A New Methodology for Object-oriented ASIP Architecture Codesign

I-HORNG JENG and FEIPEI LAI
Dept. of Electrical Engineering &
Dept. of Computer Science and Information Engineering
National Taiwan University
Taipei 106, Taiwan, R.O.C.
paul@orchid.ee.ntu.edu.tw, http://archi1.ee.ntu.edu.tw

Abstract: - Due to the developments of complex structures, the computer industry urgently needs a higher-level design hierarchy. Hardware/software codesign hence becomes a new trend for industrial applications. The paper presents a new approach to codesign ASIP efficiently. Firstly, based on a linear algebraic model, we explore the design space. The model provides a transformation algorithm to combine, separate, evaluate and verify instructions between software and hardware. Then, the model exploits object-oriented concept to build up its simulation mechanism. The simulator does executions, statistics and transformations for variant applications. Not only instruction-level parallelism but also multiprocessing synchronization applications easily apply this model. Finally, we make an example experiment environment and illustrate the results to show the optimizations.

Key-Words: - linear transformation, object-oriented, instruction-level parallelism

1 Introduction

Historically, the evolution of hardware-design entries shifts gradually: First it was polygons (physical layout), then netlist (gate-level), then HDL code (behavioral synthesis). In the same matter, the developmental process of software has also been from low-level to high-level. However, the developments of software not only seldom regard the impact from hardware architecture but also often neglect many potential factors, and vice versa. It is quite clear that performance is strongly interdependent on hardware and software.

Hardware/software codesign has been a challenge for industrial applications. No doubt, in the next five to ten years, there will be more research moving the techniques into industrial use through CAD tool development and reaching their full potential [1]. Either emphasizing on design space exploration or design automation, they build up partitioning models and abstractions. Collecting all the necessary parameters to make a best partition is common to all approaches.

The partition problem for ASIP (Application Specific Integrated Processor) codesign becomes more and more complex. Comparing with the past mature ASIP codesign like Sato et al. [2] and Huang et al. [3], they use either top-down or bottom-up instruction-set generating method. However, both are lack of large benchmark's behavior information. Moreover, their simulators are not compiled by mature CAD tools and seem not accurate enough in hardware cost estimation.

There exist some related work about object-oriented computer engineering [4] [5]. Our object-oriented simulation mechanism has some similar structure properties but totally different behaviors with [5]. In [4], they propose a tool MOOSE (Model-based Object Oriented Systems ngineering) to solve application-specific problems like pump and video controls.

Here we create a new linear-space model and an object-oriented mechanism for efficient partition. In this model, the designer writes down only the relationship rules among the old/new instructions, we can get the prototyping efficiently. Through the object-oriented mechanism, we can do simulations, statistics, and optimizations for instruction set. Here we provide a simple method to build up instruction simulator, neither trace-driven nor general execution-driven. It can run large benchmarks even SPECint9x and make information statistics extracted from collected parameters. Based on this tested foundation [6], the designers can explore the whole ASIP space freely and make their partitioning decision earlier.

2 A New Model

We model ASIP design space as a linear space. In the linear space, addition and multiplication represent executing and branching of an instruction. Furthermore, two elemental theorems of basis and replacement explain respectively why Sato et al. [2] propose basic operations and why there exist various instruction sets.

Next we show an instruction-set linear space (IsV) over field (IsF). We express that the labels and instructions of a program can be modeled as sequence of numbers ∈ IsF and vectors ∈ ISV respectively.

