数理解析研究所講究録
第547巻 1985年 173-195

Evaluation of Working Set Algorithms for Data-flow Machines

砂原 秀樹　　所 真理雄
Hideki Sunahara and Mario Tokoro

Department of Electrical Engineering
Keio University
Yokohama 223 JAPAN

ABSTRACT

This paper extends the algorithms for implementing the working set concept for data-flow machines proposed in [7] in order to cope with dynamic data-flow programs which include branches, loops, and procedure calls. The algorithms for computing the D-, E-, and L-levels are revised and the performance for static data-flow programs is reevaluated. Algorithms to manage branches, loops, and function calls are developed. The effectiveness of these algorithms is shown through computer simulation.

1. INTRODUCTION

In recent years, various data-flow architectures have been proposed as solutions to the need for parallel computation [8]. Data-flow architectures are advantageous to conventional and other parallel computing architectures in their capabilities for automatic detection and utilization of maximum parallelism in a program at the execution time.

Given a finite number of resources (e.g., execution units and instruction memory), however, a data-flow program cannot always utilize the resources efficiently. This is because a non-deterministic selection among many ready instructions may choose other than the most critical ones, and this would result in some execution units being idle for a period of time. Thus, schemes to maximaly utilize the parallelism that is provided by hardware are required.

As an answer to this requirement, the authors introduced the working set concept for data-flow machines, proposing a data-flow architecture with hierarchical memories. The algorithms to compute the D-, E-, and L-levels were proposed. Segmentation, fetch, and removal/replacement policies were devised. These algorithms/policies were shown to be effective for static data-flow programs through computer simulation [7].

However, since almost all the practical programs contain branches, loops, and procedure calls, it is necessary to devise algorithms for these program constructs. In this paper, we revise these algorithms, and develop new algorithms for dynamic data-flow programs with branches, loops, and procedure calls.

In section 2, we review the working set model. In section 3, we revise the D-, E-, and L-level algorithms. In section 4, a thorough evaluation of the algorithms/policies for static data-flow programs is made through computer simulation. In section 5, we propose algorithms to manage branches, loops, and procedure calls for dynamic data-flow programs. In section 6, we use computer

simulation to evaluate the evaluate the proposed algorithms. algorithms also by computer simulation. In section 7, we discuss some related issues.

2. THE WORKING SET MODEL

A data-flow machine has multiple execution units. A data-flow program is represented as a directed graph, where a node indicates an operation and a directed arc indicates the data dependency between the two nodes connected by it.. Since each instruction which has all of its input data available can be executed in parallel, the execution of a program proceeds like a wave front on a graph.

In static data-flow programs, the principle of locality does not fully apply as it does in conventional machines because in such machines each instruction is executed only once. Instead, we should establish the working set concept base on the **simultaneity of execution** of a program.

Three quantities will be associated with each node of a static data-flow graph: the dependency-level (D-level), the earliest execution level (E-level), and the latest execution level (L-level). The D-level of a program is the minimum number of spans from the entry node. The E-level of a program is the maximum number of spans from the entry node. The L-level of a program is the maximum number of spans from the exit node. The rule of determining these quantities for each node Nj of an acyclic graph G(N,A) are shown below, where We(Nj) represents the execution time of node Nj.

D(Nj): D-level for each node Nj of a graph G(N,A)

```
{ sum := 0;
  for all Nj of N { sum := sum + We(Nj) };
  for all Nj of N { D(Nj) := sum };
  V := empty;
  for all Nj of N where Nj has no input arcs
    { D(Nj) := 0; V := V + Nj };
  while V <> empty
    { W := empty;
      for all Nj of V
        for all Nk which has an arc Ajk
          if D(Nk) > D(Nj) + We(Nj) then
            { D(Nk) := D(Nj) + We(Nj); W := W + Nk };
      V := W;
    };
}
```

E(Nj): E-level for each node Nj of a graph G(N,A)

```
{ for all Nj of N { E(Nj) := 0 };
  V := empty;
  for all Nj of N where Nj has no input arcs
    { V := V + Nj };
  while V <> empty
    { W := empty;
      for all Nj of V
        for all Nk which has an arc Ajk
          if E(Nk) < E(Nj) + We(Nj) then
```

```
        { E(Nk) := E(Nj) + We(Nj); W := W + Nk };
    V := W;
  };
}
```

L(Nj): L-level for each node Nj of a graph G(N,A)

```
{ sum := 0;
  for all Nj of N { sum := sum + We(Nj) };
  for all Nj of N { L(Nj) := sum };
  V := empty;
  for all Nj of N where Nj has no output arcs
    { V := V + Nj };
  min_level=sum;
  while V <> empty
    { W := empty;
      for all Nj of V
        for all Nk which has an arc Akj
          if L(Nk) > L(Nj)-We(Nk) then
            { L(Nk) := L(Nj)-We(Nk);
              if L(Nk) < min_level then
                { min_level := L(Nk) };
              W := W + Nk
            };
      V := W };
  for all Nj of N { L(Nj) := L(Nj) - min_level };
}
```

In Fig. 1 the D-level, E-level and L-level are shown for each node.

Assuming that a machine incorporates an infinite number of execution units, the D-level of a node can be referred to as the level (or timing) at which at least one of its input become available. The E-level of a node can be referred to as the level (or timing) at which all of its inputs become to available. The L-level referres to the latest possible execution timing without increasing the execution time. Thus, an instruction can be executed without increasing the execution time of a program at any timing in between its E-level and L-level. The instruction whose E-level is equal to its L-level is called the critical node.

However we could not usually have a sufficient number of execution units to execute a program at its maximum parallelism. Thus we must consider executing a program with a limited number of execution units. In order to execute a program with a limited number of execution units, we need to distinguish a set of instructions which would increase the total execution time unless they would be executed at the next timing. The key to efficient execution with a limited number of execution units is how to select these critical nodes before all of the execution units become busy by uncritical instructions. Thus, the working set of a data-flow machine is defined as the set of instructions which will fire at the next timing.

3. WORKING SET POLICIES FOR STATIC DATA-FLOW PROGRAMS

3.1. A Model of the Data-Flow Machine

Fig. 2 shows the model for data-flow machines. Each component of the machine is briefly described as follows:

Execution Units(EU): Each execution unit can execute an instruction. All the execution units can execute in parallel. The execution units have an instruction queue.

