operators and functions, etc., used in the application programs.

(2) Architecture Information Generator

Architecture Information Generator (AIG) is to receive the profiled results from APA, and to decide the "optimum" instruction set and hardware architectures of the ASIP under the constraints of chip area and power consumption. This task is performed by turning the instruction set and CPU core architecture, so that the performance of the ASIP could be statistically maximized regarding the given application programs and associated data set.

(3) CPU Core Generator

CPU Core Generator (CCG) will generate the CPU Core design in the form of an HDL, according to the architecture information generated by AIG. Then the actual CPU core design can be synthesized by a high-level synthesis system.

(4) Application Program Development Tool Generator

Application Program Development Tool Generator (DTG) will produce a set of software tools, such as a compiler, a debugger including instruction-level simulator, an assembler, a run-time library, etc. Some of these tools are generated by taking advantage of GNU software tools [Stal90]. The GNU related software includes the C compiler and the source-level debugger.

3. Instruction Set Architecture Model

3.1 Outline

The Architecture Information Generator (AIG) is supposed to select the "optimum" instruction set and hardware architecture. This task is performed by turning the instruction set and CPU core architecture, so that the performance of the yielded ASIP design could be statistically maximized under the constraints of chip area and power consumption. This optimization problem needs a solution that balances the implementation of operations and functions among hardware, software, and microprogram.

Throughout this paper, we assume that the instruction set of a generated ASIP mainly includes a subset of the instruction set that can be generated by the GNU C compiler. The reason behind this assumption is as follows. Suppose that an ASIP has an efficient instruction but the compiler does not generate a code using that instruction. Then, this instruction is absolutely redundant because this instruction will never be used by any application program.

3.2 Classification of Operators and Functions

In the C language, operators and functions are treated appearently distinguished. In the design of the Architecture Information Generator, we need to establish a new concept to treat them uniformly. In the following description, the term "functionality" is introduced. "Functionality" is defined as any one of the operators or the functions independently of their implementation.

The instruction set for ASIP generated by the PEAS system is supposed to include a subset of the instruction set that can be generated by GNU C compiler [Stal 91]. The compiler-generatable instruction set can be divided into two subsets: **operators** and **functions**. The reason is as follows: (1) It is easier for a C compiler to generate instructions corresponding to operators than to generate ones corresponding to functions. (2) While the set of operators can be defined clearly, the set of functions, including user-defined ones, cannot be given a priori.

We further divide the set of operators into two subsets: **primitive operators** and **basic operators**. The set of primitive operators is chosen so that any basic operators or functions can be realized by a series of primitive operators. Thus, we divide the instruction set into three classes as follows.

(1) Primitive Instruction Set

The Primitive Instruction Set **(PIS)** can be realized by a minimal hardware component as ALU and shifter. The instruction set of the ASIP includes all instructions in PIS.

(2) Basic Instruction Set

The Basic Instruction Set **(BIS)** includes the set of operators used in C language except those included in PIS. Instructions included in BIS can be implemented by hardware modules or microprogram.

(3) Extended Instruction Set

The Extended Instruction Set **(XIS)** includes instructions which correspond to library functions or user-defined functions. The instructions in XIS could be implemented by using complex hardware modules, such as coprocessor, or microprogram.

3.3 Hardware Architecture Model

The generated CPU architecture of an ASIP is based on the GCC's abstract machine model [Stal 91]. The intermediate language of GCC is called "RTL" (Register Transfer Language). Most instructions in RTL are included in either PIS or BIS. Then the intermediate instructions are classified into primitive RTL **(PRTL)**, Basic RTL **(BRTL)** and Extended RTL **(XRTL)**, corresponding to the PIS, BIS and XIS classification. The instructions included in PRTL, BRTL, and XRTL are shown in Table 1.

The generated ASIP will include hardware modules which correspond to all of the PRTL. But only a part of BRTL and XRTL will be implemented by hardware. The decision, which element in BRTL and XRTL should be implemented by which method, is made by AIG.Implementing the functionalities included in BRTL and XRTL by hardware modules will improve the chip performance. But this choice will increase the physical chip area. On the contrary, implementing most of BRTL and XRTL by microprogram or software will use more memory space and degrade the execution time.