2.1 Instruction-set field

An instruction-set field IsF is a sequence of numbers s_n with the following definitions of addition and multiplication:

$$\begin{aligned} \forall s_i, s_j \in IsF \quad \exists \quad & s_i \oplus s_j, s_i \circ s_j \in IsF \\ Zero : 0 \quad = \quad & (s_i \oplus s_{-i}) = (s_i \ominus s_i) \qquad (1) \\ One : 1 \quad = \quad & (s_i \circ s_i) \qquad (2) \end{aligned}$$

where i, j are real numbers and $\ominus$ is the inverse of $\oplus$. The commutativity, associativity and distributivity [7] of addition and multiplication are trivial and indifferent to the real-number subscripts. As for the physical meaning, the scalars themselves in IsF are meaningless except for the respective vectors in IsV. Assume with an instruction vector $\mathbf{x}$, scalar 0 means two inverse vectors ($s_i \circ \mathbf{x}$, $s_i \circ \ominus\mathbf{x}$) take zero effect. So results in NOP.

$$\begin{aligned} s_i \circ \mathbf{x} \oplus s_i \circ (\ominus\mathbf{x}) \quad = \quad & (s_i \ominus s_i) \circ \mathbf{x} \\ = 0 \circ \mathbf{x} \quad = \quad & 0 \Rightarrow NOP \qquad (3) \end{aligned}$$

Then scalar 1 means one vector $\mathbf{x}$ jumps to itself endlessly.

2.2 Instruction-set linear space

An instruction-set linear space IsV over a field IsF consists of an instruction set on which addition and scalar multiplication are defined so that eight conditions hold [7]. We show the four major conditions:

$$\begin{aligned} \mathbf{x} \oplus 0 \quad = \quad & \mathbf{x} \oplus NOP = \mathbf{x} \qquad (4) \\ \mathbf{x} \oplus (\ominus\mathbf{x}) \quad = \quad & (1 \ominus 1) \circ \mathbf{x} = 0 \qquad (5) \\ (a \circ b) \circ \mathbf{x} \quad = \quad & a \circ (b \circ \mathbf{x}) \qquad (6) \\ a \circ (\mathbf{x} \oplus \mathbf{y}) \quad = \quad & a \circ \mathbf{x} \oplus a \circ \mathbf{y} \qquad (7) \end{aligned}$$

The physical meaning of Eqn. (4)(5)(6) is trivial, and (7) means two different instructions execute at the same time stamp a in IsF. This is a foundation extended to represent parallelism as shown in subsection 4.2.

Using this model, the relationship rule (abbreviated as RR below) is represented as a linear combination. Not only Huang's new instruction 'Compute-Condition-and-Jump' [3] but also a full program can be expressed (all present in DLX [8] mnemonics throughout this paper):

$$s_n \circ \mathbf{bcnd} = s_{n.1} \circ \mathbf{sgt} \oplus s_{n.2} \circ \mathbf{bnez}, \qquad (8)$$

where **sgt** means set-greater-than and **bnez** means branch-on-nonequal. Moreover one inverse (e.g. **sub**) per instruction (e.g. **add**), there exists at least one corresponding RR for self-verification (e.g. **sub** = ⊖**add**).

Input: A linear operator matrix $W_0 = M$
Output: A closure matrix $W_n = M^+$
Method:
1. First transfer to W_k all nonzero W_{k-1} (i, i).
2. Put [nonzero W_{k-1} (i, j), i != j] * [the above W_{k-1} (j, j)] in all the position W_k(i, j).
3. Put W_k(i, j) + $\sum_m$[nonzero W_{k-1}(i, m), i != m] * [nonzero W_{k-1} (m, j), m != j] in all the position W_k(i, j).
4. Repeat step 1, 2, 3 until $W_k = W_{k+1}$.

Figure 1: A modified Warshell's algorithm.

2.3 Instruction combination and transformation

Let's apply the model and show some toy examples. Firstly, linear combination and transformation are two basic operations in linear space. In linear space, we know a vector can be equally expressed as a linear combination of other vectors. All the same, any assembly-code program is a larger linear combination of all instructions. Meanwhile, the $\oplus$ operations represent the sequence of instruction execution and the $\circ$ symbols means label connection for instruction branching. For example, an absolute-value function **abs** is combinations of

