PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

A Cooperative Logic Design Expert System on a Multiprocessor

Yoriko Minoda, Shuho Sawada, Yuka Takizawa,
Fumihiro Maruyama, and Nobuaki Kawato

FUJITSU LIMITED
1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan
pro114@flab.fujitsu.co.jp

Abstract

CAD systems that can quickly produce quality designs are needed for the expanding VLSI market. This paper presents a cooperative design mechanism in a cooperative logic design expert system on a multiprocessor, co-LODEX. co-LODEX accepts constraints on area and speed, and outputs a CMOS standard cell netlist that satisfies the constraints. The user can even get an optimal circuit for area or speed by iteratively strengthening the corresponding constraint. Short turnaround is expected through the combination of parallel processing by several processors and their cooperation.

The cooperative design mechanism is based on an evaluation-redesign mechanism using assumption-based reasoning within a single processor. Design alternatives are considered as assumptions and constraint violations as contradictions. Redesign is implemented as a contradiction resolution. The evaluate-redesign cycle repeats itself until the design satisfies the specified constraints. Global evaluation-redesign takes place by processors exchanging design results for subcircuits in terms of gate counts and delays (in case of success) or justifications for constraint violations (in case of failure).

Experimental results show that (1) co-LODEX can efficiently carry out global optimization. For example, a circuit with the minimum number of gates has been obtained while satisfying constraint on speed. (2) Linear speedup has also been observed.

1 Introduction

CAD systems that can produce quality circuits quickly are needed for the expanding VLSI market. One of the most pressing problems is the lack of a means to iterate the cycle of evaluation and redesign until the design satisfies all constraints. Without it, it would be impossible to design a quality circuit with the desired characteristics (area and speed) by looking at the design from a global point of view. There is also demand for CAD systems that can do global optimization for the whole circuit. With such systems, designers can get a circuit with the gate count minimized and the delays kept shorter than the given constraints or vice versa.

Turnaround time seems to be another key issue. Short turnaround allows designers to rapidly implement a variety of architectural choices and to choose the solution best suited for their specific situation by comparing area and speed characteristics. Designers can thus explore their options in a way that has not been practical before.

Since design decisions may be retracted after later evaluation, they can be thought of as assumptions. Assumption-based reasoning uses both facts and assumptions that can be retracted [de Kleer 1986]. Justification, originally introduced for truth maintenance [Doyle 1979], is the key concept to manipulating information containing assumptions. In de Kleer's Assumption-based Truth Maintenance System (ATMS), all assumptions are enumerated in advance and all combinations are examined. In design, however, we are not interested in all combinations. This is because a decision's significance depends on decisions made earlier. We can prune a considerable number of combinations.

A global optimization technique using as linear programming (LP) was proposed [Kageyama 1990]; however, we can not get the exact optimal circuit, because the solution does not always give 0's or 1's for variables that must take 0 or 1.

We proposed an evaluation-redesign mechanism using assumption-based reasoning [Maruyama 1988]. In our evaluation-redesign mechanism, design alternatives are considered as assumptions and constraint violations as contradictions. Redesign is implemented as contradiction resolution. Justifications for violations, called nogood justifications (NJs), play a central role in the mechanism. NJs enable us to drastically prune the search space for constraint satisfaction or optimization problems [Maruyama 1991].

In this paper, we present a cooperative logic design ex-

pert system on a multiprocessor, co-LODEX. co-LODEX divides the whole circuit to be designed into subcircuits in advance and designs each subcircuit on each processor to exploit parallel processing. Global evaluation-redesign takes place by processors exchanging design results (in case of success) or NJs (in case of failure). In our cooperative design mechanism, NJs received from other agents help narrow down the search space for an agent in the sense that NJs made out of the received ones enable the agent to prune the search space. That is the reason why we claim co-LODEX as "cooperative". Short turnaround is expected through the combination of parallel processing by several processors and their cooperation. co-LODEX also has the advantage of exact global optimization.