Primary Instruction Memory(PIM): This memory contains active instructions. Each word of this memory can store a data-flow machine instruction. An instruction can take in input data only when it is placed in this memory. This memory is content-addressable in order to speed up the update of operand data (or ready tags).

Secondary Instruction Memory(SIM): This memory contains dormant instructions. This memory is much larger than the PIM and has the capability of reading a block of memory cells at a reasonably high speed.

Result Memory(RM): The result memory temporarily stores result data until they are taken in by all the instructions that use them. This memory is premium because of its high-speed access in concert with the associative function of the PIM.

3.2. Basic Policies

We have proposed three kind of basic policies for our data-flow machine model: instruction segmentation, fetch, and removal/replacement policies[7]. Segmentation policies define the size of a segment for transferring between the PIM and the SIM. There are two basic policies for segmenting a data-flow program: to partition a program to i) fixed size segments and ii) variable size segments. In fixed size segmentation, the size would correspond to the physical block size. In variable size segmentation, each segment can correspond to a logical set of instructions. A segment might be subdivided into a fixed size block before it is transferred between the PIM and the SIM. There are several auxiliary policies, as shown in the following:

ISP-1: segmentation by each instruction,

ISP-2: segmentation by different D-, E-, or L-levels,

ISP-3: segmentation by conditional instructions, and

ISP-4: segmentation by loop bodies, subroutines, or procedures.

The granularity of segments varies from finer to coarser with respect to ISP-1 to ISP-4. The finer the granularity, the lower the possibility of primary memory fragmentation, but the control overhead would be higher. In contrast, the coarser the granularity, the higher the possibility of primary memory fragmentation, but the control overhead would be lower. For ISP-2 to ISP-4, if the size of a segment is very large, further segmentation within a segment might be required. Segment size relates to the transfer time. If the transfer time is long, multiprogramming should be employed.

The instruction fetch policies refer to deciding when and which segments to transfer from the SIM to fill up the free memory cell in the PIM. Such policies can be based either i) on demand or ii) according to anticipation. Four basic fetch policies are shown below:

IFP-1: Fetch by result availability.
As soon as a result arrives, one or more segments of instructions which use that result as their input are retrieved and transferred from the secondary memory. This policy can be classified as demand-based.

IFP-2: Fetch by increasing D-level.
It is assumed that segments are stored in the secondary memory by their D-levels. As soon as any primary memory cell is free to store a new segment, a segment is transferred to the memory cell according to increasing D-levels. This policy is anticipatory in nature.

IFP-3: Fetch by increasing E-level.
This policy is the same as IFP-2 except that E-levels are used instead of D-levels. This policy is also anticipatory.

IFP-4: Fetch by increasing L-level.
This policy is also the same as IFP-2 except that L-levels are used instead of D-levels. This policy is anticipatory.

IFP-1 is the same as that proposed by [6]. In this policy, the possibility of an instruction residing a long time in the PIM memory is higher. Thus, the size of the PIM is large. IFP-2 is similar to IFP-1 except it is anticipatory. Thus, the size of the PIM is also large. Since this policy does not fetch instructions in order of execution, a dead-lock may occur with a small PIM. IFP-3 and IFP-4 would require less PIM than IFP-2, but both of them require an RM in which the result data waits until they are used (taken in) by instructions.

The most important but difficult issue is that for the instruction removal/replacement policies. Removal/replacement policies refer to deciding which (if any) instructions to remove from the PIM so that new instructions can be transferred from the SIM.

Let us call a **removal** policy a policy that determines which instruction can simply be discarded. There are two reasons for an instruction simply being removed:

IRP-1: Remove for execution.
An instruction is removed from the primary memory to be sent to an execution unit.

IRP-2: Remove never-executed instructions.
An instruction which is determined to never has been executed is removed. Some special mechanism would be required to detect never-executed instructions (describe later).

Let us call a **replacement** policy a policy that determines which segment will be replaced by another active segment. The segment to be replaced will simply be discarded if there has been no change in the state of any of its instructions; otherwise the copy of the segment in the swap area of the SIM will be updated according to the change. There are policies such as the following:

IRP-3: First-In First-Out (FIFO).

IRP-4: Last-In First-Out (LIFO).

IRP-5: Least Recently Updated (LRU).

Actually, we want to develop a fetch policy which does not require IRP-2 to IRP-5. However, this is not possible since there remain instructions which will never be executed in the primary instruction memory. When a program has a branch, IRP-2 will be needed. IRP-3 and IRP-5 could be effective in detecting such instructions. IRP-4 implies a considerable overhead.

4. EVALUATION FOR STATIC DATA-FLOW PROGRAMS

Computer simulations have been done to evaluate the performance of the proposed policies for static data-flow programs.

IRP-1, when it is used solely with IFP-1 or IFP-2, may cause a deadlock. It can be used with IFP-3 or IFP-4, since instructions that are fetched tend to be executed sooner than others. The removal policy IRP-1 should be used with one of the replacement policies IRP-3 to IRP-5. IRP-3 to IRP-5 can be used with any fetch policy. IRP-2 is not needed for static data-flow programs since they do not include branches. Thus, simulations have been done for these possible policy combinations.

The unravelled 5th order Gaussian elimination program was used in our simulation. The programs consisted of 145 instructions, containing an average parallelism of 9.66. Thus, we decided to use ten execution units.

4.1. Scheduling Effect (case 1)

Since the purpose of simulation is to investigate the fundamental behavior of data-flow programs under a limited resource environment, we have made the following assumptions in the simulations:

(1) a segment consists of one instruction,
(2) the execution time of each instruction constant and is one unit of time, and
(3) all the other consumed time, such as the transfer time between the PIM and the SIM, are almost zero.

First, the execution times, utilization factors of execution units, and the required sizes of the result memory for the possible combinations of the policies are shown in Fig. 3 (a), (b), and (c). All the different removal/replacement policies (unless specified are represented by one curve because there was little difference between them). This is because the efficiency of IRP-1 is higher than IRP-3, 4, and 5.

IFP-2 shows the worst performance in execution time. IFP-1 and IFP-3 show better performances in the execution time. IFP-4 presents the best performance in execution time. It is very interesting to observe that when the size of the primary memory increases, the execution time increases asymptoticly to that of IFP-3. This is explained as follows: when the primary memory size exceeds the number of L-level instructions executed at the next time unit, there is the possibility that non-critical instructions will be selected first. (This situation could happen in IFP-3, although it is not observed in the simulations.)

The graph of the utilization factor of the execution units is the inverted shape of the graph of the execution time. This is because the utilization of the execution units is reflected in the execution time.