$$\begin{aligned} \mathbf{abs} = s_1 \circ \mathbf{lw}.r2, -4(r15) \quad \oplus \quad & s_2 \circ \mathbf{sle}.r1, r2, r0 \\ \oplus s_3 \circ s_8 \circ \mathbf{bnez}.r1, s_8 \quad \oplus \quad & s_4 \circ \mathbf{nop} \\ \oplus s_5 \circ \mathbf{st}. -8(r15), r2 \quad \oplus \quad & s_6 \circ s_{10} \circ \mathbf{j}.s_{10} \\ \oplus s_7 \circ \mathbf{nop} \quad \oplus \quad & s_8 \circ \mathbf{sub}.r0, r2, r3 \\ \oplus s_9 \circ \mathbf{sw}. -8(r15), r3 \quad \oplus \quad & s_{10} \circ \mathbf{nop}, \end{aligned} \qquad (9)$$

where s_3 and s_6 are two branching instructions jumping to s_8 and s_{10}, respectively. Secondly, the Warshell's algorithm's [9] transitive closure is suitable to represent linear transformation via a little modification (refer to Fig. 1). Through an instruction-set transformation matrix, any program can be transformed into a new efficient program achieving the partition goal. Similarly, performance evaluation is done in this manner.

2.4 Efficient evaluation

The evaluation method exploits linear transformation property of linear space. This property is usually represented from a matrix. From technical point of view, efficient evaluation is concerned with sparse matrix and low complexity of matrix multiplication [10]. In fact, the evaluation needs continuous transformation until unchanged. This transitive closure concept is formulated as

$$\mathbf{V} \circ \mathbf{M}^{+} = \mathbf{V}^{+}, \tag{10}$$

where V is in IsV, M is matrix of transformation and superscript + is transitive closure operation.

For example, given a toy instruction set = {+, -, <<, *}, the associated frequencies are {6, 5, 1, 3}. If multiplier can be substituted for an adder and a shifter, and adder can be replaced from subtracter, then the transformation result will be

$$\begin{bmatrix} 6 \\ 5 \\ 1 \\ 3 \end{bmatrix}^{t} \circ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}^{+}_{4\times 4} = \begin{bmatrix} 6 \\ 5 \\ 1 \\ 3 \end{bmatrix}^{t} \circ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}_{4\times 4} = \begin{bmatrix} 0 \\ 14 \\ 4 \\ 0 \end{bmatrix}^{t}, \tag{11}$$

where t means transposition. If the average power consumption of instruction set is {1.20μW, 1.45μW, 0.23μW, 16.19μW}, then the power saving is (6*1.20 - 9*1.45 - 3*0.23 + 3*16.19) = 106.83μW.

3 Object-oriented Mechanism

We propose an easy constructed object-oriented simulator mechanism based on previous model in C++ programming language. Object-oriented conceptual encapsulation, inheritance, and polymorphism satisfy hardware/software codesign concept. First, object-oriented's **class** type constructs instructions into objects quickly: The inheritance and polymorphism exist among many instructions. Especially for encapsulation, every instruction contains individual data information parameters, hardware characteristics etc. all encapsulated as well.

3.1 Executions and statistics

The instruction object declared in class type is prototyped as shown in Fig. 2. The object is the basic element to construct the simulator. In Fig. 2, the constructor *OP_ADD(float, float, float)* initializes the parameter information inputted from outside for evaluation purpose. These hardware parameters come from either CAD tools [11] or experienced formula [12]. Then by *replace_flag*, *add*-procedure makes a substitution decision. These declarations are all in **public** area for the purpose of execution control.

Actually, we execute the simulation easily by linking all instruction objects sequentially. Then the *switch-case* statement in Fig. 2 represents relationship rules among instructions.

```
class OP_ADD {
  int replace_flag;
  float power, delay, area;
  static int count;
  int add( int a, int b) { count++;
      switch (replace_flag) {
        case 1: return Sub.sub(a,-b);
        case 2: return . . .
        default: return a + b; }; }
   OP_ADD( float p, float d, float a) {
      power = p; delay = d; area = a };
} Add;
```

Figure 2: Instruction object.