The next section gives an overview of co-LODEX. Section 3 describes its cooperative design mechanism. We give some experimental results in Section 4 and concluding remarks in Section 5.

```
UHDL;
interface_view: interface_example01;
  inputs: .xi(12), .yi(12), .dxi(12), .ui(12), .ai(12);
  outputs: .xo(12), .yo(12);

behavior_view: behavior_example01;
  define: const5 = 5, const3 = 3;
  terminal: u1(12), u2(12), u3(12), u4(12), u5(12), u6(12), y1(12), FF;
  operator: 2stage_pipelined_multiplier(x, y, z) = ( len = 2 ), z <- x * y; end_op;
  function: main: clk;
  while (FF) do
   2a: '2stage_pipelined_multiplier'(u, dx, u1);
   3a: '2stage_pipelined_multiplier'(x, const5, u2);
   4a: '2stage_pipelined_multiplier'(const3, y, u3);
   5a: '2stage_pipelined_multiplier'(u2, u1, u4),
       x <- x + dx;
   6a: '2stage_pipelined_multiplier'(u, dx, y1),
       FF <- x < a;
   7a: '2stage_pipelined_multiplier'(u3, dx, u5),
       u6 <- u - u4;
   8a: y <- y1 + y;
   9a: u <- u6 - u5, xo := x, yo := y;
  enddo;
  1a: stop(x<a), x <- xi, y <- yi, dx <- dxi, u <- ui, a <- ai;
endUHDL.
```

Figure 1. Example of behavioral specification

2 co-LODEX Overview

2.1 Inputs and Outputs

The user specifies a behavioral specification, a block diagram of the datapath, and constraints on area and speed. co-LODEX outputs a CMOS standard cell netlist that satisfies the constraints. The resulting netlist can be input to an automatic place-and-route system for CMOS standard cells.

The specification language for behavior used in co-LODEX is UHDL [Fujisawa 1989], an extension of DDL [Duley and Dietmeyer 1969]. Figure 1 shows the specification for a circuit that solves a second-order differential equation (DiffEQ). The program might be used to describe a subsystem of a controller or have a digital signal processing application. [Brewer 1987]

A block diagram of the datapath is shown in Figure 2. The boxes signify functional blocks. COMP, MULTI, ADD_SUB, MUX, REG, FF, and the others represent a comparator, a multiplier, an ALU(add/subtract), a multiplexer, a register, a flip-flop and input/output buffers.

Constraints on area are expressed as inequalities in the gate count, for example, "(Total gate count) ≤ 2000." The user can specify as an area constraint the maximum gate

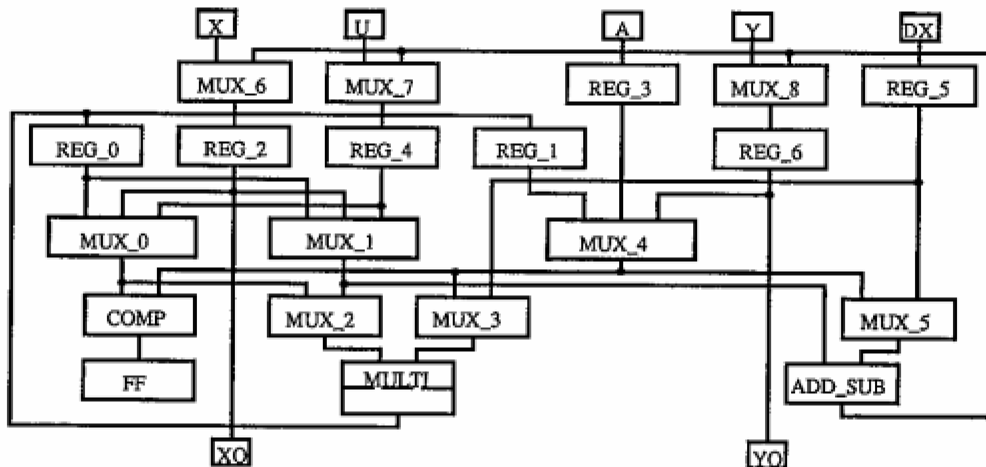

Figure 2. Block diagram

count that could be squeezed into a given LSI device. Constraints on speed are expressed as inequalities in the propagation delay, for example, "(Maximum delay) ≤ 120 ns." The user can specify as a timing constraint the clock cycle the LSI device should operate with.

2.2 Brief Overview

co-LODEX divides the whole circuit to be designed into subcircuits. Each subcircuit is designed by a design agent. Figure 3 shows the five subcircuits for the DiffEQ example and the agents in charge. It should be noted that the control circuit, CTRL, is included. co-LODEX establishes a finite-state machine from the behavioral specification and extracts the specifications for the control circuit in terms of logical expressions. It then divides the whole circuit so that the blocks along critical path candidates are distributed to as few agents as possible. It is likely that agents along a critical path candidate need a considerable amount of mutual communication since agents sharing a constraint must communicate with each other.