IFP-1 requires the smallest result memory. An increasingly larger result memory is required from IFP-2 to IFP-3. And the required size of the result memory is largest with IFP-4. This is because a result has to wait until it is taken in at the latest E-level or L-level among the instructions that use it.

4.2. Instruction Execution With Weight (case 2)

With exception of the execution time of each instruction, the assumptions are same as case 1. The execution time of each instruction was defined as follows:

ADD,SUB	:	1	(0.57)
MUL	:	2	(1.14)
DIV	:	3	(1.71)

The execution time and the required size of the result memory for the possible combinations of the policies are shown in Fig. 4 (a) and (b). Each result shows the same characteristics as case 1. This shows the efficiency of the working set algorithms in this situation.

4.3. Random Execution Time (case 3)

With exception of the execution time of each instruction, all the assumptions are also same as case 1. In this case, the execution time of each instruction varies according to the values of the data. In the simulation, execution time followed the random function of the negative exponential with a mean value of 1. Note that D-, E-, and L-levels cannot be properly computed by the compiler. Thus, the compiler computes these levels assuming the execution time of each instruction is constant and is one time unit.

The execution time and the required size of the result memory for the possible combinations of the policies are shown in Fig. 5 (a) and (b).

When the number of cells in the PIM is 1 or 2, the execution time is the shortest with IFP-1. However, when the PIM size is larger, the execution times with IFP-3 and IFP-4 are shorter. This is explained as follows: when the PIM size is small, the possibility that non-critical instructions will be fetched is high. On the other hand, the execution time becomes shorter with a larger PIM size since result data hits the pre-fetched instructions.

The required sizes of the result memory show the same tendency as described in case 1.

4.4. Swapping Overhead (case 4)

In the discussion so far, except for the execution time of each instruction we assumed that all the consumed time is almost zero. Here, let us assume the following:

(1) a segment consists of some instructions,
(2) the number of cells in the primary instruction memory is thirty-two,
(3) the execution time of each instructions is constant and is one unit of time,
(4) the time to transfer one segment between the PIM and the SIM is one unit of time, and

(5) All the other consumed time, such as the access time for each memory, is almost zero.

The execution time with respect to the number of instructions in one segment the possible combinations of the policies except IFP-2 are shown in Fig. 6. A dead-lock occur with IFP-2 because this policy does not fetch instructions in the order of execution.

When the size of the segment is one, the execution time is determined by the number of swaps between the PIM and the SIM. At the mean time, most execution units are idle. However, when the size of the segment is larger (the size of the PIM is larger), the execution times are shorter. This is explained as that the execution units become busy because the active instructions are increased in the PIM. But when the size of the segment become too large, the execution time become longer or similer. This is the result of the fragmentation.

In IFP-3 or IFP-4, the segment are divided by different E- or L-levels. This is ISP-2. Since there are not always enough node in same level to fill the segment, the too large segment makes a lot of empty cells in the segment. Because of the point of the resource utilization, the size of segment would be small.

4.5. Synthetic Evaluation for Static Data-flow Programs

In order to evaluate each algorithm we should define elapsed times, such as for the matching in the PIM, sending fired instructions to the execution units, and getting a result token from the execution units in the RM. However, it is very difficult, since our model is too abstract, to define such elapsed times exactly.

Thus, we propose a measure for evaluation called the figure of merit. The figures of merit were given by:

$$(Time)^2 \times (PIMsize)$$

where *Time* is the execution time of the program, and *PIMsize* is the number of cells in the PIM. The first term is concerned with the execution time and the second term is concerned with the cost of the memories. The PIM is the most premium because of its content addressable capability and its high-speed of access. Thus, other cost can be negligible.

All the assumptions are same as case 4 except for the (1) and (2). We made the size of the PIM and the fetch segment equal. This assumption seems strange. However, since we expand our concept to multiprogramming environment, each program is given the minimum number of cells in the PIM which the program requires. Thus we think that these case can be seen in the multiprogramming environment. The graph of execution speed and these figures of merit for each policy with assumptions of case 1 is shown in Fig. 7 (a) and (b). According to these results, we conclude that IFP-3 and IFP-4 give the highest performances. And, best point is that the sizes of the segment and the PIM are equal 8. This value means the average parallelism of this program.

5. WORKING SET POLICIES FOR DYNAMIC DATA-FLOW PROGRAMS

5.1. Dynamic Data-flow Programs

We call a program whose data-flow graph change at the execution time a dynamic data-flow program. For example, assume that a program includes a function call. Before execution, the function call is represented by one node in the data-flow program. When the function is called in the course of execution, that node will be replaced by the data-flow graph for the body of the function. The data-flow program for the body is then executed the same as other nodes. Loops are executed in the same manner. When the number of times a loop will be executed can not be determined before execution, we call the dynamic character of the program high.

There are two basic schemes for the replacement of a function call node by its body: i) to copy the body and link to the executing graph and ii) to provide a token identifier called **color** to each instruction in the body [4]. In the former scheme, the copy overhead would be high, but there would be no control overhead. There is no copy overhead in the latter, but the control overhead for token identifiers would be higher.

In our data-flow model, a data-flow graph which includes branches is changed at the execution time, since the instructions that have not been and will not be executed are removed from the data-flow program. Thus, data-flow programs which include branches, loops, and function calls are called dynamic data-flow programs in our models. In this section, we describe management algorithms for dynamic data-flow programs.

5.2. Branch Management Algorithms

In order to execute a branch instruction, the following four instructions have been added: test, T-gate, F-gate, and merge instructions. A test instruction checks conditions and sends its result (called a control token) to the next instructions. T-gate and F-gate are gate instructions. If a control token at input 0 of a T-gate is true, it passes the data at input 1 to the next instructions. If a control token is false, the data at input 1 is discarded. An F-gate works inversely. A merge instruction will pass the data that first reaches one of its input links to the next instructions and the other input data will be discarded.

As is the nature of branch instructions, one of the two sets of instructions divided by a branch instruction will not be executed. Thus, such instructions need to be removed from the PIM. There are three algorithms for the management of never-executed instructions, as shown in the following:

BMA-1: No strategy.
Never-executed instructions wait to be replaced by IRP-3, 4, and 5.

BMA-2: Propagation of NULL tokens.
When a T-gate or an F-gate discards data, it sends a special token called the NULL token. A NULL token is the same as other tokens, except that the instruction which receives the NULL token is not executed sends a NULL token to next instructions. A merge instruction does not send the NULL token when it receives one.