Before a program execution, we must declare a link list data structure *p[] for it. Every element in the list represent an instruction simulation action. Any assembly program can be translated into this linking list easily, for example, p[0] = &Add; p[1] = &Mul and so on. Finally, through C++'s type conversion, we convert the **void**-type pointer into the corresponding **class** type such as the last three lines in Fig. 3.

Now, we are able to run some short assembly programs. Unfortunately, large benchmarks like SPECint9x are diffcult to deal with except omit these system-function calls. In the contrast, we can execute these system-function calls directly in C++. That is just restoring the corresponding assembly codes into original C++ codes to get the execution result efficiently.

3.2 Optimizations

Optimization is goal of partition in codesign. And transformation is the proposed method for optimization. There are three general ways to decide how to partition [4]. They are integrated in our simple methodology and with regard to our statistics, linear transformation, and RRs strategies respectively:

- Performance experiments on prototype system [13].
- The analysis of system cost factors and the optimization of these through mathematical techniques [14].
- The analysis of system cost factors and the optimization of these through guided user selection [15].

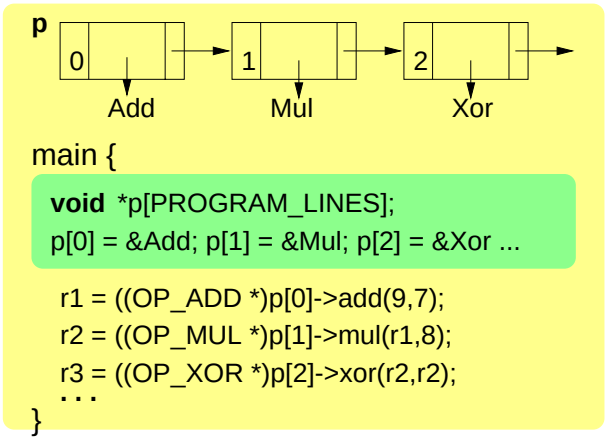

Figure 3: Instruction execution and statistics.

From implementation viewpoint, optimization is achieved by efficient transformation procedure and is concerned with sparse matrix and low complexity of matrix multiplication. The complexity is **O**(product of row number and sum of two sparse matrix's element numbers) [10] which is far more simple than the general **O**(cube of row number). The abstract data type of sparse matrix such as matrix's create, transpose, multiply are described in [10] circumstantially.

From evaluation viewpoint, we guide user to build up their RRs by *class* as shown in Fig. 2. Then *replace_flag* will enable instruction transformation to optimize the hardware/software partition.

4 Applications

Modern processors focus on specific applications. Here we mention two examples about multimedia and semaphores for instruction-level parallelism and multiprocessing synchronization applications respectively.

4.1 Instruction-level parallelism

So far not a real application discussed, in fact application-specific multimedia instructions have a novel demonstration. As in [16], there are seven new instructions and seven multimedia characters are induced. According to these characters, a multimedia program can use the VLIW (Very Long Instruction Word) concept to extend its parallel execution. For example, a typical graphic algorithm Bresenham line-drawing program determines pixel positions using only integer arithmetic and suits to execute in subword packed data types (refer to Fig. 4).

