Predicting Area-Time Tradeoffs For Pipelined Design

R.Jain, A.C.Parker
Dept. of EE-Systems
Univ. of Southern Calif.
Los Angeles, CA 90089

N.Park
Dept. of EE
Univ. of Calif.
Irvine, CA 92717

Technical Report CRI-87-09

This research was supported by the Semiconductor Research Corporation contract 86-01O75 and by the Army Research Office Grant DAAG29-83-K-0147.

This paper was presented at the 24^{th} Design Automation Conference, ACM SIGDA, IEEE Computer Society, Miami, 1987.

ABSTRACT

In this paper we give a model for predicting the shape of cost-speed tradeoff curves for pipelined designs. The model includes prediction of the number of operators, registers and multiplexers from a behavioral specification. It has been verified with the designs generated by an automated pipeline synthesis program, Sehwa. This model was developed as a part of the ADAM Advanced Design Automation System of the University of Southern California.

1. INTRODUCTION

Synthesizing datapaths automatically is computationally expensive for production designs. Many trial synthesis passes or computations are made to experiment with different sets of modules and varying degrees of concurrency and resource sharing. For example, Sehwa [3], a part of the USC ADAM system, a pipeline datapath synthesis program, is a good example of such software. The input to Sehwa is a dataflow graph, and a set of module types which can be used to implement the operations of the dataflow graph. Sehwa gives as an output the number of each type of operator required and the scheduling of the dataflow graph. Sehwa also takes into consideration conditional branching within the dataflow graph and resynchronization due to resource conflicts and data dependencies. The scheduling is a static scheduling which takes into account all possible combinations of conditional branches which can occur. First, Sehwa produces the fastest and the cheapest designs to fix the design boundary. Sehwa then requests the user for a speed or cost constraint and generates several solutions meeting the constraint. The user then changes the constraints and iterates. Finally, the user can perform exhaustive search in a small part of the design space to tune the design. Once a design is selected, redesigning may occur with a different module set. Fig. 1 shows a cost-delay tradeoff curve produced by Sehwa.

If one could predict approximately where designs would occur in the design space, the search for a satisfactory design could be narrowed considerably. At the University of Southern California, we have developed a technique for predicting an approximate cost/speed tradeoff curve for pipelined datapaths, from a specification of the desired functional behavior.

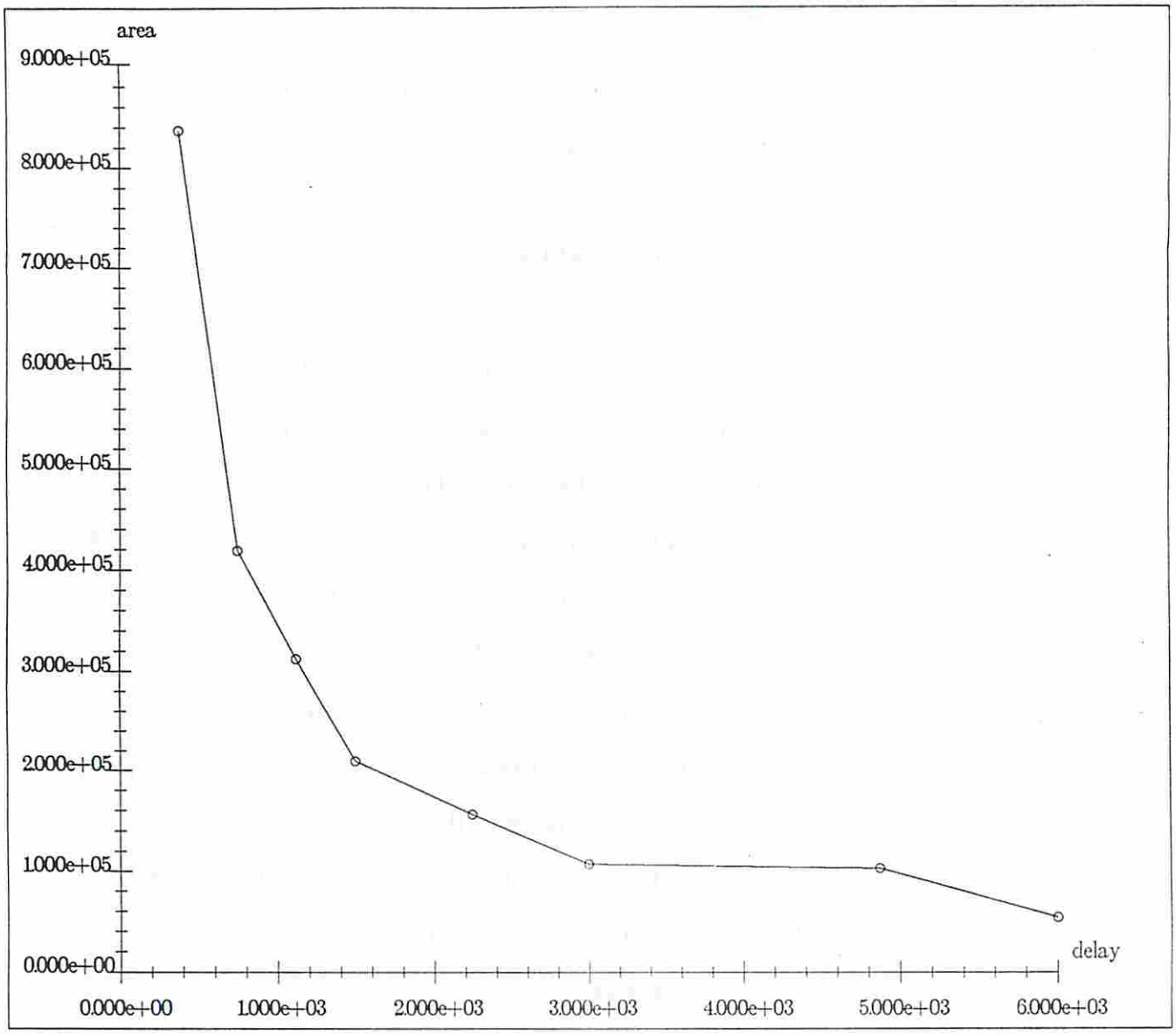


Fig.1 Sample Area-Delay Curve as Produced by Sehwa

The prediction technique presented here is part of the ADAM Advanced Design AutoMation [4] system being constructed at USC, and works in conjunction with Sehwa. However, the technique is general and can be applied to any pipeline synthesis system which produces near-optimal results.

In the next section we give the problem description and the solution approach. Section 3 discusses the lower bound estimates for area and time. Section 4 contains the list of experiments which were conducted using the estimation techniques and Sehwa. The last section concludes with experimental results and observations.

2. PROBLEM DESCRIPTION AND APPROACH

Datapath synthesis programs usually input a dataflow graph which specifies the required behavior of the hardware to be synthesized, and a set of possible module types which will implement the operations. Given this input, we are able to predict an area-time curve in the design space which forms a lower bound, for all possible designs; all the design points lie either on the curve, or above it. The design points lying on this curve represent optimal designs. In this paper *optimality* will be defined with respect to resource utilization. A *resource-optimal* design is one in which all the operators perform useful computations every clock cycle. Of course, in practical pipeline designs, it is often not possible to utilize all the operators in every cycle, yielding suboptimal designs.

Our prediction technique works as follows. We estimate the cost-speed tradeoff curve to have the shape $AT = k$ where k is a constant. This curve is a *lower bound estimate* on the cost-speed curve of the resultant design. An estimate of register and multiplexer costs is added to this curve.

The theoretical results obtained from these lower bound estimates have been verified through designs generated by Sehwa. (Sehwa guarantees near-optimal designs and hence such a comparison is possible.) We are not aware of any other pipeline synthesis package and, as such, further comparison to actual design software is not possible.

3. THEORETICAL FOUNDATION

In Section 3.1, we present techniques for operator cost estimates, and estimation of the $(AT)_{\min}$ curve. Then we describe two techniques for register cost estimation in Section 3.2 and a technique for multiplexer estimation in Section 3.3.