BMA-3: Remove an instruction when it receives a NULL token.
This algorithm is similar to BMA-2 except that an instruction which receives a NULL token will immediately be removed from the PIM and it then sends a NULL token to the next instructions.

BMA-1 is not compatible with IFP-2, 3, and 4, because these anticipatory fetch policies do not go to the next level until all the instructions at the current

level have been executed. In the case of the combination with IFP-1, cells in the PIM are not wasted by never-executed instructions. This is because IFP-1 does not fetch never-executed instructions.

BMA-2 is compatible with all of the instruction fetch policies. In case of the combination with IFP-2, 3, and 4, never-executed instructions are transferred to the PIM just in order to receive the NULL tokens to be removed. Thus, cells in the PIM are wasted by never-executed instructions. With IFP-1, the cells in the PIM are not wasted by never-executed instructions, provided there is a mechanism which prevents instruction transfers from the SIM to the PIM that are initiated by NULL tokens. However, NULL tokens will be accumulated in the RM; thus we need to introduce another mechanism to solve this problem.

BMA-3 works like BMA-2. The utilization of the PIM would be higher than BMA-2. For further improvement of the utilization of the PIM, we must devise an algorithm that does not fetch the never-executed instruction, such as is shown in the following:

BMA-4: Fetch one of the alternative segments.

The programs are segmented by the conditional instructions as shown in Fig. 8. Each segment is treated equally until the test gate tests the condition. After the condition is tested, the disuse segment will not be fetched from the SIM. The instructions which have already been fetched from the PIM will reside in the PIM until being removed by BMA-2 or 3.

BMA-4 would improve the utilization of the PIM, but segment control overhead will be high.

5.3. Loops and Function Calls

As stated above, loops and function calls in data-flow programs are performed by replacing a calling node by its body. The conventional data-flow architecture uses the coloring method for preventing copy overhead.

We can use the coloring method in our model with ISP-4 and by holding all the loop body or function body instructions in the PIM. However, this is not always possible since there might be a body larger than the size of the PIM. Also, the coloring method seems not to be compatible with prefetching schemes using D-, E-, and L-levels. Thus, we decided not to take the coloring method for sharing body instructions, although we use color to identify tokens.

Since instructions are always copied from the SIM and the PIM, we can use this mechanism to replace a call node by its body instructions. We treat loops as a tail recursion, so that loops can be managed as functions.

There is a question of how to give a level for each instruction in a program which includes loops and function calls. In the case of "for" loops and function calls, a level can be given to each instruction by unravelling. However, we cannot give a level to an instruction in a "while" loop or recursive function. Thus, the level of an instruction for such program constructs has to be given dynamically at the execution time.

Hierarchical leveling is one method which gives levels dynamically. In the SIM, each data-flow graph is stored as like shown in Fig. 9 (a). When a function call node is fetched, the instructions for this function body are fetched. In this case, each instruction has a hierarchical level as is shown in Fig. 9 (b). By using the same algorithm, we can give levels for instructions for recursive programs.

This hierarchical leveling can be applied IFP-2, 3, and 4 described in section 3.

6. EVALUATION OF DYNAMIC DATA-FLOW PROGRAMS

6.1. Evaluation of the Branch Management Algorithms

First, the branch management algorithms are evaluated. We have made the following assumptions in the simulations:

(1) a segment consists of four instructions when the size of the PIM is not smaller than four. When it is smaller than four, the size of the segment is equal to the size of the PIM,

(2) the execution time of each instruction is constant and is one unit of time,

(3) the time to transfer one segment between the PIM and the SIM is one unit of time, and

(4) All the other consumed time, such as the access time for each memory, is almost zero.

The size of segment was determined by considering the parallelism of the program. BMA-1, when it is used with IFP-2, 3, or 4, may cause a deadlock. It can be used with IFP-1. BMA-2, 3, and 4 can be used with any fetch policy. Thus, simulations have been done for these possible combinations of the policies.

We used a test program which consisted of 43 instructions containing the average parallelism of 2.24. Thus, three execution units were provided. The execution times in **then case** and **else case** are shown in Fig. 10 (a) and (b).

Since the length of the **then case** in the test program is longer than the **else case**, the execution time of the **then case** are longer than the **else case**. Except for BMA-4 there is no difference in the execution time between algorithms. BMA-4 shows best performance, since this algorithm avoid wasting the PIM by never-executed instructions. BMA-2 and 3 show little difference in execution times. This is explained as that difference of timings of first and second input data become available is small.

The graph of the figure of merit that was defined in section 4 is shown in Fig. 11. From this result, we conclude that BMA-4 shows the highest performance with IFP-3 or 4.

6.2. Efficiency of the Hierarchical Leveling

The assumptions in this simulation are the same as those described above. We used two test programs consisting factorial algorithms: one uses a loop *(lfact)* and the other uses the divide-and-conquer method *(dcfact)*. These programs contain the average parallelism of 1.74 and 4.13, respectively, when the factorial of 5 is computed. The execution times and the figures of merit were shown for each program in Fig. 12 (a), (b) and Fig. 13 (a) (b).

IFP-3 and IFP-4 with BMA-4 show the highest performances in the *lfact*. In the *dcfact* it is very interesting to observe that the graph of the execution times shows a shape similar to that of the execution times of the static data program in case 3. This is explained as follows: when a level for each node is calculated, the weight 1.0 is assumed for a function call node. However, a function body usually has a longer execution time. This means the prediction fails. The figure of merit for this is shown not to be good. Thus, we should consider the management algorithms for functions. We perspect that function should be treated as individual processes.

7. RELATED ISSUES

7.1. Data Memory Management

In the discussion so far, it has been assumed that only scalar data are used for computation. The execution of an instruction returns the result value with its id or a list of destinations. However, when structured data is required, it is not possible to return values as we can do with scalars; values are too large to be sent. Thus, methods are proposed in which the execution of an instruction returns only the result id; the result value is stored in a data memory for structures called structure memory [1,2,3]. When the structure memory becomes large, it would be economical to provide a two-level structure memory. Thus, the working set for data should be considered.

We have proposed the active token count for the structure memory [5]. Each cell in the structure memory is given a reference count for tokens called the active token count. The fetch policy moves data cells from the Secondary Data Memory (SDM) to Primary Data Memory (PDM) according to the value of their active token counts. So far, not enough simulations have been done to present the performances for the structure memory in general.