Each agent designs given functional blocks hierarchically using the top-down method. It keeps splitting up functional block and subblocks into sub-subblocks until all given blocks are implemented with CMOS standard cells. This is done by referring to the library that includes knowledge about functional block design, knowledge about technology mapping, and standard cells data. Then it counts the number of gates and estimates delays to evaluate the implemented circuit against constraints on area and time.

An agent usually designs its subcircuit independently and in parallel with the other agents. However, since the design results of the other agents are necessary for evaluation against global constraints, agents exchange their results every time they finish design/redesign. An agent redesigns when it detects a constraint violation for which it is responsible, for example, if a path passing through it is too slow. If it designs a standard cell netlist that satisfies all the local constraints, it notifies the resulting gate count and delays. If it cannot, it notifies information about constraint violation.

3 Cooperative Design Mechanism

We propose a cooperative design mechanism on a multiprocessor. It is based on the redesign mechanism within each agent. Moreover, (1) exchanging design results and NJs among agents and (2) combining the NJs received from other agents are necessary. Agents exchange the design results (gate counts and delays) of subcircuits when they succeed in design. They exchange the resulting NJs when they fail to design subcircuits without any stored NJ satisfied.

3.1 Redesign within Each Agent

The area a circuit requires and its delay are the sum of their constituent parts. The delay of a path, for example, can be attributed to that of the components along it. This fact lets us break a global condition into local conditions. A hierarchical structure is useful for this. We explain a redesign