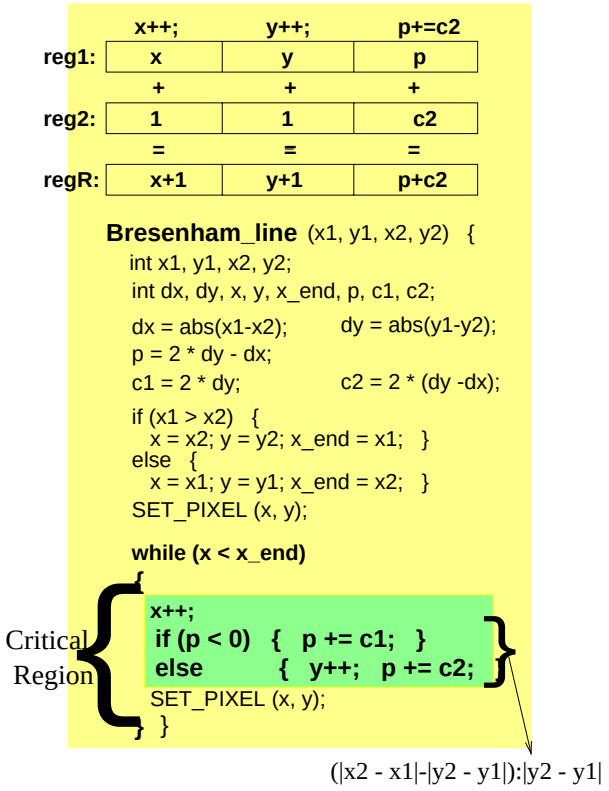

Figure 4: Line-drawing algorithm.

The seven multimedia characters such as small native data types, large data set sizes, large amounts of inherent data parallelism, highly predictable branches etc. [16] can be replacement conditions of RRs. These new rules benefit the critical section in Bres_line(x1, y1, x2, y2) program about 0.1% speedup.

The principle is the **if-else** branch blocks can be predicted exactly as (|x2-x1|-|y2-y1|) : |y2-y1| and the adjacent additions in the block can be combined to single subword execution to save cycle time as shown in (12). The optimization can be probability-oriented even the $x{+}{+}$ instruction may be pulled into the branch block and forced to combine with another instruction

(refer to Fig. 4).

$$s_n \circ sub\_\mathbf{add} = s_{n.1} \circ \mathbf{add} \oplus s_{n.2} \circ \mathbf{add} \quad (12)$$

$$s_n \circ \mathbf{lhi} \oplus s_{n+1} \circ \mathbf{addui} = s_{n.1} \circ new\_\mathbf{lhi}, \quad (13)$$

where **lhi** means load-high-immediate and *sub*_**add** means subword-addition.

By the way, it takes 1.25% of the total cycles on average for a special instruction 'Load-High-Bits' in multimedia programs. It used to transfer a full word data like 'label' and appears with 'Load-Low-Bits' instruction just because the movement cannot be done once. Technically speaking, by reserving the leftmost of instruction and giving one bit to the addressing space we can transfer a full word at a time as shown in (13).

```
General P: wait(S)          General V: sig(S)
    S--;                        S++;
    if S<0 { add to List;       if S<=0 { del from List;
             block;  }                    wakeup(P);  }
Binary P:                   Binary V:
    wait(s3); wait(s1);         wait(s1);
    C--;                        C++;
    if C<0 { sig(s1); sig(s2) } if C<=0 sig(s2);
    else     sig(s1);           sig(s1);
    sig(s3);

main () {
   int cpid;    class OP_ADD shared;
   Semid = SemInit(SEMKEY);  cpid = fork();
   Shmid = ShmInit(SHMKEY, sizeof(shared));
   switch (cpid) {
      case 0:  for ( ; ; ) {  // child process
         P(Semid); cout << shared++; V(Semid);  };
      case -1:  cout << "fork() error!\n";
      default: for ( ; ; ) {  // parent process
         P(Semid); cout << shared++; V(Semid);  };
}
```

Figure 5: Two-process synchronization.

As above, we explore the instruction space efficiently and get he performance increase totally up to 1.35% with RR after Chen's PO (Peephole Optimization) techniques [6]. This application is different from [2] [3] in not extracting characters from narrow benchmarks but widely gathering new multimedia information to codesign.

4.2 Multiprocessing synchronization

There exist a kind of parallel execution mechanism called cooperating process [17]. It can be represented as $a \circ \mathbf{x} \oplus a \circ \mathbf{y}$: At the time stamp a, two instructions are executing. If they concurrently access to the shared data, they may result in data inconsistency. Thus, semaphore mechanism appears to solve this problem involved in multiprocessing environment.

Semaphores are implemented at least in two forms, general and binary [17], but all satisfied P (wait)/V (signal) pair protocol. The general form is just minus/plus operations yet suffers from busy waiting. The binary semaphores are designed to solve this disadvantage but cost more variables and calculations. How to tradeoff them is an issue. The issue contains 'How about making semaphores as hardware not software?' (e.g. SPARC's **ldstub**[18]), 'The time proportion between semaphore and critical section?' and so on.