7.2. Process Priority in Multiprogramming

In the above discussion, we assumed that only a single program is executed at a time. However, multiprogramming is efficient from the resource utility viewpoint. Data-flow machines can easily execute multiple programs because of their functionality. However, there is a problem concerning how to give a priority to a process and how to manage prioritized processes.

There is one method wherein a priority is given to each token. Multiple queues classified by priorities are placed in between the PIM and the EU. In our data-flow model, only instructions which are fetched to the PIM can be executed. Thus, we can fetch a segment of a higher priority process before that of a lower priority process. This simple policy could manage multiprogramming with prioritized processes very well.

In this context, **the working set size** of a program can be considered to be the lower bound of the required number of cells in the PIM without extending execution speed. Thus, the working set size is defined as follows:

The working set size is the parallelism of a program at each execution moment.

The number of processes which are concurrently executed in a data-flow machine should be limited according to the working set sizes of the processes.

8. CONCLUSION

This paper has proposed several working set algorithms for data-flow machines giving one criterion for the construction of cost-effective general-purpose data-flow machines. This concept is based on the **simultaneity of execution** of a data-flow program.

Segmentation, fetch, and removal policies have been proposed based on the working set concept for regulating the flow of instructions in a data-flow machine in order to give efficient utilization of limited resources. Among these policies, the fetch policies based on increasing L- and E-levels for static data-flow programs showed the best performance. This would be because of the anticipatory nature of L- and E-level scheduling. Further improvements could be

achieved by a policy combining L- and E-level scheduling.

When the size of a segment for instruction transfer is equal to the average parallelism of the program, the figure of merit is shown to be the best with a small size of the PIM.

Branch management algorithms have also been proposed. The removal of an alternative segment worked well. Fetch policies based on the hierarchical leveling for dynamic data-flow graphs were evaluated for loops and recursion. Fetch policies based on increasing L- and E-level scheduling work well without the replacement policies.

The number of processes in multiprogramming. The evaluation of the structure memory management algorithm is left for future work.

REFERENCES

[1] W. B. Ackerman, "A Structure Processing Facility for Data Flow Computers," *Proc. 8th International Conference on Parallel Processing*, 1978.

[2] W. B. Ackerman, "A Structure Controller for Data Flow Computers," CGS Memo 156, MIT, Jan 1978.

[3] Arvind and R. Thomas, "I-structures: An efficient data type for functional languages," Univ. of California Irvine Technical Report, 1980.

[4] Arvind and V. Kathail, "A Multiple processor Data-Flow Machine that Supports Generalized Procedures," *Proc. 8th Annual Symposium on Computer Architecture*, Mineapolis, Minnesota, 1981, 291-302.

[5] H. Asakura, H. Sunahara and M. Tokoro, "A Study on the Structure Memory for the Data-flow Machines with the Working Set Concept," *Proc. of the 28th National Conference of Information Processing Society of Japan*, Tokyo Japan, March 1984, 4F-10 (in Japanese).

[6] J. B. Dennis and D. P. Misunas, "A preliminary Architecture for a Basic Data Flow Processor," *Proc. 2nd Annual Symposium on Computer Architecture*, Jan 1975, 126-132.

[7] M. Tokoro, J. R. Jagannathan and H. Sunahara, "On the Working Set Concept for Data-Flow Machine," *Proc. of the 10th Annual Symposium on Computer Architecture*, Jun. 1983, 90-97.

[8] P. C. Treleaven, D. R. Brownbridge and R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *ACM Computing Surveys 14*,1 (Mar. 1982), 93-143.

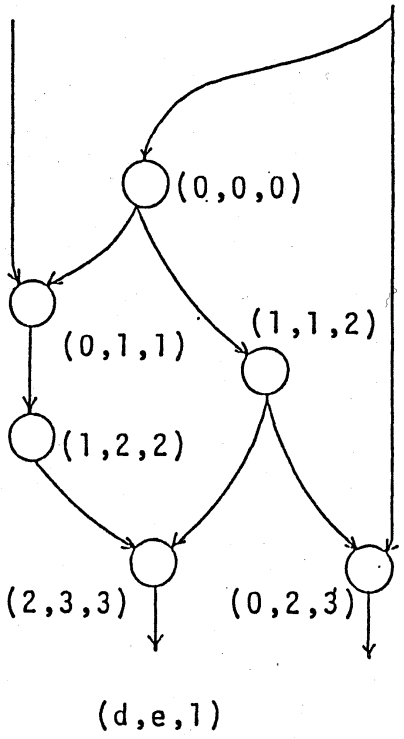
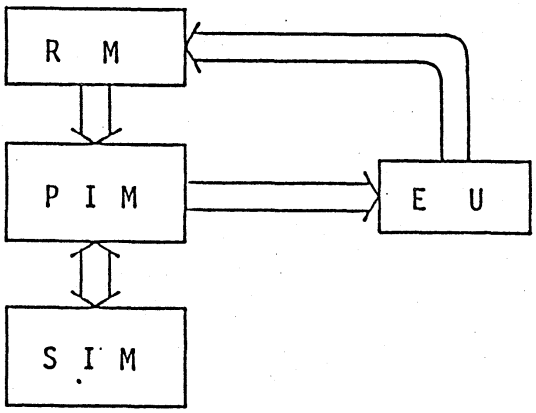


Fig. 1 D-, E-, and L-levels



EU(Execution Units): execute instructions.
PIM(Primary Instruction Memory): this memory contains active instructions.
SIM(Secondary Instruction Memory): this memory contains dormant instructions.
RM(Result Memory): temporarily stores result data.