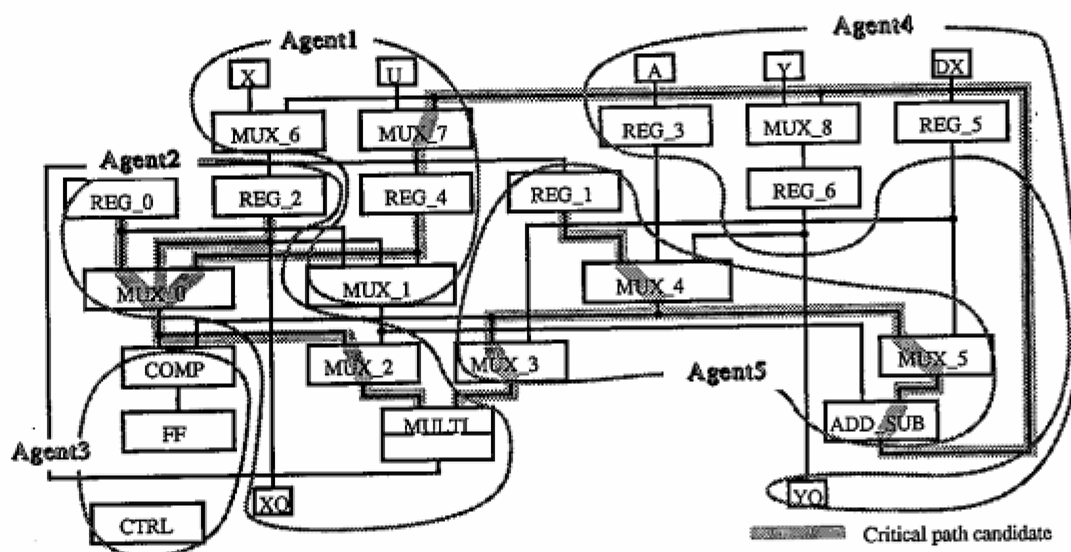

Figure 3. Sub-circuits and agents

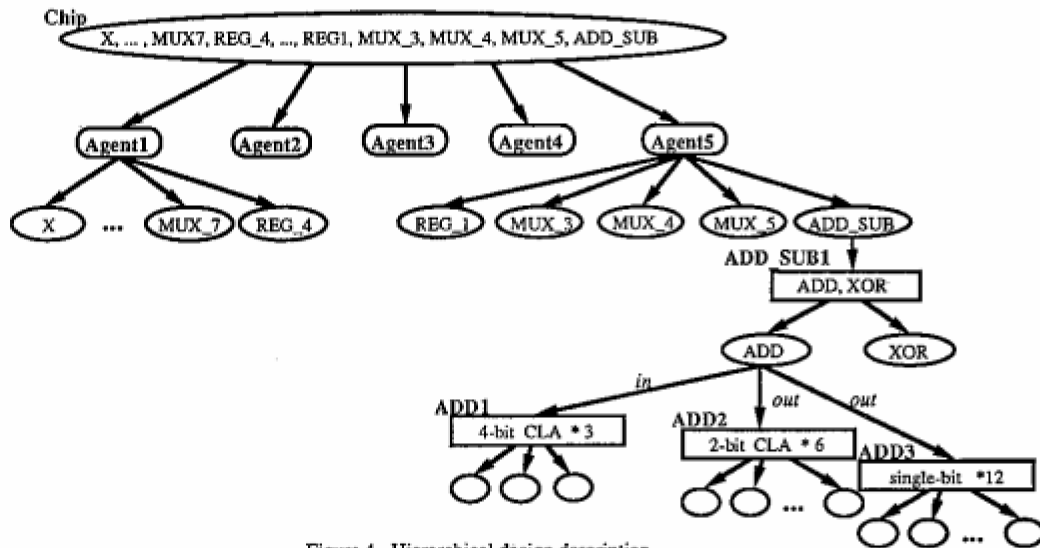

Figure 4. Hierarchical design description

mechanism using assumption-based reasoning, which operates on a hierarchical design description.

Hierarchical Design Description

Design objects are represented in a hierarchy. Figure 4 shows part of the hierarchy corresponding to Figure 3. There are three types of nodes; agent nodes (capsules), component nodes (ovals) and alternative nodes (rectangles). An agent node is responsible for one or more component nodes. A component node associates alternative nodes as possibilities of implementation. There is a special component node called the chip node that corresponds to the whole chip. An alternative node contains information about the connection between subcomponents and has the subcomponent nodes as children. An alternative is called either "*in*" or "*out*" based on whether it is adopted or discarded. Each component node has at most one *in* alternative node. Other alternative nodes are stored in the *out* alternative list to be recalled later if necessary.

Figure 4 shows the following:

- The whole chip (Chip) consists of an input buffer (X), registers (REG_1 and REG_4), multiplexers (MUX_3, MUX_4, MUX_5, and MUX_7), an add/subtract unit (ADD_SUB), and other parts.
- Agent5 is responsible for five components.
- ADD_SUB consists of an adder (ADD) and an exclusive or (XOR).
- ADD, the 12 bit adder, consists of three 4-bit CLA (carry-lookahead adder) cells connected serially. Current *out* alternatives might include a serial connection of six 2-bit CLA adder cells and 12 single-bit adder cells.

Justifications for Constraint Violations (NJs)

An NJ (nogood justification) is a logical expression that must not hold during design. Satisfying an NJ means a constraint violation and invokes the redesign mechanism.

The following default NJ at Chip (in Figure 4) is equivalent to the original constraint on gate count in that any design violating the constraint satisfies it.

$$X(a) + REG_1(a) + REG_4(a) + MUX_3(a) + MUX_4(a) + MUX_5(a) + MUX_6(a) + ADD_SUB(a) + \ldots > CHIP \quad (1)$$

The form, "component ('a')", represents gate count of each component. This says that if the total gate count of the input buffer, the registers, the multiplexers, and so on, exceeds the value of variable CHIP, it means a constraint violation. CHIP is the variable that refers to the currently valid constraint value on gate count, for example 2000. co-LODEX transforms each constraint specified by the designer into default NJs.

A timing constraint in terms of the clock cycle is transformed into a set of default NJs, that is, an inequality representing that the sum of the delays of the components along a path from source to destination exceeds the constraint value. For example, one of the default NJs represents that the path

from REG_1 via MUX_4, MUX_5, ADD_SUB, MUX_7, to REG_4 is longer than the clock cycle. It is as follows:

$$\text{REG_1(p2)} + \text{MUX_4(p1)} + \text{MUX_5(p2)} + \text{ADD_SUB(p2)} + \text{MUX_7(p2)} + \text{REG_4(p1)} > \text{CLOCK} \quad (2)$$

The form, "component ('p' number)", represents a path within each component. CLOCK is the variable that refers to the currently valid constraint value of the clock cycle, for example, 120.

Starting from default NJs, new NJs are added during redesign through NJ expansion and generation as described below. NJs save us doing direct evaluation against constraints. All we have to do is to check to see if any NJ is satisfied.

NJ Expansion

NJ expansion is used to narrow the scope and go down the hierarchy to resolve contradictions, or constraint violations. NJ expansion is formally defined in the following three steps. The NJ to be expanded is the one that is satisfied at the moment.

Step 1: Select a component appearing in the NJ to be expanded. Call it C.

Step 2: Replace C in the NJ with its *in* alternative's subcomponents. If the *in* alternative is at the leaf of the hierarchical structure (at the standard cell level), replace C with its actual gate count or its delay value.