Our mechanism can easily model a simple two-process synchronization as shown in Fig. 5. The child-process identifier equals zero and the parent equals positive number. P and V can be described in **class** like Fig. 2 and be substituted for many user-defined RRs.

5 Experimential Result

5.1 Example design environment

We construct an extended environment as shown in Fig. 6 for industrial codesign. This environment integrates four tools (compiler, simulator, synthesizer, expert system) to complete instruction transformation (1.ISA in Fig. 6), hardware-parameter evaluation (2.PDA in Fig. 6) and RRs design (guided user selection).

In fact, the previous model and mechanism are enough to make prototyping. However, we involve some mature tools like the expert system to assist users describing their RR rules. Then expert system outputs IDO (Instruction Description Objects) file for optimization.

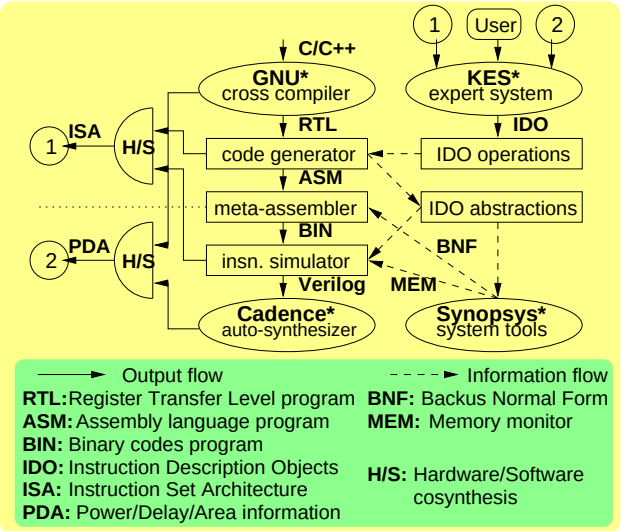

Figure 6: Example experiment of instruction object model.

5.2 Multimedia benchmarks

The result shows that Newton and Ackermann is great optimized by PO (Peephole Optimization) option. As for RR option, Newton and Bresenham_line benefit much. The main RR types we exploit are subword execution: e.g. Eqn. (12) and improved 'Load-High-Bits': e.g. Eqn. (13).

Table 1 and Fig. 7 indicate the cycle times of executed instructions and the speedups. The speedup is calculated based on the execution cycles of the cosynthesis software (benchmarks) and hardware architecture (ASIP) on CAD tool, SYNOPSYS.

Table 1: Cycle times of executed instructions.

Program	*Original*	*with PO*	*PO+RR*
Huffman [19]	314794	302410	296890
Newton [20]	56154	20127	17256
Ackermann	68338	32934	32357
Bresenham_line [21]	82179	79812	78317
Bresenham_circle	361318	356866	352532

PO: Peephole Optimization; RR: Replacement Rules

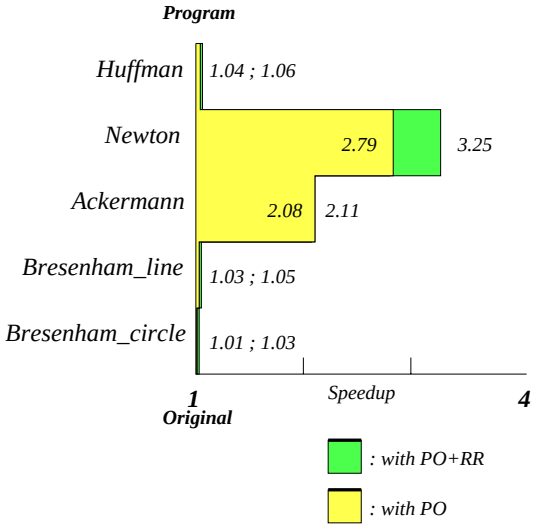

Figure 7: Speedup diagram for Table 1.