Therefore, to maximize the efficiency of the generated ASIP, such a solution is required that balances the implementation of the instruction set among hardware choices, microprogram, and software, under the constraints of chip area and power consumption. This solution will give a statistically minimum execution time for the given application program.

4. Formalization

The ASIP instructions shown in Table 3 can be implemented by different methods. In order to maximize the performance of designed ASIP chip, a set of instructions with their implementation methods which minimize the average execution time for the given application should be chosen. In the rest of this paper, following definitions and notations are used.

(1) "n" denotes the number of functionalities.

(2) "N_i" denotes the number of possible implementation methods for functionality #i.

(3) "x_i" denotes an implementation method of functionality #i.

(4) "f_i" denotes the execution frequency count of functionality #i, which can be delivered by the APA.

(5) "$a_i(x_i)$" denotes the area required for functionality #i when implemented by method x_i.

(6) "$p_i(x_i)$" denotes the power consumption of functionality #i when implemented by method x_i.

(7) "$t_i(x_i)$" denotes the execution time of functionality #i when implemented by method x_i.

(8) "A_max" denotes the available chip area for computing modules.

(9) "P_max" denotes the upper limit of the chip power consumption.

The instruction execution frequency of each instruction can be obtained from the static and dynamic analysis of the application programs done by APA. Suppose that the upper bounds of chip area and power consumption are provided, the ISSP can be formalized as an integer programming problem shown in Figure 2.

According to the experience, about 40 to 60% of the total chip area is used for wiring. Therefore, the parameter A_max denotes the rest (40 to 60%) of the total chip area, which can be used for computational modules and memories. This constraint is effctive when a cell-based technology is used for fabrication. In the case where a gate-array or an SOG (Sea of Gate) technology is used, it would be more reasonable and effective to use the following constraint C1' instead of Constraint C1 in Figure 2.

$$\text{Constraint C1':} \quad \sum_{i=1}^{n} g_i(x_i) \leq G\_max$$

Where, $g_i(x_i)$ denotes the number of gates (or basic cells) needed to implement the instruction i by method x_i. And G_max denotes the number of available gates (or basic cells).

Note also that the number of implementation methods can be more than three, i.e., hardware, microprogram and software. For some instructions, there could exist several hardware choices. For example, "mul" (multiply) instruction can be implemented by the "sequential add-shift multiplier," "modular array multiplier," or "bit-pair recording multiplier." In this case, the total number of implementation methods for instruction "mul" will be five.

5. ISSP Solver

5.1 Method

The ISSP, described in the previous section, is NP-hard, which means that the worst case computation time would be exponential to the size of input (the number of functionalities to be considered). Therefore, an efficient approach should be used to solve ISSP in reasonable computation time. The branch-and-bound method is known as one of the most effective methods to solve such an intractable problem.

The effectiveness of the branch-and-bound methods has been investigated by many researchers [FiTo89]. Because the branch-and-bound method is simple and cost effective, it was chosen to solve the ISSP.

5.2 Input and Output

The input of the algorithm includes the following values:

(1) f_i's for i = 1, ..., n,

(2) parameter A_max, and

(3) parameter P_max.

The output of the algorithm is the implementation method for each functionality. The chosen implementation method is optimum in the sense that the solution will yield the minimum execution time of the given application programs under the given constraints. Regarding the constraints, there are two cases to be considered.

(1) If there exists any feasible solution which satisfies the given constraints, the algorithm decides the implementation methods for the functionalities.

(2) When the constraints are so tight that no feasible solution exists, no implementation method would be assigned to any functionality.

5.3 Description

The task of finding the optimal instruction set is viewed by the algorithm as finding a node in a search tree. Each node in the search tree represents a partial solution or a complete solution if it is a leaf node. Each node corresponds to particular implementation methods for part of functionality. The number of branches for the node depends on the number of possible implementation methods for an instruction. Each node can be identified by the following attributes:

(1) Node cost (Cost): partial sum of the objective function $f_i * t_i(x_i)$ for a particular node.

(2) The power gain (P_gain): partial sum of the power consumption of the hardware modules for a particular

node.

(3) The area gain (A_gain): partial sum of the area of the of the hardware modules for a particular node.

(4) Currently selected instructions (Sol): The functionalities and their chosen implementation methods included in for a particular node.