Step 3: Go down the hierarchy to the alternative node and store the NJ obtained in Step 2.

(End)

NJ Generation

If every alternative of a component causes a constraint violation, NJ generation enables us to get a new NJ, the logical product of the NJs corresponding to each alternative. The generated NJ does not refer to that component. It is put at the alternative node one level up. This procedure is justified by resolution [Robinson 1965]. In general, the generated NJ is a logical product of NJs about gate count and NJs about delay.

Evaluation-Redesign Algorithm within Each Agent

The redesign algorithm within each agent uses NJ expansion and generation. Redesign is invoked when an NJ turns out to be true, since satisfying an NJ means a constraint violation.

Step 1: Set ALT to the agent node and proceed to Step 2.

Step 2: Check to see if there is any satisfied NJ at the ancestor alternative nodes (including itself) of ALT. If so, set ALT to the alternative node where the satisfied NJ is put, and proceed to Step 3. Otherwise, go to Step 7.

Step 3: If there is a subcomponent of ALT appearing in the NJ, proceed to Step 4. Otherwise, go to Step 5.

Step 4: Expand the NJ. Set ALT to the current alternative node and return to Step 3.

Step 5: Make ALT *out*. Select another alternative node that is not inhibited by an NJ, make it *in*, set ALT to it, and go to Step 2. If every alternative is inhibited by NJs, proceed to Step 6.

Step 6: Generate an NJ. Set ALT to the current alternative node and go to Step 3. If there is no alternative node one level up, output the generated NJ and exit (Fail!).

Step 7: If there is no component node whose alternative nodes are all *out*, exit. (Succeed!). Otherwise, select an alternative node that is not inhibited by NJs, make it *in*, set ALT to it, and go to Step 2.

(End)

In Step 5, selection is done either by recalling an *out* alternative or by generating a new implementation.

The above algorithm starts when an agent receives information from the other agents. Once the algorithm terminates in success or failure, the agent sends information to the other agents.

3.2 Cooperative Design Algorithm

We propose a cooperative design algorithm by describing the procedure for each agent.

Step 1: Design its subcircuit. Repeat redesign by the evaluation-redesign algorithm. The gate counts and delays of the other subcircuits are assumed to be 0. If any agent fails, the algorithm terminates in failure. Otherwise, proceed to Step 2.

Step 2: Exchange the design results, that is the gate counts and delays of the subcircuits, with the other agents. Proceed to Step 3.

Step 3: Set the gate counts and delays of the other subcircuits to the design results received in Step 2. If no stored NJ is satisfied, go to Step 9. If some of the stored NJs are satisfied and the design results of each agent are the same as in the previous cycle (caught in a loop), go to Step 7. Otherwise, proceed to Step 4.

Step 4: Redesign its subcircuit. If at least one agent succeeds in redesign without any stored NJ satisfied, go to Step 2. Otherwise (all agents fail), proceed to Step 5

Step 5: Exchange the generated NJs with the other agents. Proceed to Step 6.

Step 6: Combine the NJs received in Step 5. Go to Step 1.

Step 7: Set a temporary constraint and proceed to Step 8.

Step 8: Design its subcircuit. Repeat redesign by the evaluation-redesign algorithm until all the constraints including the temporary one are met. The gate counts and delays of the other subcircuits are assumed to be 0. If all the agents fail, the algorithm terminates in failure. Otherwise, go to Step 2.

Step 9: Put together all the subcircuits. The algorithm terminates in success.

(End)

Only default NJs are stored initially. As the algorithm proceeds, new generated NJs and combined NJs are added. In Step 7, select one of the violated constraints with the fewest agents related, and set the current value corresponding to that constraint as a temporary constraint.

Once the above algorithm terminates in success or failure (In Step 1, Step 8, and Step 9), the design run is finished, and the user can retry by changing the constraints. The user can look for a faster circuit by tightening the delay constraint, or can rerun by relaxing the constraints in case of failure. When the constraints are changed, the system updates them and re-evaluates by checking all the stored NJs. As more NJs are accumulated, the efficiency of the algorithm is further improved.

3.3 Combining NJs

When an agent fails in redesign with the evaluation-redesign algorithm described in the above section, it generates an NJ and sends it out to the agents that share it. Each agent "combines" the NJs received from other agents and makes a new NJ out of them. Considering an NJ from an agent as a condition where design is impossible for the agent, the combined NJ can be seen as a condition where design is impossible for agents other than the recipient agent. Agents are required to design without any combined NJ satisfied.