6 Conclusions

We propose a new model to integrate ASIP codesign. It is an induction endpoint for past and a deduction start to ASIP development whatever intra- or inter-operation parallelism. Then we modify Warshell algorithm to deal with linear transformation. Based on the fast algorithm, we can either explore design space or automate design efficiently.

Furthermore, we plan a simple but efficient object-oriented simulation mechanism to implement the proposed model. Its abstract data type models the codesign issues easily and quickly, either for instruction-level parallelism or multiprocessing synchronization application. Finally, this model can extend to a larger CAD cosynthesis environment. According to the same methodology, the flow going from software downto the embedded hardware iterates and anneals to an optimal product.

References

[1] J. Harrison, Editorial for Special section on hardware/software codesign for embedded systems, *IEE Proc.-Comput. Digit. Tech.*, Vol.145, No.3, 1998, pp. 153.

[2] J. Sato et al., PEAS-I: A Hardware/Software codesign system for ASIP development, *IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences*, Vol.E77-A, No.3, 1994, pp. 483-491.

[3] I.-J. Huang et al., Synthesis of application specific instruction sets, *IEEE Trans. Computer-Aided Design*, Vol.14, No.6, 1995, pp. 663-675.

[4] D. Morris et al., *Object Oriented Computer Systems Engineering*, Springer-Verlag, 1996.

[5] P. Darche et al., *ActNet: the actor model applied to mobile robitic environments*, Object-Based Parallel and Distributed Computation, selected papers of OBPDC'95, LNCS N. DG 1107, Springer-Verlag, 1996.

[6] T.S. Chen and F. Lai et al., Peephole optimizer in retargetable compilers, *IEICE Trans. Information and Systems*, Vol.E79-D, No.9, 1996, pp. 1248-1256.

[7] S.H. Friedberg et al., *Linear Algebra*, Prentice-Hall, 1996.

[8] D.A. Patterson and J.L. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, 1996.

[9] J.L. Mott et al., *Discrete Mathematics for Computer Scientists and Mathematicians*, Prentice-Hall, Inc., 1986.

[10] E. Horowitz et al., *Fundamentals of Data Structure in C*, W.H. Freeman and Company, 1993.

[11] S.Y. Yuan et al., *Static power analysis for power-driven synthesis*, IEE Proc.-Comput. Digit. Tech., Vol.145, No.2, 1998, pp. 89-95.

[12] A.P. Chandrakasan et al., *Optimizing power using transformations*, IEEE Trans. CAD, Vol.14, No.1, 1995, pp. 12-31.

[13] M.B. Strivastava et al., *Using VHDL for high-level, Mixed-Mode System Simulation*, IEEE Design and Test of Computers, Vol.9, No.3, 1992, pp. 31-40.

[14] S. Kumar et al., *A Framework for Hardware/Software Codesign*, IEEE Computer, Vol.26, No.12, 1993, pp. 39-45.

[15] J.G. D'Ambrosio et al., *Configuration Level Hardware-Software Partitioning for Real Time Embedded Systems*, Proc. 3rd International Workshop on Hardware/Software Codesign, IEEE Press, Los Angeles, USA, 1994.

[16] T.M. Conte et al., *Challenges to Combining General-Purpose and Multimedia Processors*, IEEE Computer, Vol.30, No.12, 1997, xpp. 33-37.

[17] A. Silberschatz and P.B. Galvin, *Operating System Concepts*, Addison-Weslay, 1994.

[18] B. Catanzaro, *Multiprocessor System Architectures*, SunSoft Press, A Prentice-Hall Title, 1994.

[19] R. Sedgewick, *Algorithms in C*, Addison-Wesley, 1990.

[20] S.D. Conte and C. Boor, *Elementary Numerical Analysis: An Algorithmic Approach*, McGraw-Hill, 1980.

[21] D. Hearn and M.P. Baker, *Computer Graphics*, Prentice-Hall, 1986.