Nomenclature and Assumptions : Before we describe our technique in detail, we present some nomenclature and assumptions. In our discussion we call a logical operation which appears in the dataflow graph an *operation* and the implementation in the resultant design an *operator*. We are given a dataflow graph with m different operation types. For $0 \leq i \leq m-1$, let opt_i be the number of operators of type i used for implementation, opn_i the number of operations of type i in the dataflow graph, $delay_i$ the delay of operator i, and $area_i$ the area of operator type i respectively. Also, let the total area $A = \sum_{i=0}^{m-1} opt_i \times area_i$. *Clock_cycle* is the clock rate. The clocking is restricted in that if an operator starts execution in a particular *clock_cycle*, then it must terminate in the same *clock_cycle* as well. This imposes the condition $clock\_cycle \geq maximum\,(delay_i)$ where maximum is taken over all the operators chosen for implementation. *Latency* is the number of clock cycles between two successive data inputs, and the *initiation delay* $T = latency \times clock\_cycle$.

A particular type of operation, for example add, may be implemented with several types of modules, e.g., carry-look-ahead adder, ripple-carry adder. An assumption in our work is that all the operations of a particular type are implemented with only one type of module. For example, if there were 5 add nodes in the dataflow graph, all the add operations would be performed on ripple-carry type adder(s). Furthermore, for the sake of estimating *lower bounds*, we assume a zero resynchronization rate for the pipeline. This means we assume no data dependency or resource conflicts. It does not affect our results as resynchronization serves only to increase the delay for the same design area and such designs lie above our predicted lower bounds.

We also assume $clock\_cycle = maximum\,(delay_i)$. This assumption is valid for our lower bound estimate studies, and will be explained at the end of Section 3.1. We are able to use the simple estimation technique mentioned above because the theory of pipeline synthesis predicts such curves. We now present the theoretical foundation of our technique.

3.1 Operator and AT Curve Estimation

We shall first compute the effective number of operations of a given type c_opn_i in a dataflow graph. Consider a dataflow graph with one conditional branch and one type of operation *foo*. The *YES* path from the *IF-CONDITION* node has 10 *foo* nodes and the *NO* path has 5 *foo* nodes. Also, the remaining (with no conditional branches) dataflow graph has 3 *foo* nodes. Then, the maximum number of *foo* nodes which can be executed for any set of inputs is 3+*maximum* (5,10)=3+10=13. Thus our c_opn_{foo}=13 for this dataflow graph. We can generalize this counting mechanism over a dataflow graph with many conditional branches and operations.

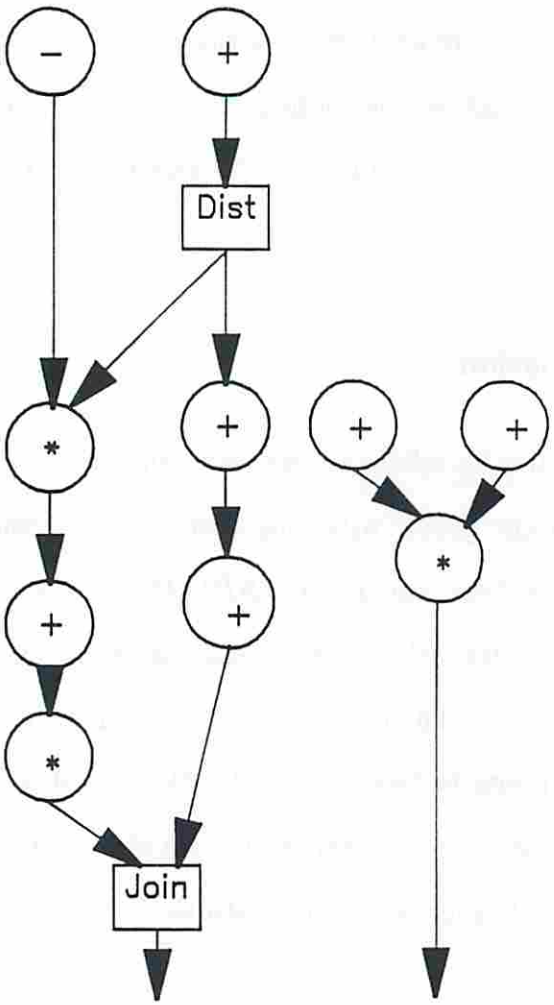


Fig. 2 Dataflow Graph with Conditional Branching

For a dataflow graph with no conditional branching, let c_opn_i, the effective number of operations of a given type, be the same as opn_i. For a dataflow graph with conditional branching, we compute c_opn_i in the following manner. For the graph portion for which there is no conditional branching, c_opn_i will be calculated in the same way as opn_i. For the conditional branches we take the longest path emanating from the *distribute* node and find the number of operations in that path. Then we take the other paths and add to each c_opn_i the number by which the operation count in the shorter path exceeds the count in the previously traversed paths. For example, in Fig. 2 $c\_opn_{subtract}=1$, $c\_opn_{add}=5$ and $c\_opn_{multiply}=3$. We use c_opn_i to define the utilization of each operator. A utilization of 1 is optimal.

Definition 1: The utilization of each operator $0 \leq i < m$ is defined as

$$utilization_i = \frac{c\_opn_i}{latency \times opt_i}$$

Using this definition, we base our estimation technique on the following theorem.

Theorem 1: Given a specific dataflow graph, for resource-optimal pipeline design ($utilization_i=1$), $(AT)_{\min}=constant$, where $(AT)_{\min}$ is the lower bound of all possible ATs for that graph.

Proof: We know that, for any operator, $latency \geq \frac{c\_opn_i}{opt_i}$ [3]. For the particular case where the operator i is utilized every clock cycle, which is the best case,

$$latency \times opt_i = c\_opn_i$$

and therefore,

$$latency \times area_i \times opt_i = area_i \times c\_opn_i$$

Summing this over m operations and multiplying by the *clock_cycle*,

$$clock\_cycle \times latency \sum_{i=0}^{m-1} (area_i \times opt_i) = clock\_cycle \sum_{i=0}^{m-1} (area_i \times c\_opn_i)$$

$$A \times T = clock\_cycle \sum_{i=0}^{m-1} (area_i \times c\_opn_i)$$

For a given dataflow graph and a set of possible module types, the right hand side is a constant, and hence

$$(AT)_{\min} = constant \,. \; \square$$

---

procedure **estimate_lower_bound** (*latency*)

begin

for $0 \leq i < m$ **calculate** $opt_i = \left\lceil \frac{c\_opn_i}{latency} \right\rceil$;

for $0 \leq i < m$ **calculate** $A = A + opt_i \times area_i$;

$clock\_cycle = maximum\,(delay_i)$;

$T = clock\_cycle \times latency$;

print $A \times T$;

end;

Algorithm 1. Procedure to Estimate $(AT)_{lb}$

The above theorem predicts a curve $(AT)_{\min} = constant$ for optimal designs. This means that for every possible latency and for all operators, $utilization_i = 1$. For many dataflow graphs, $utilization_i = 1$, and hence $(AT)_{\min} = constant$, is not possible for every latency and every operator. In this case, although a lower bound curve $(AT)_{lb}$ does exist, it is not equal to a constant due to the ceiling function computing opt_i. Our estimation technique, Algorithm 1 calculates the $(AT)_{lb}$ curve. In case the design is optimal it calculates the $(AT)_{\min}$ curve, otherwise it calculates the lower bound $(AT)_{lb}$ curve. If $d = \sum_{i=0}^{m-1} opn_i$, the above procedure is called d times with latency varying from 1 to d

to get the lower bound curve.

We shall now justify the assumption that $clock\_cycle = maximum\,(delay_i)$, where maximum is taken over all types of operators. We make two observations for our justification. For a given latency we can compute the minimum number of operators of each type required for the implementation of the dataflow graph as

$opt_i = \left\lceil \frac{c\_opn_i}{latency} \right\rceil$. Also, the area of implementation, A, which depends on the number of operators, can be computed as $A = \sum_{i=0}^{m-1} (area_i \times opt_i)$. We observe that making *latency* a constant, also fixes the minimum required number of each type of operator. Thus for a fixed *latency*, the minimum area A of the design becomes a constant.