For example, suppose Agent5 received the following two generated NJs (3) and (4) originated from default NJ (1) and (2) from Agent1 and Agent4, respectively ("$\wedge$" signifies logical product):

$$192 + Agent2(a) + Agent3(a) + Agent4(a) + Agent5(a) > CHIP \wedge 19.2 + Agent5(p1) > CLOCK \quad (3)$$

$$Agent1(a) + Agent2(a) + Agent3(a) + 96 + Agent5(a) > CHIP \quad (4)$$

Agent5 combines the above NJs and makes the following new NJs:

$$288 + Agent2(a) + Agent3(a) + Agent5(a) > CHIP \wedge 19.2 + Agent5(p1) > CLOCK \quad (5)$$

$$96 + Agent2(a) + Agent3(a) + Agent5(a) > CHIP \quad (6)$$

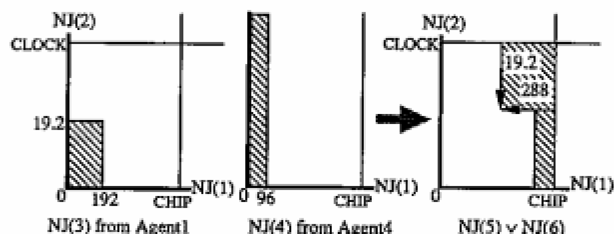

Figure 5. Example of combining NJs

(5) and (6) are added to Agent5.

Figure 5 illustrates the above. The two axes of each graph correspond to default NJs (1) and (2). NJ (3) means that any design by Agent1 is either 192 gates or more, or 19.2 ns or longer along the path for default NJ (2). The left graph shows that Agent1 cannot design inside the hatched part. Similarly, the middle graph shows that Agent4 cannot design inside the hatched part. Combining (3) and (4) gives a condition for the agents other than Agent1 and Agent4 to be unable to design without violating any constraint. If the agents other than Agent1 and Agent4 design inside the hatched part of the right graph, that will cause constraint violation. NJs (5) and (6) represent the hatched part.

4 Experimental Results

We implemented co-LODEX on Multi-PSI [Taki 1988] in KL1 [Ueda 1986] to evaluate the performance of the cooperative design mechanism, and tested as examples to design a specific circuit and usual circuits.

4.1 Optimization

Optimization using co-LODEX proceeds as follows: First, co-LODEX requests the user for area and speed constraints and produces a solution satisfying the constraints. The user then changes area or speed constraint value to the value for the solution just obtained minus 1, and iterates as long as the constraints are satisfied. If constraint satisfaction fails, the previous solution is used as the optimal solution.

Figure 6 shows some of the results for the MAG example. MAG approximates $(a^2 + b^2)^{1/2}$. At first, the area constraint was large enough, and the timing constraint was 130. We obtained the circuit shown at the right. As the area constraint was strengthened, different results were achieved. The smallest circuit, we find is shown at the left. Finally, the above optimization failed in constraint satisfaction with NJ, 1224>CHIP. This means that design is impossible if

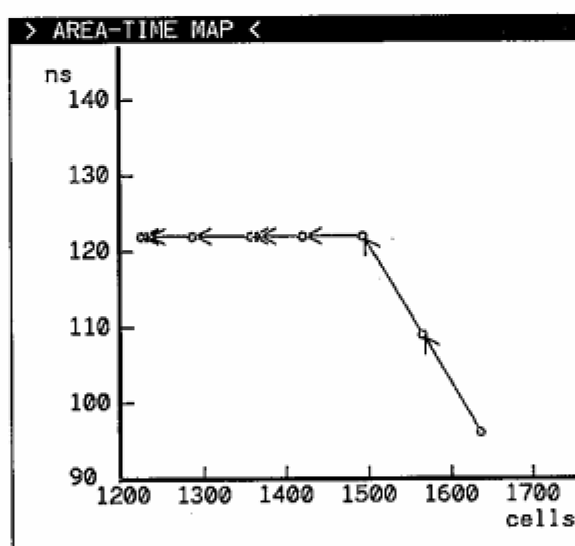

Figure 6. Experimental result for MAG circuit

the specified set of constraints satisfies the NJ. We must thus relax the constraints so that the above NJ is not true any more.

4.2 Speedup

Speedups were examined by increasing the number of agents from 1 to 15. Agents correspond to processors on a one-to-one basis. We had one extra processor for distributing the functional blocks to the other processors and taking statistics, so we used up to 16 processors altogether. We expected that speedups would increase in proportion to the number of agents.