(5) Depth index (d): The depth where instruction i is under consideration. (1, 2, ... , d-2, d-1: already considered, d+1, .., n : not yet considered)

(6) Lower_bound (Lowb): The lower bound of the current node which is equal to :

$$\text{node cost} + \sum_{j=i+1}^{n} f_j * \min\{t_j(x_j) \mid x_j \in N_j\}$$

The description of the branch-and-bound algorithm is shown in Appendix A. The procedure "Reorder" sorts the variables (functionalities) in a descending order according to a heuristic function h. This process is essential to improve the efficiency of the algorithm by reducing the search space in the whole search tree.

The procedure "Search" looks for an optimum solution of the given problem. This procedure is based on the depth-first strategy, which visits the node in the search tree according to the depth-first manner in the "for" loop in this procedure. Each time a node is generated, an functionality #i implemented by method x_i is added to the solution list. Then the cost, A_gain, P_gain and Lower bound are calculated according to the newly added functionality. If the lower bound exceeds the cost of the currently best solution (C_opt), then the subproblem (node) is pruned. Every time a leaf node is reached, the cost of the current solution is compared with the best solution so far found. If the current solution is better than that so far found, current solution is updated. Procedure "Lower_bound" calculate the lower bound for each subproblem.

5.4 Heuristic Functions

Several heuristic functions have been considered and examined to choose the most suitable one to be used in the system. Three reasonable heuristic functions among others are as follows:

$$h_1(i) = f_i * \max \{ t_i(j) \mid 1 \le j \le N_i \}$$

$$h_2(i) = f_i * \min \{ t_i(j) \mid 1 \le j \le N_i \}$$

$$h_3(i) = f_i * \frac{\max \{ t_i(j) \mid 1 \le j \le N_i \}}{\max \{ a_i(j) \mid 1 \le j \le N_i \}}$$

These heuristic functions will try to improve following instructions:

(1) h_1 will try to implement first such functionality by hardware that takes the longest computation time when it is implemented by software library.

(2) h_2 will try to implement first the functionality by hardware that takes the shortest computation time when it is implemented by the fastest hardware.

(3) h_3 will try to implement first the functionality by hardware which will be the most cost-effective when it is implemented by software library.

6. Experiments and Results

6.1 ISSP solver Implementation:

A prototype of the algorithm was written in C language and runs on Sun-3/60 workstation (about 3 MIPS as fast).

6.2 Database

The data of the functionality implementation to be fed to the program was generated using the Structured Function description Language (SFL). The SFL is the HDL used in a high-level synthesis system called PARTHENON [RaWa91].

Various hardware modules were designed and evaluated using PARTHENON to generate the database of the ASIP functionalities. The database contains 18 PRTL functionalities and 24 BRTL functionalities with a total implementations of 50. The whole search tree consist of $3.35*10^7$ nodes.

6.3 Sample Application Programs

The instruction set selection method adopted in this paper is tested using four application program samples:

(1) Servo-motor current PI control program.

(2) The solution of tan(x) = x using the Newton method.

(3) The integral of $x^2 * \exp(-x^3)$ using the Simpson method.

(4) Multiple virtual terminal program.

6.4 Heuristic Consideration

A good heuristic function should be effective and stable for a large class of problems, which means that it should reduce the search space effectively

even if the size of given problem changes in a wide range. It was known from the experimental results that heuristic function $h_1(i)$ is much more effective and stable compared to other heuristic functions $h_{2(i)}$ and $h_3(i)$.

6.5 Designed ASIP example

In this experiment, the ISSP solver was applied to design the functionalities of a hypothetical ASIP chip. The information obtained from analysis of the four application programs mentioned in section 6.3 were fed to the ISSP solver, which selects the optimal implementations for these application programs. This design method will automate a complex part of the ASIP chip design and enable the designer to estimate the performance of the hypothetical ASIP design chip.

Figure 3 shows the optimal functionality execution speed measured in Million Functionality Per Second (MFPS) versus the area constraint (in gates) for the four application programs. From this figure, it is known that when the area is small, MFPS is small. This is because all the functionalities (except the PRTL) is implemented by software. As the area increases, MFPS increases drastically since more instructions are implemented by hardware. A saturation condition is reached when all the functionalities are implemented by hardware.