Consider two pipeline designs of the same dataflow graph with the same latency, and hence same area, but different clock cycles. Then the design with lower clock cycle will have a lower AT (as the latency and area of the two designs is the same). Under the above assumptions, the minimum value *clock_cycle* can take is $maximum\,(delay_i)$. Hence our choice of $clock\_cycle = maximum\,(delay_i)$.

Note that a design with the above computed minimum $(AT)_{\min}$ or $(AT)_{lb}$ may not exist. In fact, the best possible actual design might not be one with a minimum clock cycle. However, there can be no design which will have an AT curve lower than the one computed by Algorithm 1 under the above assumptions.

3.2 Register Estimation

We shall now discuss the effect of register cost and delay on the curve. It is assumed that each operation is diadic and produces exactly one output. We shall later

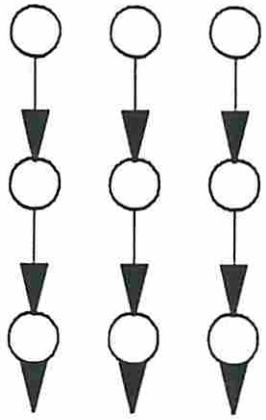
show how this restriction can be relaxed. In our discussion we first estimate the number of internal registers which will be required for implementation. The number of registers (internal and external) can be estimated as *maximum(number of external registers, number of internal registers)*. The external registers are those required at the input or the output of the dataflow graph. We state that a value is consumed if it is not required in any subsequent clock cycles.

Theorem 2: The minimum number of internal registers required for a pipeline implementation of a dataflow graph is

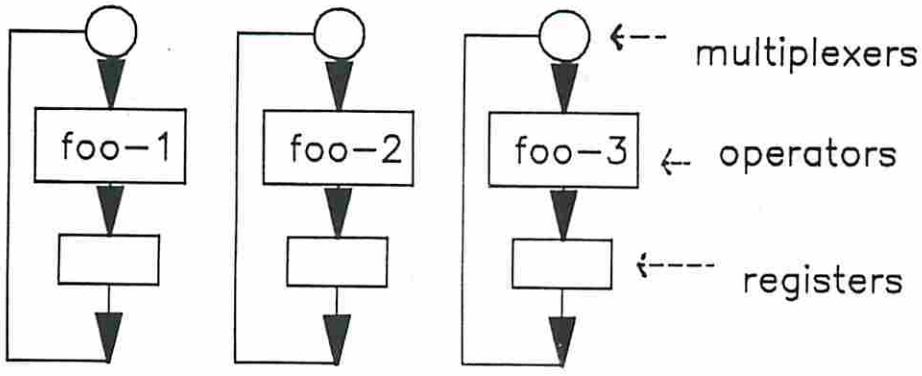
$$\frac{\sum_{i=0}^{m-1} c\_opn_i}{latency}$$

Proof : Any pipeline implementation of a dataflow graph will require registers to store the output of an operator, which will be used in a future clock cycle. It may be used in the immediately next clock cycle, or several clock cycles later. The best case for our analysis of lower bound is the former case, i.e. where a value is produced in one clock cycle and it is consumed in the immediately next one, thus freeing the register to be used again. $\sum_{i=0}^{m-1} c\_opn_i$ gives the total number of operations which will be executed in the dataflow graph for any one set of input data. In the best case, an average $\frac{\sum_{i=0}^{m-1} c\_opn_i}{latency}$ number of operations will be performed every clock cycle, and hence this number of registers will be required. □

If an operation is not diadic and produces more than one output, we simply add each additional output to the summation in the above expression.



(a) Dataflow Graph; all operations are of type "foo"



(b) Implementation using 3 operators; latency = 3

Fig. 3 Register Estimation Example

An example of a dataflow graph with its implementation and scheduling performed by Sehwa is given in Fig. 3. In this case the estimated number of registers and the number required by the implementation is the same.

If one considers the registers required at the input and the output of the dataflow graph, then it is not possible to make a deterministic analysis of the total number of registers required, given only the dataflow graph. This is because the consumption of an input value and the generation of an output value depend on the scheduling of the dataflow graph. A better deterministic minimum estimate of the number of registers required for implementation can be computed. This estimate requires the pipeline schedule of the dataflow graph as well as the latency. Assume that the dataflow graph is pipelined into n stages.

Definition 2: Given a data flow graph and a pipeline schedule, cut_i is defined to be the number of edges cut by the i^{th} stageline.

Definition 3: The number of cuts for each clock cycle j, $0 \leq j \leq latency-1$, in a scheduled graph, $total_j$, is calculated as follows :

for $j=0$ **step** 1 **until** $latency-1$ **do**

  for $i=j$ **step** $latency$ **until** $n-1$ **do**

    $total_j = cut_i + total_j$.

Theorem 3: The minimum number of registers required for a given schedule and latency is given by $maximum\,(total_j)$, where maximum is over $0 \leq j < latency$.

Proof : For every clock cycle $0 \leq j < latency$, $total_j$ gives the number of edges crossing a time step. This gives the number of registers required in that clock cycle. As the registers can be shared between any two clock cycles, the minimum number of registers required, assuming all registers can be shared, will be for the clock cycle with

maximum ($total_i$). □

The above theorem gives an estimate of the number of registers required for a given latency. Thus, to get register estimates over all possible latencies, the above computation is performed with latency varying from 1 to $\sum_{i=0}^{m-1} opn_i$. A new schedule of the dataflow graph has to be supplied each time the latency is varied.

To evaluate the effect of registers on the lower bound curve, the *clock_cycle* will now be the sum of

1. the time required to read a register, D_{ss},
2. the time required to latch the result in the register, D_{sp}, and
3. the *maximum* ($delay_i$).

Thus

$$overall\_clock\_cycle = maximum\,(delay_i) + D_{sp} + D_{ss}$$

Also,

$$A = \sum_{i=0}^{m-1} (area_i \,.\, opt_i) + number\_of\_registers \times area_{register}$$

3.3 Multiplexer Estimation

We now estimate the number of multiplexers required.

Let R be the (actual or estimated) number of registers used in a design. Let inp_i be the number of input terminals of opt_i. Also, $O = \sum_{i=0}^{m-1} (inp_i \times opt_i)$, i.e. the design has O terminals to which the registers are connected. For example, a design having 5 adders and each adder having two input terminals, the total number of terminals $O = 2 \times 5 = 10$. Assume that all the registers will be connected to the terminals through

multiplexers alone (i.e. no bus interconnect). Furthermore, if we restrict the usage of multiplexers to only one type (i.e. d-to-1 type multiplexers)*, then we can calculate a lower bound on the number of multiplexers required for the design, as in the theorem below.

Theorem 4 : To interface R registers to O terminals via d-to-1 multiplexers, at least M multiplexers will be required, where

$$M = \left\lceil \frac{R-O}{d-1} \right\rceil$$

Proof : We shall prove this equation for a fixed number of terminals O, by induction. Assume multiplexers and registers are uniformly distributed.

Basis : For $R=O$, the above equation evaluates to $M=0$; i.e. no multiplexers are required. In this case each register can be connected to a terminal directly and no multiplexing is needed.