Specific Circuit

The example presented here is to design a multi-argument adder (array adder). The function of this circuit is to calculate the sum of nine integers represented in two's complement format. This circuit is adopted in ALUs and multipliers in other example circuits described below. This circuit consists of 122 one-bit adders. The function of a one-bit adder is to calculate the sum and the carry-out of one-bit integers. Each one-bit adder has many design methods, so the whole circuit has over 50 million design combinations. Table 1 lists the number of design methods with the number of inputs and outputs. Each one-bit adder can be implemented with CMOS standard cells immediately. Thus, we have tested only the cooperative design mechanism of co-LODEX. We used 30 default NJs.

Figure 7 shows a part of this circuit. The boxes represent one-bit adders and the number inside them represent the number of input bits. The arrows represent default NJs. The upper and lower side or upper-left and lower-right side of the arrangement of neighboring blocks has a relationship to the same default NJ. Accordingly, co-LODEX divided the whole circuit and the boundary lines between subcircuits as vertical or slanting (from upper-left to lower-right).

We averaged design costs to agents in this test. We assumed that design costs depend on the total number of design methods for the agent in charge. Taking the number of design methods into account, co-LODEX divided the whole circuit into subcircuits as many as agents. The shaded areas in Figure 7. show two of the subcircuits, where the number of agents is 10. co-LODEX can easily divide this circuit with agents, since it is orderly.

The relation between the number of agents and the speedups is shown in Figure 8, which shows a change in

Table 1. The number of combinations for design method

inputs	sum	carry-out	number of design methods
1	1	0	1
2	1	1	1
3	1	1	1
4	2	1	12
5	2	2	30
6	2	2	15
7	3	2	105
8	3	3	420
9	3	3	84

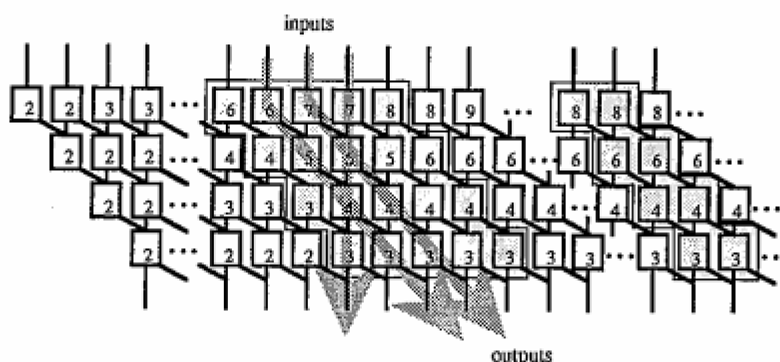

Figure 7. Array adder

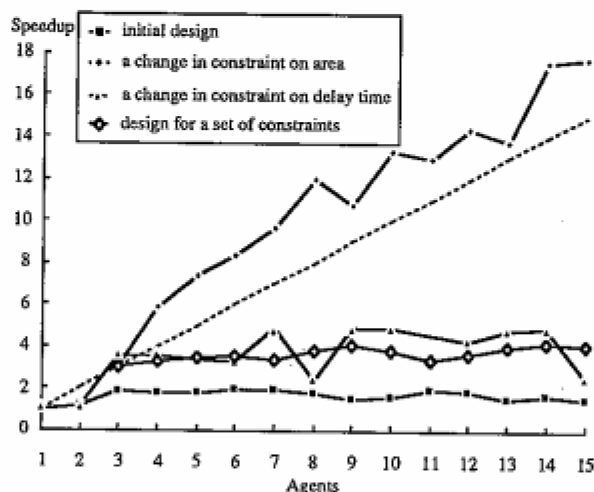

Figure 8. Relation between the number of agents and speedup

design time according to the number of agents. The slanting straight line represents the ideal line. All agents are active in consequence of a change in area constraint, while some agents are active and others are inactive in consequence of a change in delay time constraint. A change in area constraint thus increases speedups and the result surpasses the ideal line. The reason seems to be that our cooperative mechanism reduces the amount of computation by saving useless combinations of alternatives from each agent. Initial design time, time taken until evaluation-redesign occurs, is roughly constant, because the increase in distribution work of the entire specification to agents cancels out the decrease in each agent's design work due to an increase in the number of agents. Figure 8 also shows the speedups for a design, including initial design, when a set of constraints are given.