Because we assume that the system clock is 10 MHz, the maximum MFPS is 10, which should be obtained if all functionalities are implemented by hardware. However, since the fastest hardware divider in the database needs 17 clocks, the maximum case can not be reached in the applications that uses divider. These are PI controller, Newton and Simpson. The figure shows the effectiveness of the ISSP solver in obtaining the optimal set of functionalities that maximize the performance of ASIP for the four application programs under the constraint of chip area.

The PI control program uses heavily the divider modules, therefore it has the smallest MFSP among the four samples. On the contrary, the virtual terminal program doesn't use the divider module, therefore the maximum MFPS is reached in this case.

The ISSP solver investigated a small search space compared with the total search space (about $1/10^6$ in average) of the problem. Also the ISSP solver gives the optimal solution of the problem in less than 1 second. This is the result of the good heuristic function and the implemented tight lower bound.

7. Conclusion

This paper formalizes the instruction set implementation method selection problem (ISSP) for ASIP design. Then, a branch-and-bound algorithm was proposed to solve the problem. Several heuristic functions were tested to find a effective and stable one. Finally, the efficiency of the proposed algorithm has been investigated. According to the experimental results, the proposed algorithm was found to be able to solve ISSP in a reasonable computation time. The algorithm will be used as a part of the architecture information generator (AIG) of the integrated design environment for ASIPs.

Acknowledgments

Authors would like to express their thanks to the members of VLSI Design Laboratory of Toyohashi University of Technology, especially Chan Namkue, Takeharu Nakata, and Yoshimichi Honma for their helpful comments.

References

[CoRo89] Composano, R., and Rosenstiel, W.:"Synthesizing Circuits from Behavioral Descriptions," IEEE, Trans. on CAD, Vol. 8, No.2, pp. 171-180, Feb. 1989.

[DeNe87] Devadas, S., and Newton, A.R.: "Algorithms for Hardware Allocation in Datapath Synthesis", IEEE, Trans. on CAD, Vol. 8, No. 7, pp.768-781, Jul. 1987.

[FiTo89] Fischetti, M., and Toth, P.: "An additive Bounding Procedure for Combinatorial Optimization Problems," Operations Research, Vol. 37, No.2, March-April 1989.

[IkTa90] Ikenaga, T., Takezawa, T., and EvaluationShirai, K.: "Design and of Special Purpose Procesor Executing Multiple Algorithms," Proc. of IPSJ (Information Processing Society of Japan) 40th National Convention, pp. 1283-1284, Mar. 1990, (in Japanese).

[Phil89] Philipson, L., et al.: "A Seven-Week Microprocessor Design Project Based on High-Level Tools," Proc. of the 1989 Decennial Caltech Conference, pp. 209-226, 1989.

[RaWa91] Raul, C., and Wayne, W.: High-Level VLSI Synthesis, Kluwer Academic Publishers, 1991.

[SaFu89] Sato, J., Fukuda, K., and Imai, M.: "Study on an Applicaion Specific CPU Core Synthesis Method," IEICE Tech. Rep., VLD89-70, pp. 1-8, 1989, (in

Japanese).

[SaKi90] Sato, J., Kimura, T., Imai, M., et al.: "The Architecture of a Flexible Servo Motor Control Processor: FSP-3," Trans. of IEICE, Vol. E73, No. 4, pp. 513-515, Apr. 1990.

[SaIm91] Sato, J., Imai, M., Hakata T., Alomary, A., and Hikichi, N.: "An Integrated Design Environment for Application Specific Integrated Processor," Proc. of ICCD'91, pp. 414-417, Oct., 1991.

[Stal91] Stallman, R.: Using and Porting GNU CC, Free Software foundation, Version 1.4, 1991.

[Step89] Stephen, B.: VLSI RISC Architecture and Organization, Marcel Dekker. Inc., 1989.

[ThDi88] Thomas, D.E., Dirkes, E.M., et al.: "The System Architect's workbench" Proc. of 25th DAC, pp. 337-343, June 1988.