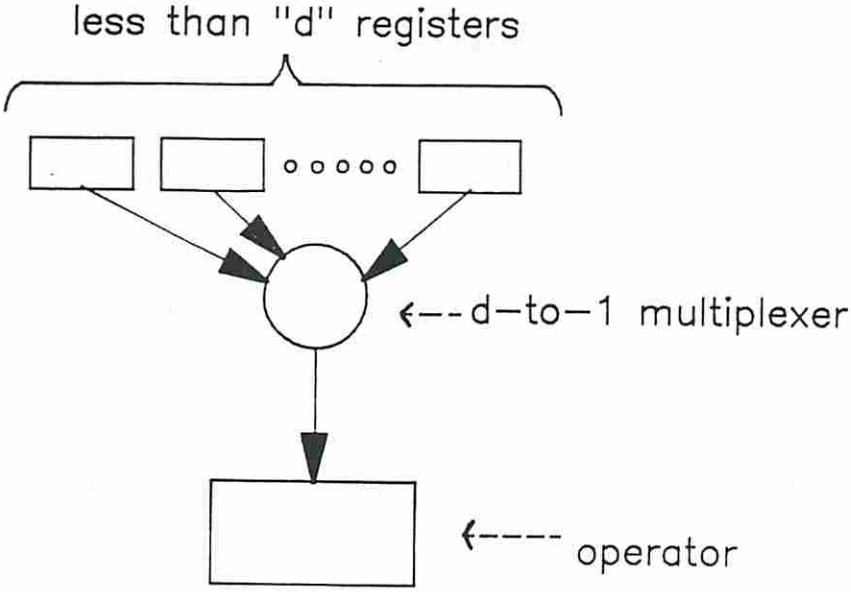
Hypothesis : Let the above equation be correct for $R=n$ and the number of multiplexers computed be $M=x$. Note that $x(d-1)\geq(n-O)$.

Induction : Let $R=n+1$, and the number of multiplexers computed be $M=y$. We have two subcases here.

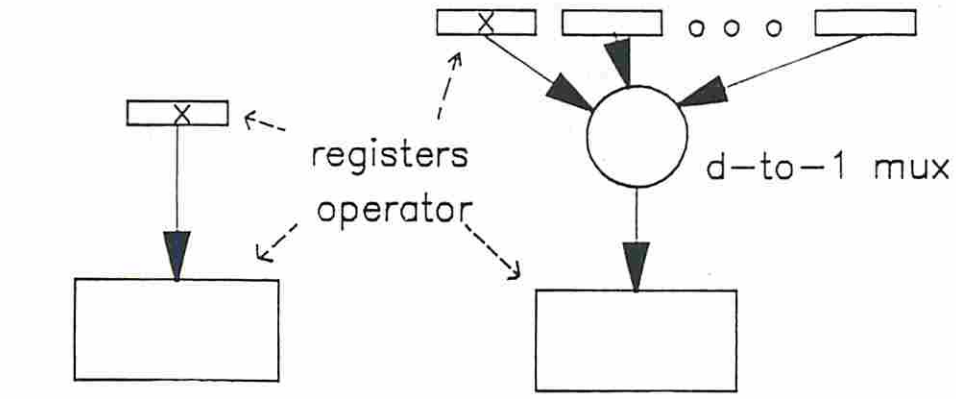
Subcase 1: $x=y$. This is the case where $x(d-1)$ is strictly greater than $(n-O)$. Hence, we can add a register (increase n by 1) without having to increase x. We observe that this case arises when there is at least one multiplexer with at least one of its d inputs unconnected, and the the additional register can be connected to this input (Fig. 4a).

Subcase 2: $x+1=y$. In this case $x(d-1)=(n-O)$, and all the multiplexers have all

* This restriction can be relaxed, as shown later.



(a) $x(d-1) > (n-0)$



(i) before adding (ii) after adding

(b) $x(d-1) = (n-0)$

Fig. 4 Multiplexer Estimation Examples

their d inputs connected. Thus, to connect another register to the design (increasing n by 1), we have to increase the number of multiplexers by 1. Following Fig. 4b we shall observe the effect of connecting this new multiplexer. To connect the new multiplexer, an already connected register will have to be removed from the design, the new multiplexer connected in place of the removed register, and this removed register added to the input of the new multiplexer. We now have $d-1$ additional inputs to which $d-1$ additional registers can be connected. Hence, for every new multiplexer which is added to the design $d-1$ additional registers can be connected without increasing the number of multiplexers. This $d-1$ also gives the divisor in the equation. The new register can now be connected to an input of the new multiplexer. □

The restriction on using d-to-1 multiplexers can be relaxed in that any d-to-1 multiplexer can be replaced at the same level in the tree by a d'-to-1 multiplexer such that the replacement does not increase the number of multiplexers. For example, if in a design we have to connect 6 registers to 2 terminals using $4\text{-}to\text{-}1$ multiplexer, then we can replace these by either two 3-to-1 multiplexers (if they exist) or by one 4-to-1 multiplexer and one 2-to-1 multiplexer.

Thus if D_{md} is the delay through a $d\text{-}to\text{-}1$ multiplexer, then the effect of the multiplexers is calculated as

$$overall\_clock\_cycle = maximum\,(delay_i) + D_{sp} + D_{ss} + D_{md} \left\lceil \log_d M \right\rceil$$

Also,

$$A = \sum_{i=0}^{m-1} (area_i \times opt_i) + number\_of\_registers \times area_{register} + M \times area_{d\text{-}to\text{-}1\,multiplexer}$$

4. EXPERIMENTS

Several experiments using Sehwa [3] were conducted. Of these, the three examples described herein are : (i) an AR lattice filter systolic array element [1] which was converted to a data flow graph and expanded for complex operations (Fig. 5); (ii) a random data flow graph generated using a random number generator, (Fig. 6), and (iii) a data flow graph with conditional branching, [see Fig. 5.1 of Ref. 3]. Three sets of modules with different area-delay parameters were used for each dataflow graph. These area-delay characteristics were obtained from PLEST [2] and are given in Table 1. In this paper we only present the results obtained from one set of modules. The results from the remaining two sets of modules were consistent with the results reported here.

Figs. 7 to 9 show the area-delay tradeoffs for the above three examples and Figs. 10 to 12 show the same results drawn on a log scale. Each graph has four curves corresponding to the following four cases :

1. $(AT)_{\min}$=*constant* .

2. The $(AT)_{lb}$ curve obtained from *procedure* **estimate_lower_bound.**

3. The design curve produced by Sehwa, neglecting the cost and delay of registers.

4. The estimated cost of registers, as computed in Theorem 3, is added to $(AT)_{lb}$.

5. RESULTS AND FUTURE RESEARCH

The curves show that $(AT)_{lb}$ is a good lower-bound approximation for pipelined design. For the AR lattice filter, several optimal design points have been achieved by Sehwa, as can be seen in Fig. 10. This approximation allows us to narrow our search of

the design space when synthesizing pipelines. The nonlinearity in the curve drawn on the log scale reflects the non-optimal use of the operators. In the case of non-optimal designs also, the $(AT)_{lb}$ produced by *procedure* **estimate_lower_bound** is a good approximation. This can be seen in the curves for the random dataflow graph (Figs. 8 and 11) and the graph with conditional branches (Figs. 9 and 12).

From the above graphs certain observations can be made. Consider only one operator type. We know that the minimum value of $opt_i = \left\lceil \frac{c\_opn_i}{latency} \right\rceil$. (For $utilization_i = 1$, the ratio $\frac{c\_opn_i}{latency}$ has to be an integer.) Thus if we increase latency, we decrease opt_i proportionally. However in many general cases we have a pipeline consisting of one or two very expensive (in terms of area and delay) operators with very low usage as against several cheap ones with high usage, resulting in non-optimal designs. The design, in this case, will not be optimal if latency is greater than the number of these operations (as $c\_opn_i < latency$ and $utilization < 1$). If the *latency* is increased beyond c_opn_i, the expensive operator will remain idle for $(latency - c\_opn_i)$ clock_cycles. In the curves for a random dataflow graph (Figs. 8 and 11), one can observe the effects of having 2 expensive operations (multiply) as against 26 cheap operations (add and subtract). Thus selecting module sets such that the non-optimality is minimized is a major problem. An approach to reducing the imbalance in the operators cost/delay may be achieved by the decomposition of the expensive node to smaller, cheaper nodes in the dataflow graph.

The example having conditional branches shows that the theoretical curve is non-linear and for this type of dataflow graph also, the above procedure is a very good approximation of the results produced by Sehwa.

Comparing the estimated $(AT)_{lb}$ curve with the estimated $(AT)_{lb}$ and estimated register curve in Fig. 10 it is seen that the register cost becomes more significant as the area is reduced. Thus even though the operators can be optimized, the dataflow graph is so partitioned that it leads to a non-optimal number of registers. It would be interesting to see if a partitioning of the graph which optimizes the use of both registers and operators could be arrived at.