Fig. 2 The model of data-flow machine

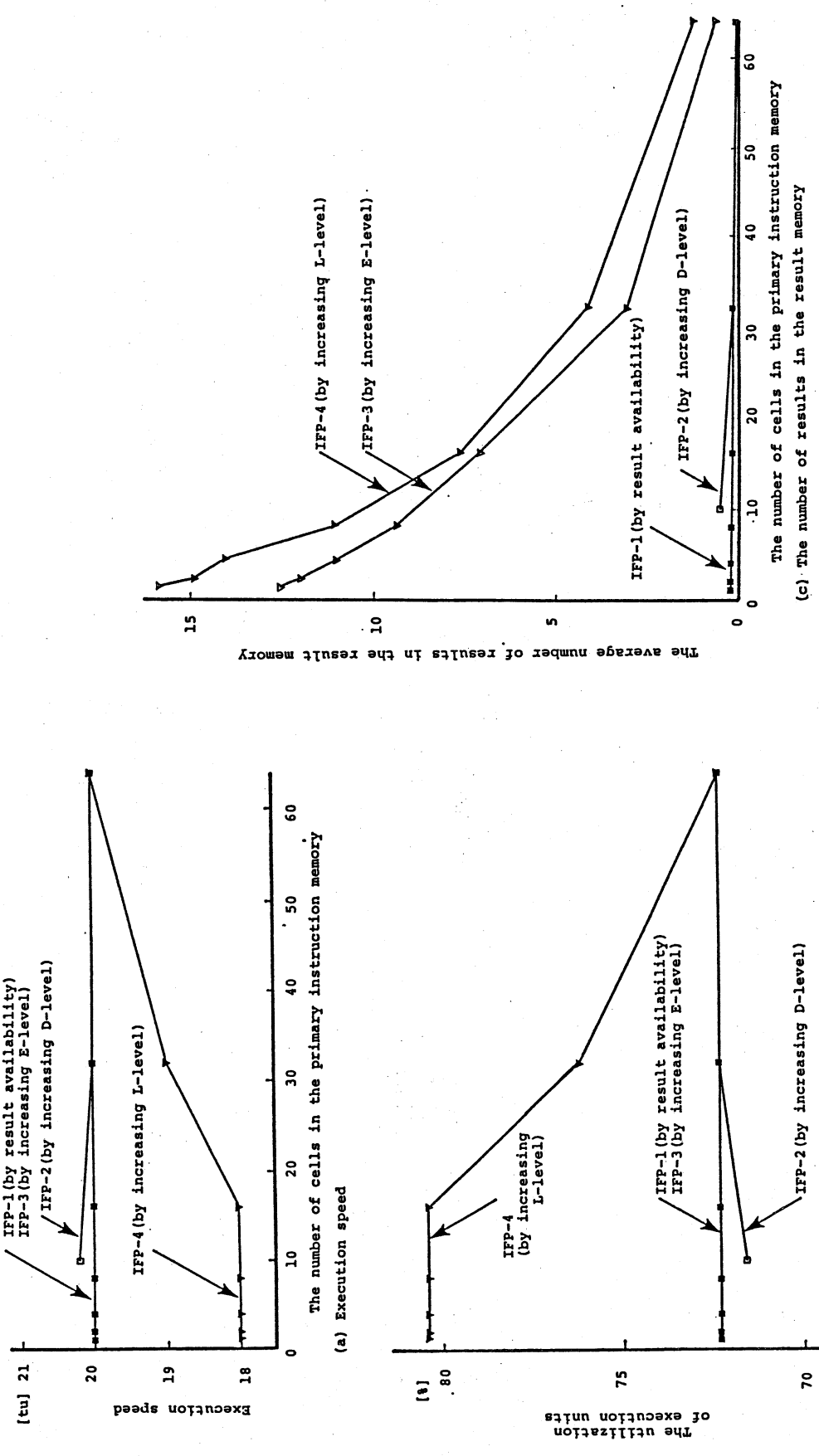
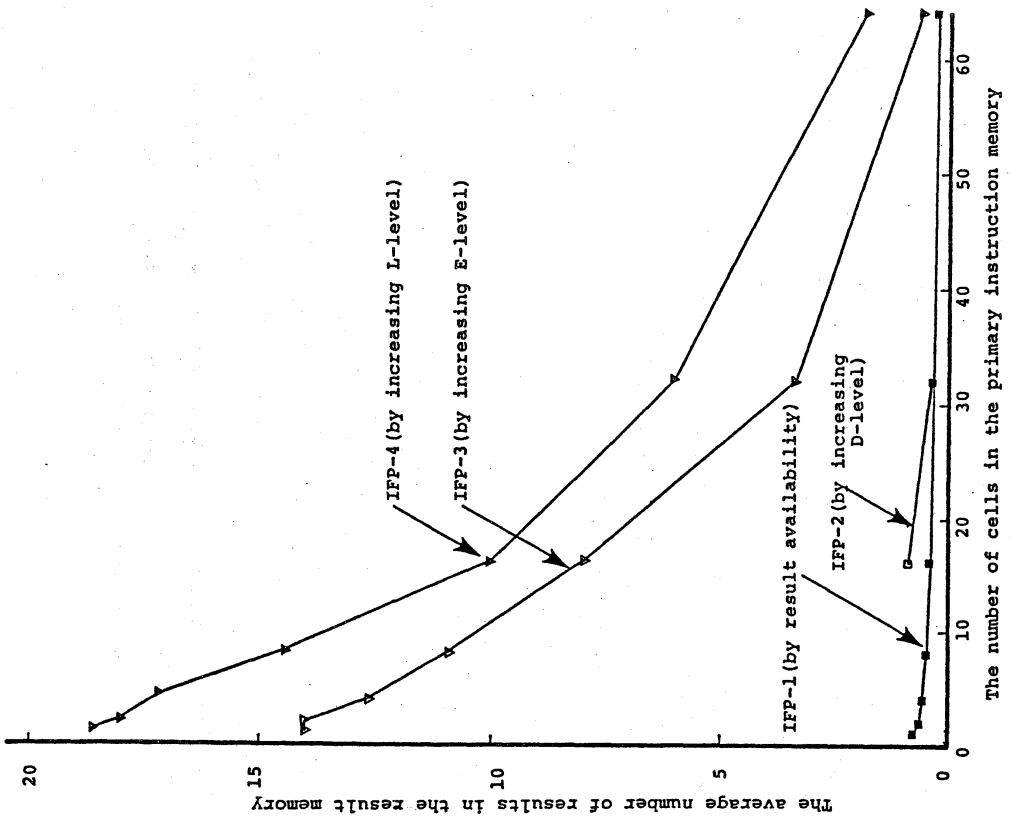


Fig. 3 Simulation result for scheduling effect (case 1)