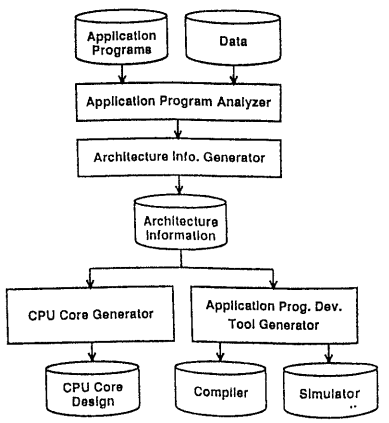

Fig. 1 PEAS System Configuration

Table 1: PEAS Functionalities

class	category	instruction
PRTL	arithmetic	add sub and ior xor one_cmpl neg ashl ashr lshl lshr
	transfer	mov
	control	jmp beq nop
BRTL	arithmetic	mul umul div mod udiv umod cmp cmpstr tst trunc extend zero_extend float floatuns ftrunc fix fixuns fix_trunc fixuns_trunc
	transfer	movstrict movstr (mov)
	control	call call_value return bne bgt bgtu blt bltu bge bgeu ble bleu casesi tablejump
	others	extv etxzv insv seq sne sgt sgtu slt sltu sge sgeu sle sleu
XRTL		abs sqrt ffs sin cos tan etc. user-defined functions

Find a solution vector

$$X=(x_1,x_2,\ldots,x_n)$$

which minimizes the objective function:

$$T(X) = \sum_{i=1}^{n} f_i * t_i(x_i)$$

subject to

Constraint C1: $\sum_{i=1}^{n} a_i(x_i) \leq A\_max$,

and

Constraint C2: $\sum_{i=1}^{n} p_i(x_i) \leq P\_max$,

where $1 \leq x_i \leq N_i$, for i = 1, 2, .. ,n.

Figure 2 – The Formalization of ISSP.

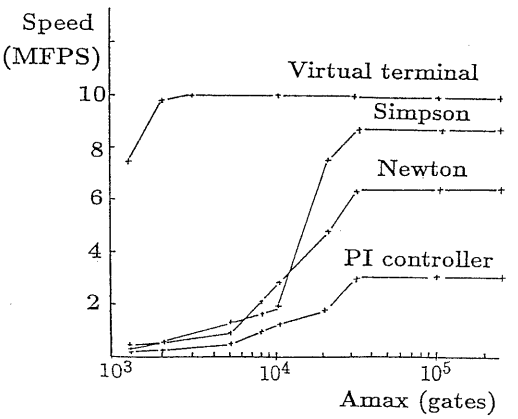

Fig.3

Appendix A (ISSP_Solver)

```
prodecure ISSP_Solver ( { find an optimum solution for ISSP }
{ input }
     fi : array [1..n] of real;        { execution frequencies }
     ai : array [1..n, 1..Ni_Max] of real;    { area for module }
     pi : array [1..n, 1..Ni_Max] of real;    { power consumption }
     Ni : array [1..n] of integer;    { # of implementations }
     A_max, P_max : real;      { constraints for area and power }
{ output }
     var Opt_sol : array [1..n] of integer;  { optimum solution }
     var C_opt : real );       { cost of the optimum solution }
const
     INFINITY = 999999;  { very big number }
     NULL = 0;           { no method selected }
var
     Sol : array [1..n] of integer;  { current solution }


     procedure Search (   { depth-first search }
          i : integer;     { instruction under consideration }
          A_gain : real;   { current area }
          P_gain : real;   { current power }
          cost : real );   { current cost }
          lowb : real );   { lower bound }
     var
          j, m : integer; { loop control variable }
     begin
          if ( A_gain <= A_max ) and ( P_gain <= P_max ) and
              ( lowb < C_opt ) then
               { constraints satisfied, promissing status }
               if ( i = n ) then { leaf node }
                    begin { better solution is found }
                    C_opt := cost; { update optimum solution }
                    for j := 1 to n do
                         Opt_sol[j] := Sol[j]
                    end { then }
               else { non-leaf node, expand }
                    for m := 1 to Ni[i] do begin
                     new_cost := cost + fi* ti[i][m]
                     Sol [i] := m; { choose m-th method }
                         { depth-first search }
                         Search ( i + 1, A_gain + ai[i,m],
                         P_gain + pi[i,m],
                         new_cost, lower_bound (new_cost, i))
                    end { for m }
          else;    { the status violates constraints }
                    { or not promising, prune this node }
     end; { procedure Search }

begin { procedure ISSP_solver }
     C_opt := INFINITY;        { initialization }
     for i := 1 to n do
          Opt_sol [i] := NULL;
     Reorder( )
     Search ( 1, 0.0, 0.0, 0.0, 0.0 )
end;  { procedure ISSP_solver }
```