6. REFERENCES

1. S. Y. Kung, "On Supercomputing with Systolic/Wavefront Array Processors", *Proc. IEEE,* Vol.72, No.7, July 1984.

2. F. J. Kurdahi and A. Parker, "PLEST: A program for Area Estimation of VLSI Integrated Circuits", *Proc. 23rd Design Automation Conference,* ACM SIGDA, IEEE Computer Society, June 1986.

3. N. Park and A. Parker, "SEHWA: A program for Synthesis of Pipelines", *Proc. 23rd Design Automation Conference,* ACM SIGDA, IEEE Computer Society, June 1986.

4. J. Granacki, D. Knapp and A. Parker, "The ADAM Advanced Design Automation System : Overview, Planner and Natural Language Interface", *Proc. 22nd Design Automation Conference,* ACM SIGDA, IEEE Computer Society, June 1985.

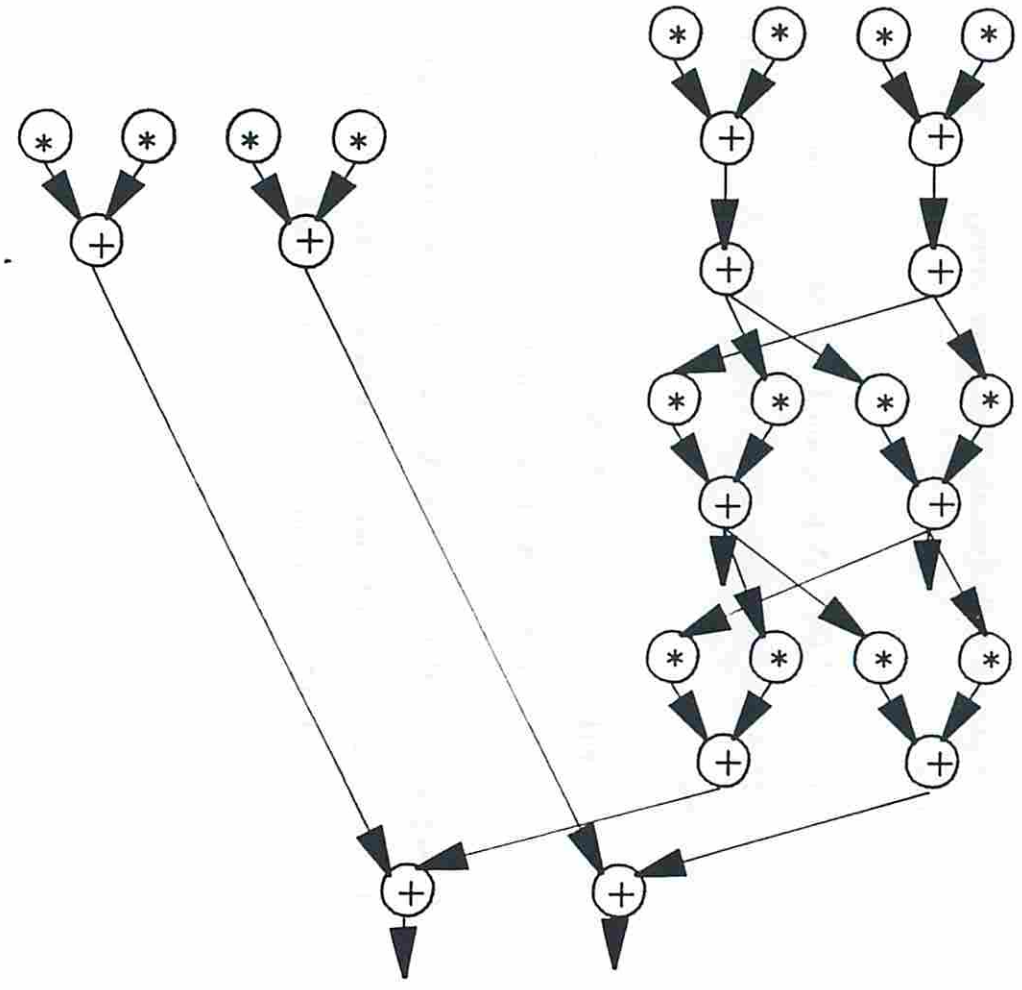


Fig. 5 AR Lattice Filter

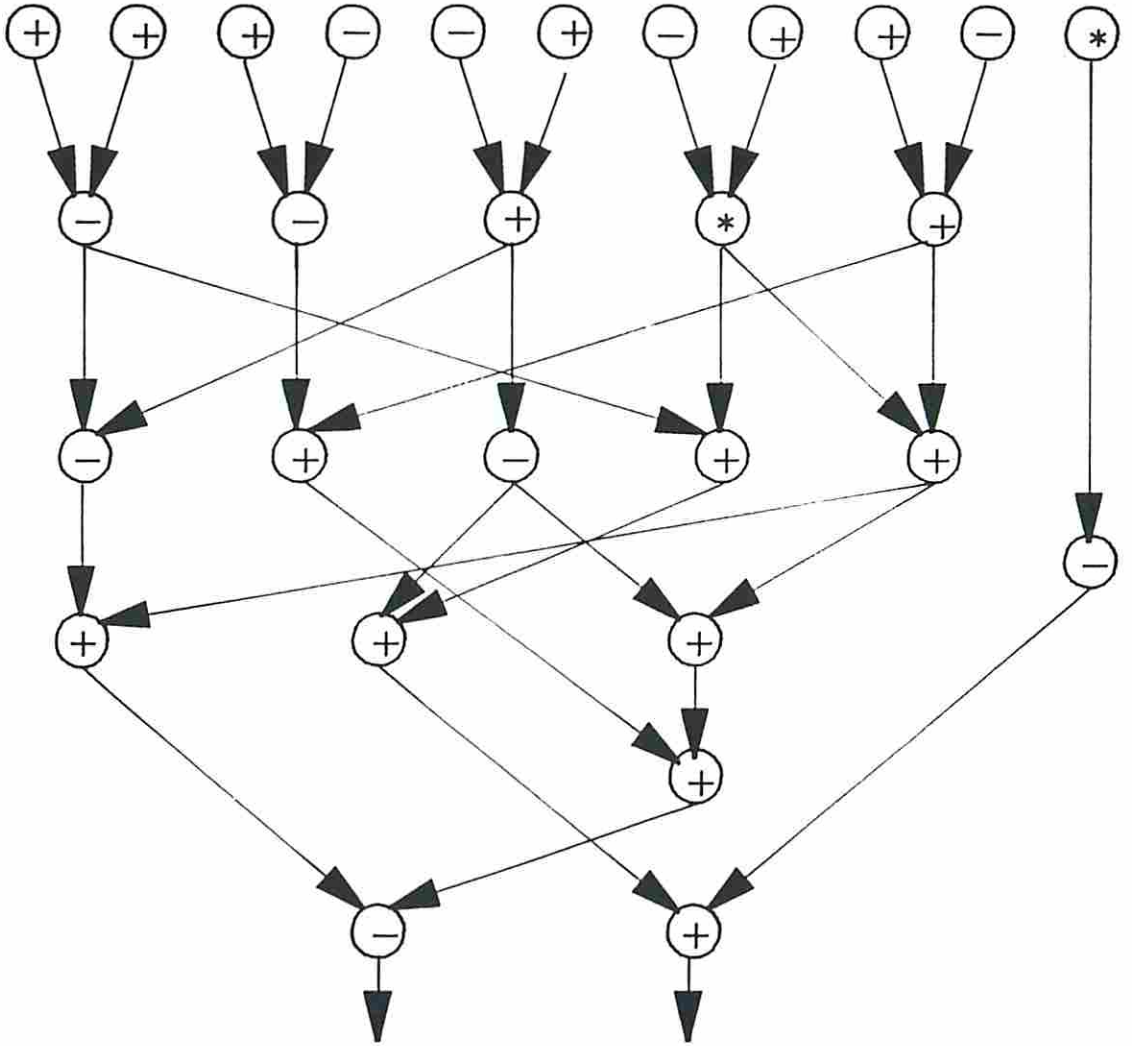


Fig. 6 Random Dataflow Graph

Function	Type	Area (10^4 mils)	Delay (nS)
Adder	Fast	4200	340
	Medium	2880	530
	Slow	1200	1510
Subtracter	Fast	4200	340
	Medium	2880	530
	Slow	1200	1510
Multiplier	Fast	49000	375
	Medium	9800	2950
	Slow	7100	7370
Register (per bit)		15.62	5, 10