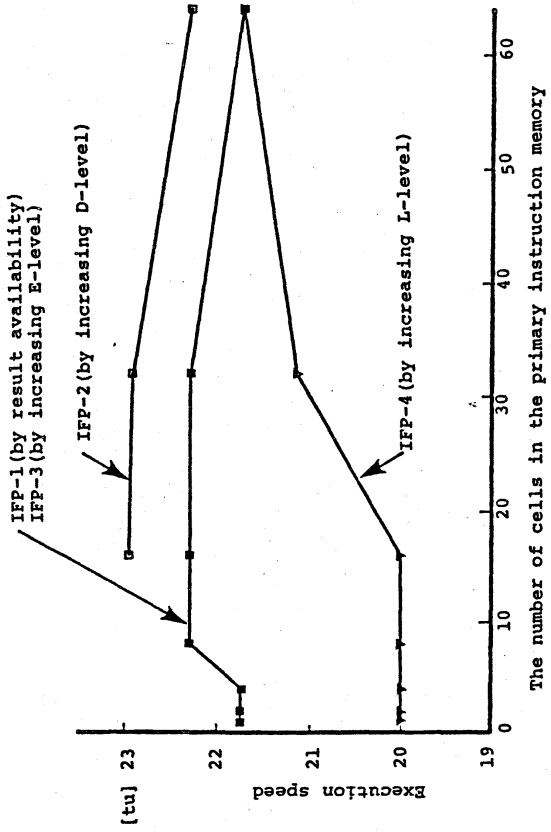


Fig. 4 Simulation result for instruction execution with weight (case 2)

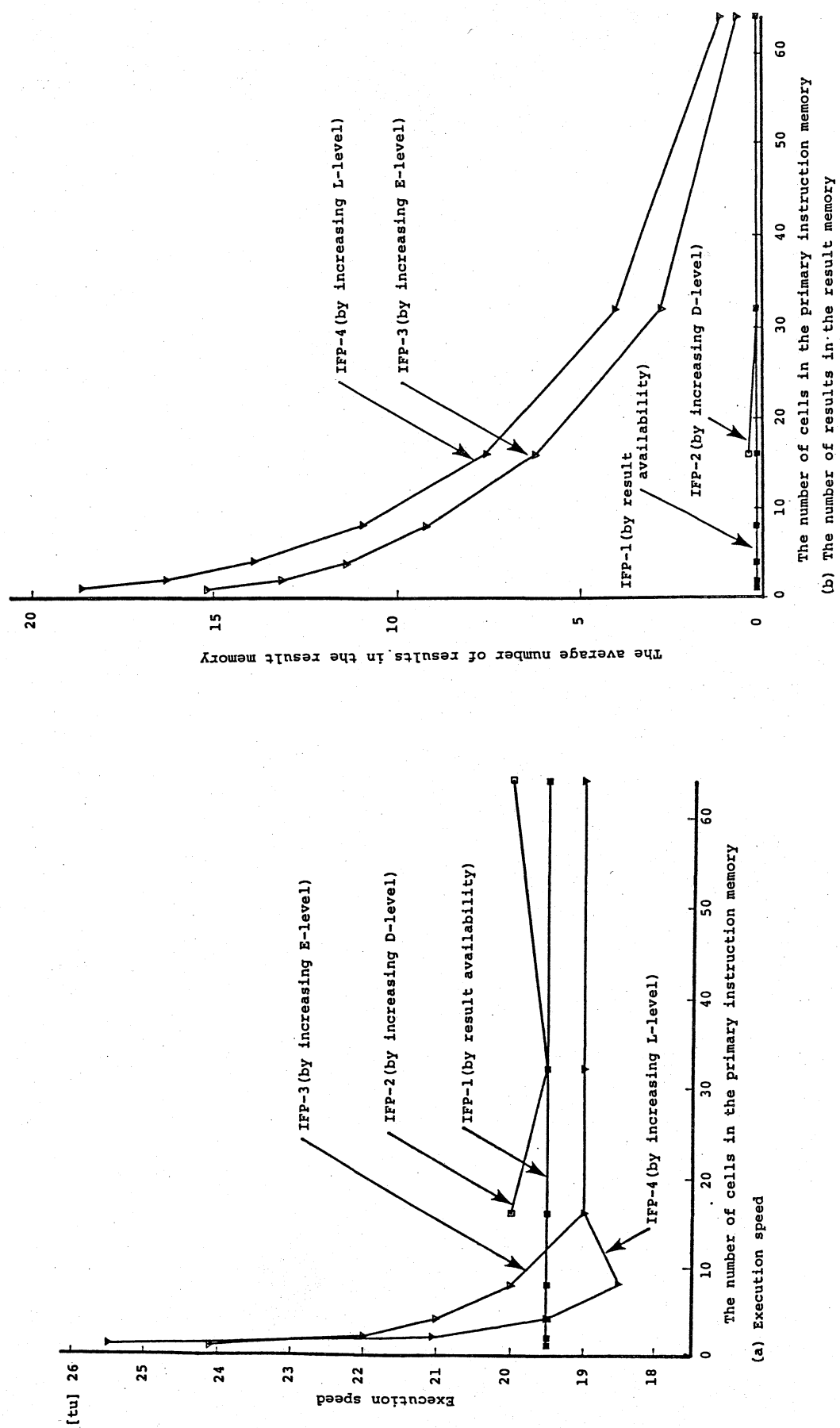


Fig. 5 Simulation result for random execution time (case 3)