Usual Circuits

Table 2 lists the results of speedups for design of six usual circuits, including initial design, when a set of constraints are given, together with the optimal number of agents and the time.

A block diagram of the datapath includes various functional blocks. Some functional blocks such as ALU are complex, and others are simpler. We observed that one or two special agents work hard but that the other agents spend time waiting for messages from busy agents. Processing time depends on the busy agents which manage complex functional blocks.

To take advantage of our cooperative design mechanism on a multiprocessor, distribution strategy would need, in addition to focusing on critical path candidates, (1) to look ahead in the library when distributing the functional blocks, and (2) to set up sub-agents if necessary.

5 Conclusion

We presented a cooperative logic design expert system on a multiprocessor, co-LODEX. co-LODEX divides the whole circuit to be designed into subcircuits in advance and designs each subcircuit on each processor to take advantage of parallel processing. Global evaluation-redesign takes place by processors exchanging design results or NJs. A cooperative design algorithm based on assumption-based reasoning makes this possible. Short turnaround is expected through the combination of parallel processing by several processors and their cooperation.

co-LODEX can efficiently carry out global optimization. For example, a circuit with the minimum number of gates has been obtained while satisfying constraint on speed. By

Table 2. Results of experiments

Circuit	Number of Components	Main Components	Speedup	Optimal # of Agents	Time(sec)
Greatest common devisor	11	1 subtracter, 1 comparator	1.1	2	1.7
Differential equation $y'' + 5xy' + 3y = 0$	28	1 multiplier, 1 ALU(add/subtract) 1 comparator	1.3	3	72
MAG(1)	14	1 ALU(add/subtract), 1 comparator 1 two's complementer	1.7	4	3.3
MAG(2)	13	1 ALU(add/subtract/compare) 1 two's complementer	1.2	3	6.4
MAG(3)	16	1 adder, 1 subtracter, 1 comparator 2 two's complementers	5.0	15	3.7
Correlational function $y[i] = \sum_{j=1}^{N-1-j} x[j] * x[i+j]$	22	RAMs, 1 ALU(multiply/add) 1 adder, 1 comparator 1 decrementer, 1 incrementer	2.1	4	113

MAG: Approximation of $(a^2 + b^2)^{1/2}$

increasing the number of agents up to 15, the best linear speedup has been observed.

Our future plans include working on parallel processing of design, evaluation, and redesign within an agent. Distribution strategy is also important for load balancing among processors.

Acknowledgments

This work has been done as part of the Fifth Generation Computer Systems (FGCS) Project of Japan. We would like to thank Dr. Nitta, manager of the Seventh Laboratory of ICOT, for his support.

References

[de Kleer 1986] J. de Kleer: "An Assumption-Based Truth Maintenance System," Artificial Intelligence 28, pp.127-162 (1986).

[Doyle 1979] J. Doyle: "A Truth Maintenance System," Artificial Intelligence 24 (1986).

[Kageyama 1990] N. Kageyama et al.: "Logic Optimization Algorithm by Linear Programming Approach," Proc. of the 27th Design Automation Conference, pp.345-348 (1990).

[Maruyama 1988] F. Maruyama et al.: "co-LODEX: a cooperative expert system for logic design," Proc. of FGCS'88, pp.1299-1306 (1988).

[Maruyama 1991] F. Maruyama et al.: "Solving Combinatorial Constraint Satisfaction and Optimization Problems Using Sufficient Conditions for Constraint Violation," Proc. of the Fourth International Symposium on Artificial Intelligence (1991).

[Fujisawa 1989] H. Fujisawa et al.: "UHDL (Unified Hardware Description Language) and its support tools," Int. J. Computer Aided VLSI Design (1989).

[Duley and Dietmeyer 1969] J. R. Duley and D. L. Dietmeyer: "A digital system design language (DDL)," IEEE Trans. Computers, Vol.C-17, No. 19, pp.850-861 (1968).

[Brewer 1987] F. D. Brewer: "Knowledge Based Control in Micro-Architecture Design," Proc. of the 24th Design Automation Conference, pp.203-209 (1987).

[Robinson 1965] J. A. Robinson: "A Machine Oriented Logic Based on the Resolution Principle," Journal of the ACM, Vol.12, No.1, pp.23-41 (1965).

[Taki 1988] K. Taki: "The Parallel Software Research and Development Tool: Multi-PSI system," Programming of Future Generation Computers (1988).

[Ueda 1986] K. Ueda: "A Parallel Logic Programming Language with the Concept of a Guard," ICOT Technical Report, TR-208 (1986).