Table 1. Module Set Parameters

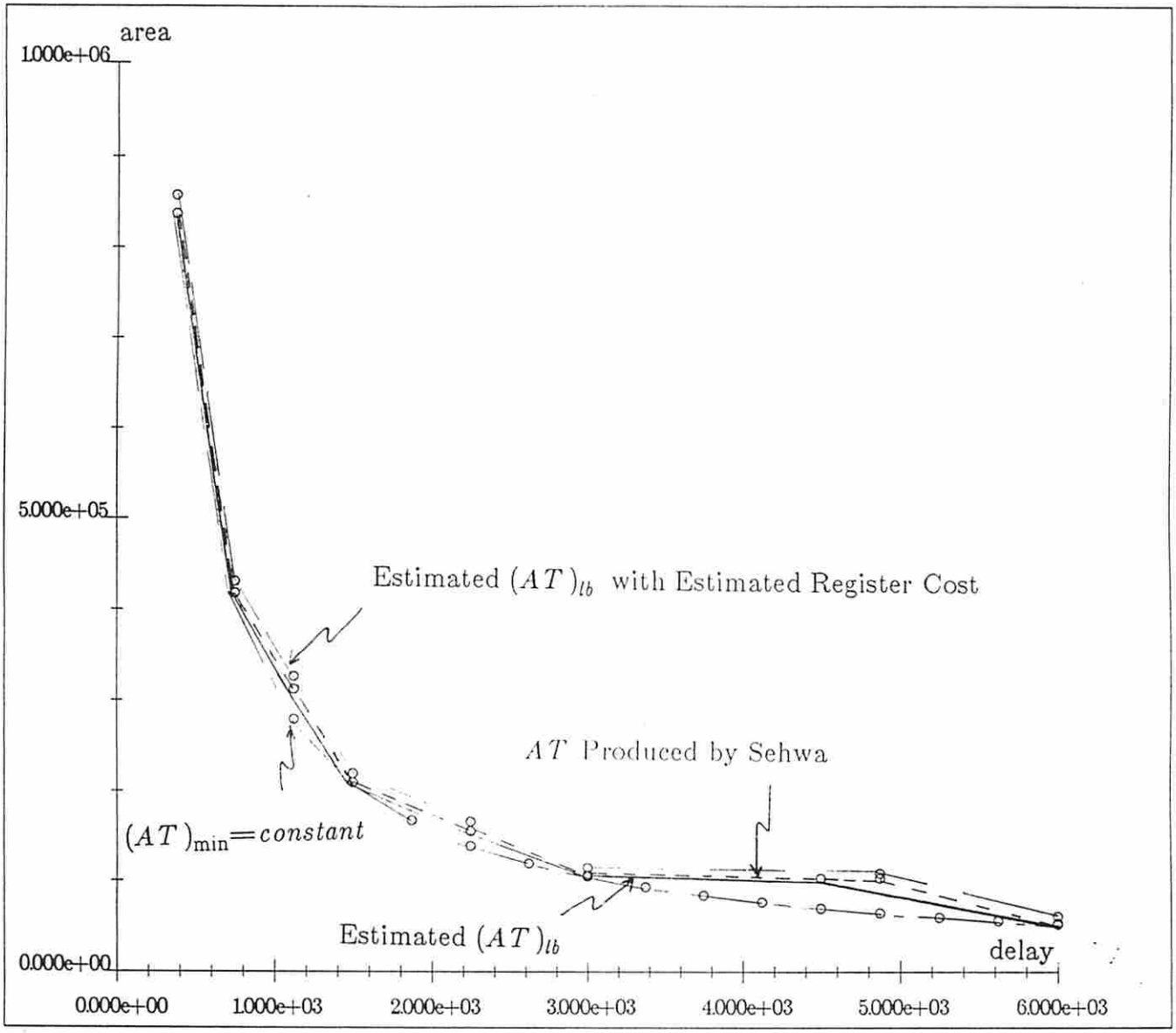


Fig.7 Area-Delay Curves for AR Lattice Filter

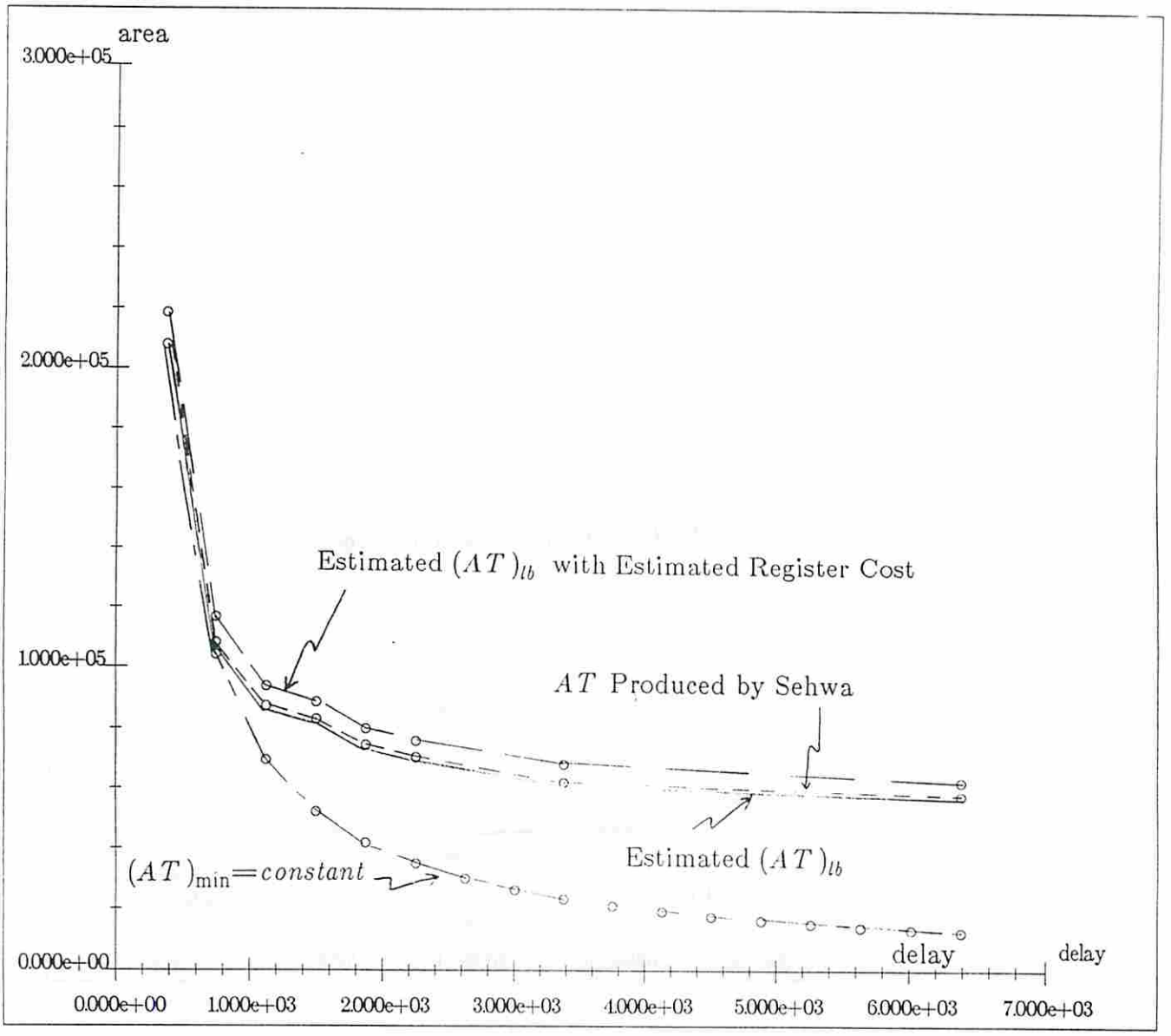


Fig.8 Area-Delay Curves for Random Dataflow Graph

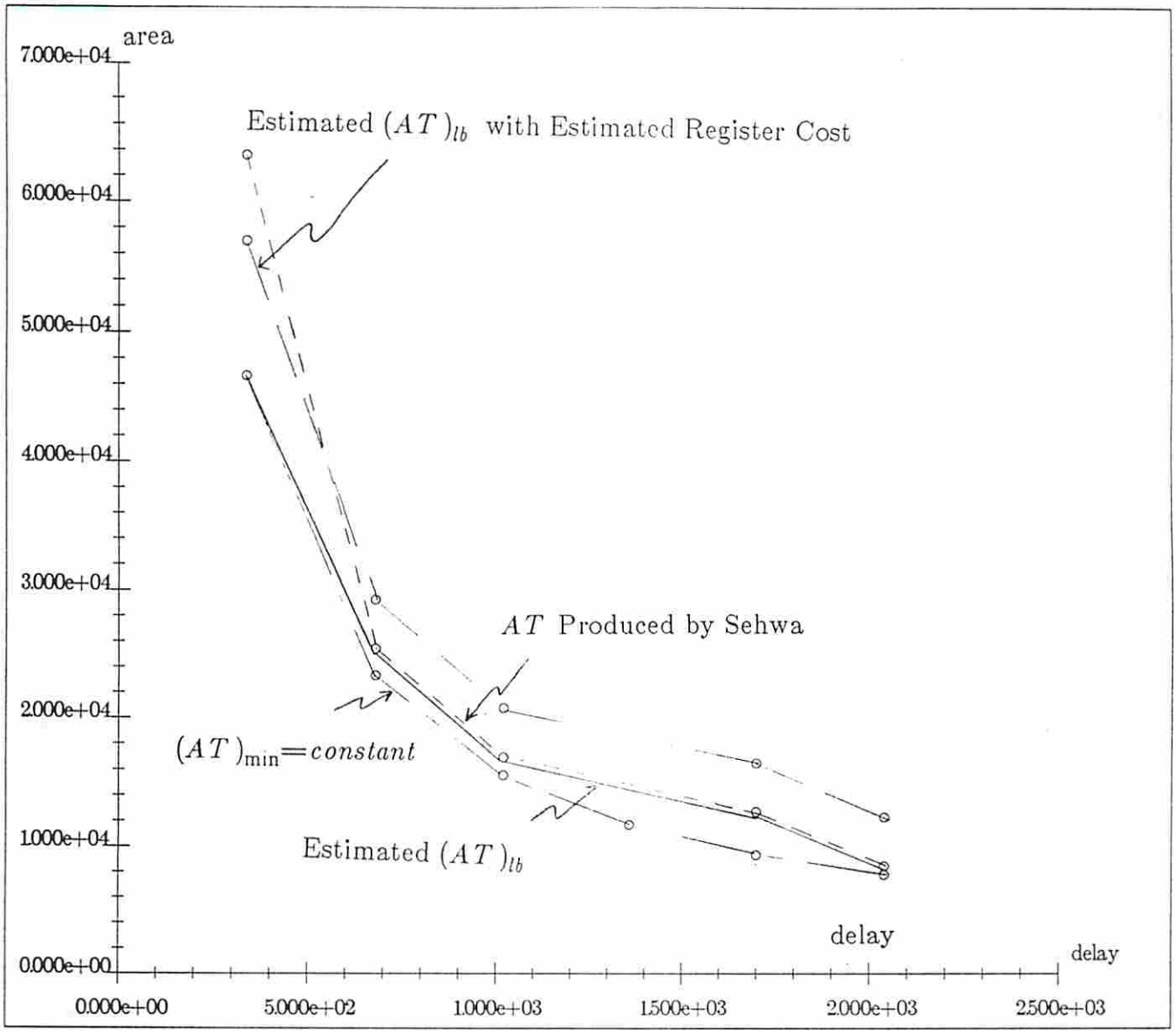


Fig.9 Area-Delay Curves for Conditional Dataflow Graph

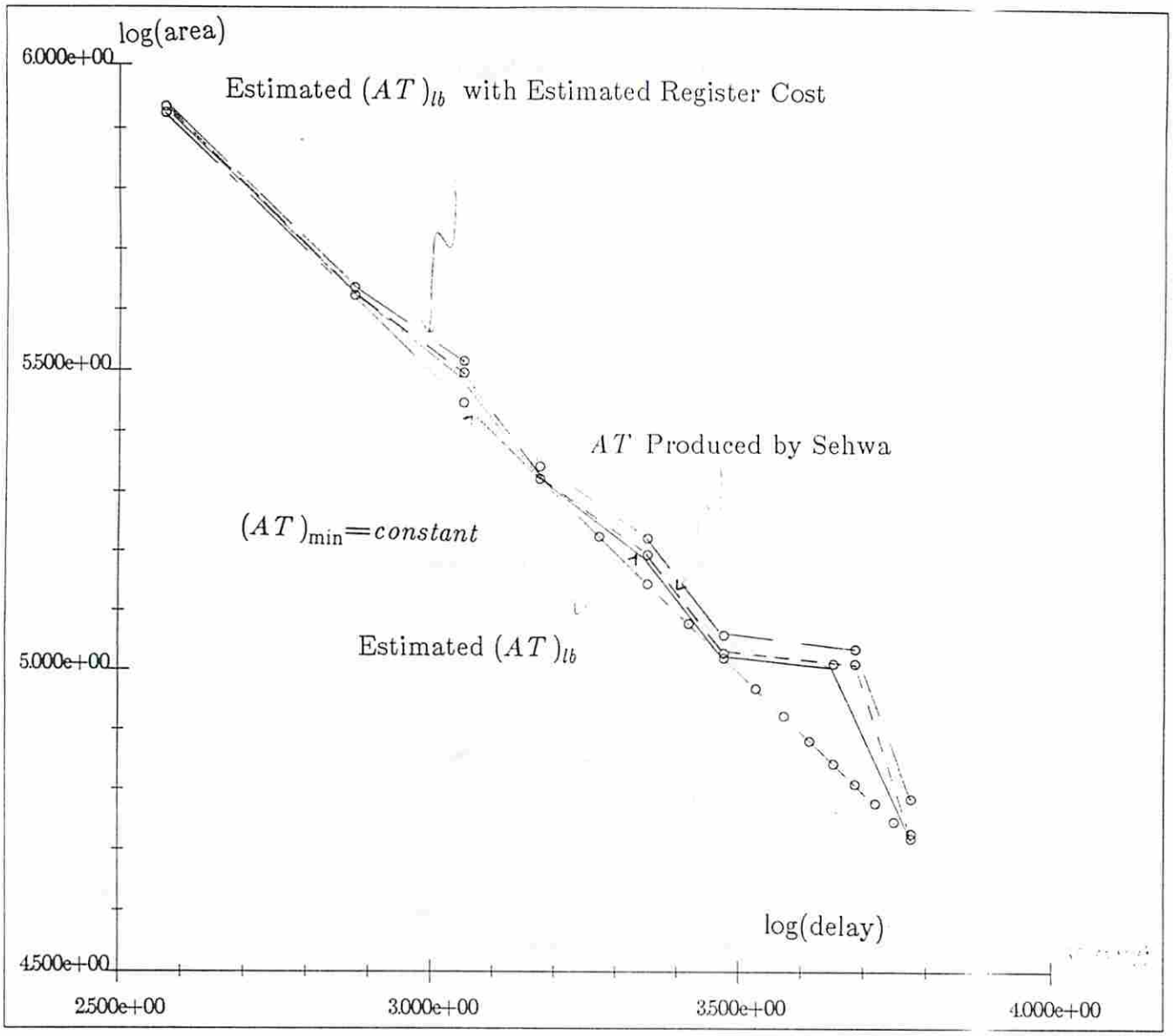


Fig.10 Area-Delay Curves for AR Lattice Filter
(log-log scale)

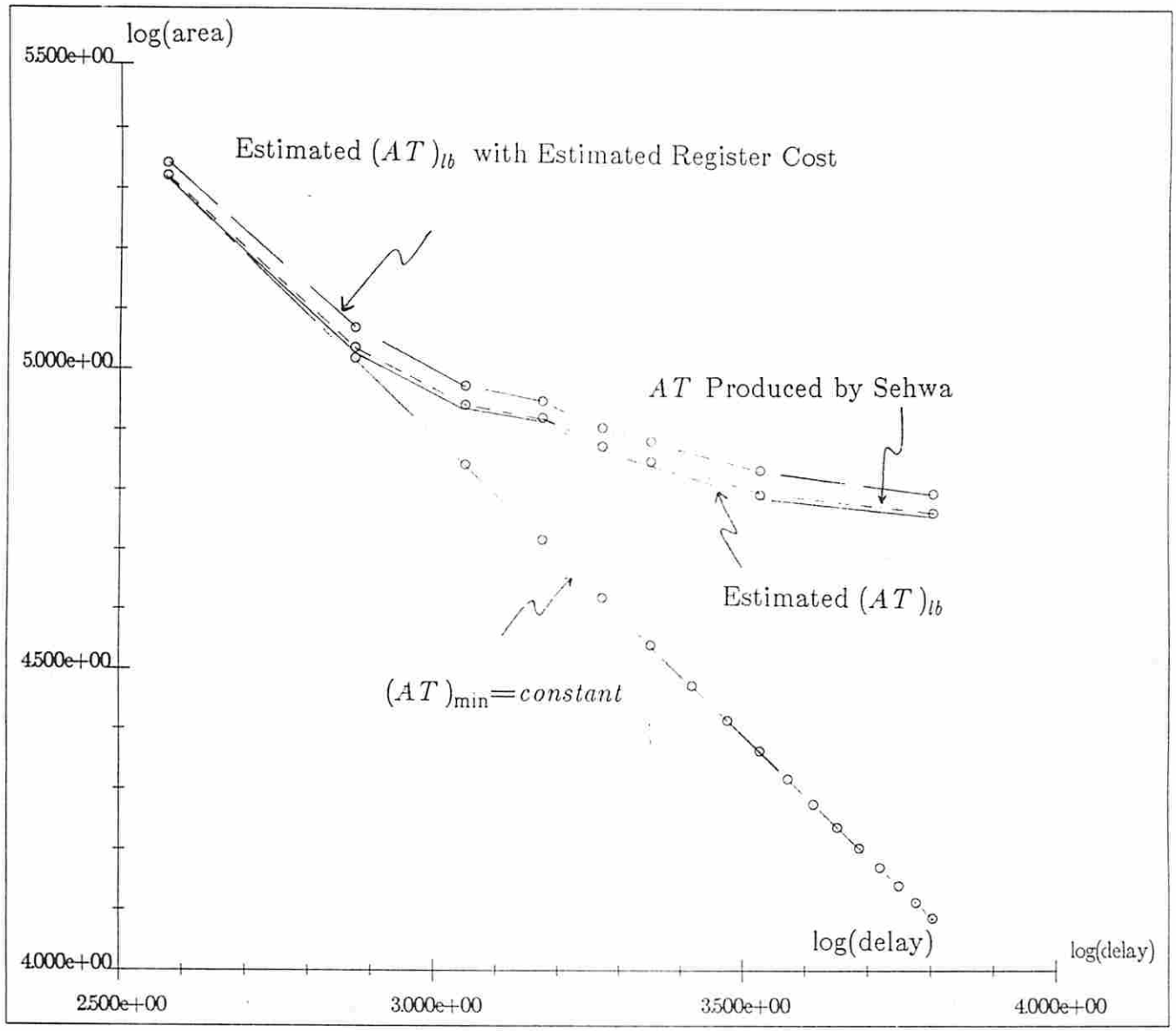


Fig.11 Area-Delay Curves for Random Dataflow Graph
(log-log scale)

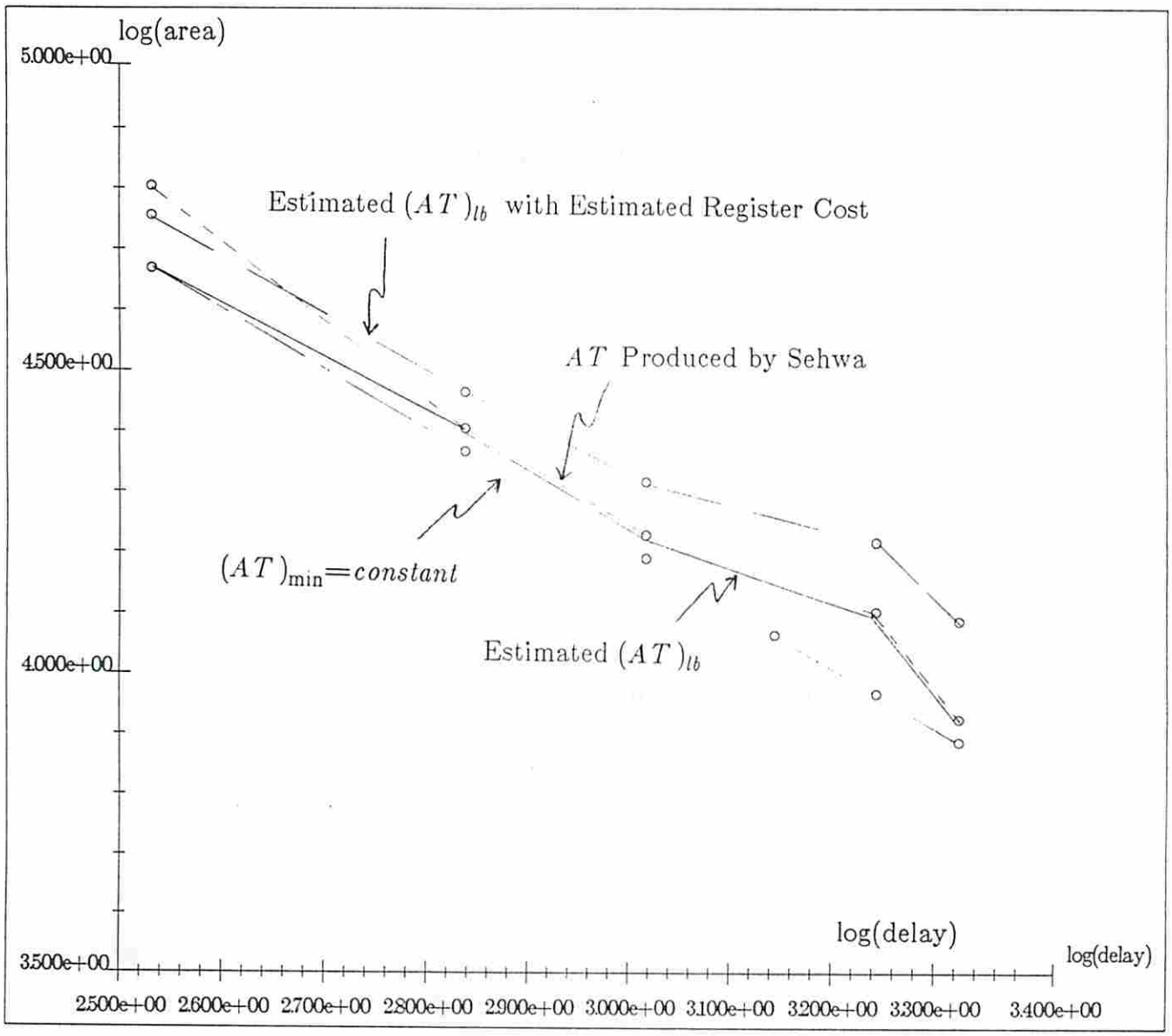


Fig.12 Area-Delay Curves for Conditional Dataflow Graph (log-log scale)