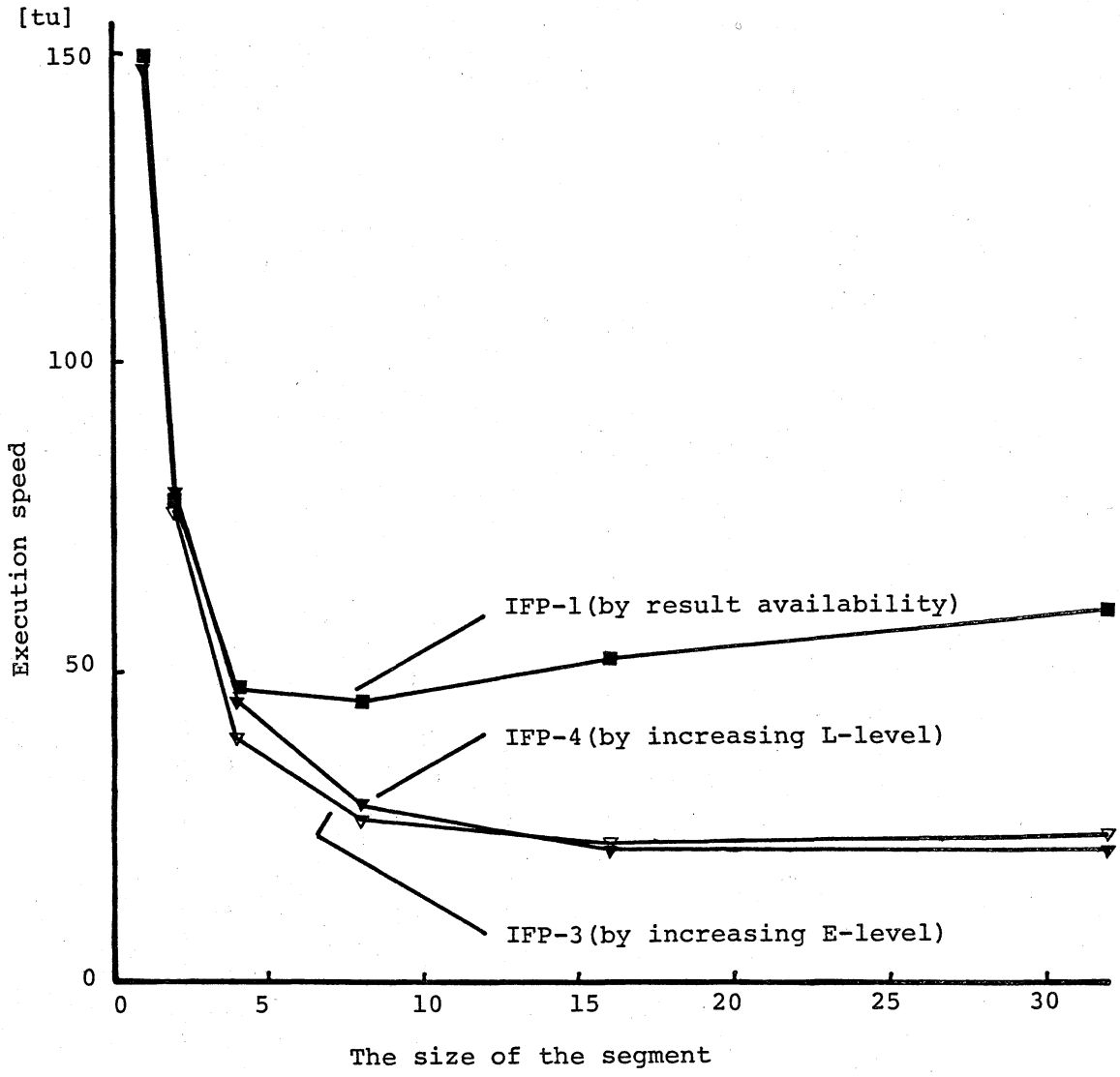
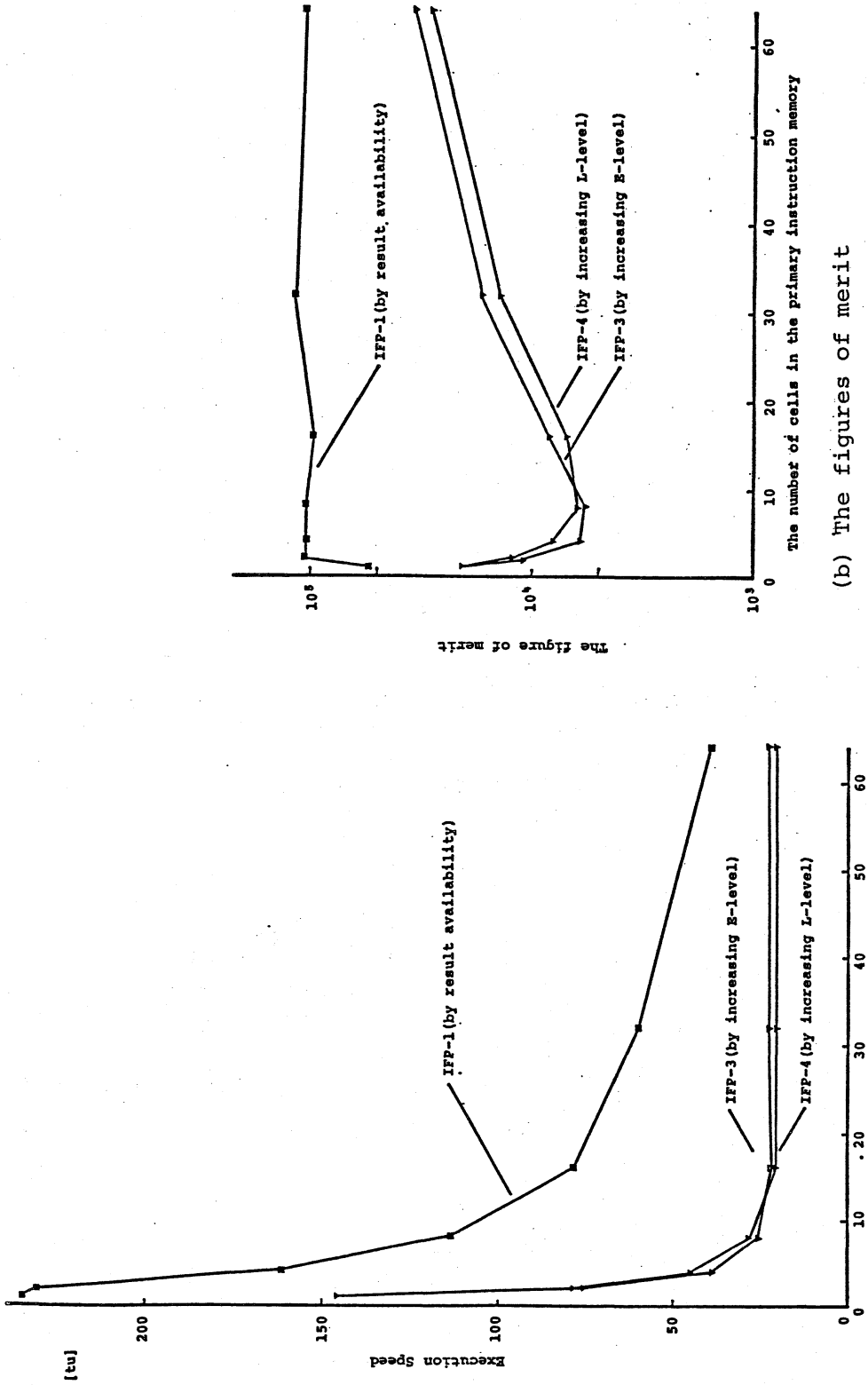


Fig. 6 Simulation result for swapping overhead (case 4)



(a) Execution speed

(b) The figures of merit

Fig. 7 Simulation result for synthetic evaluation for static data-flow programs

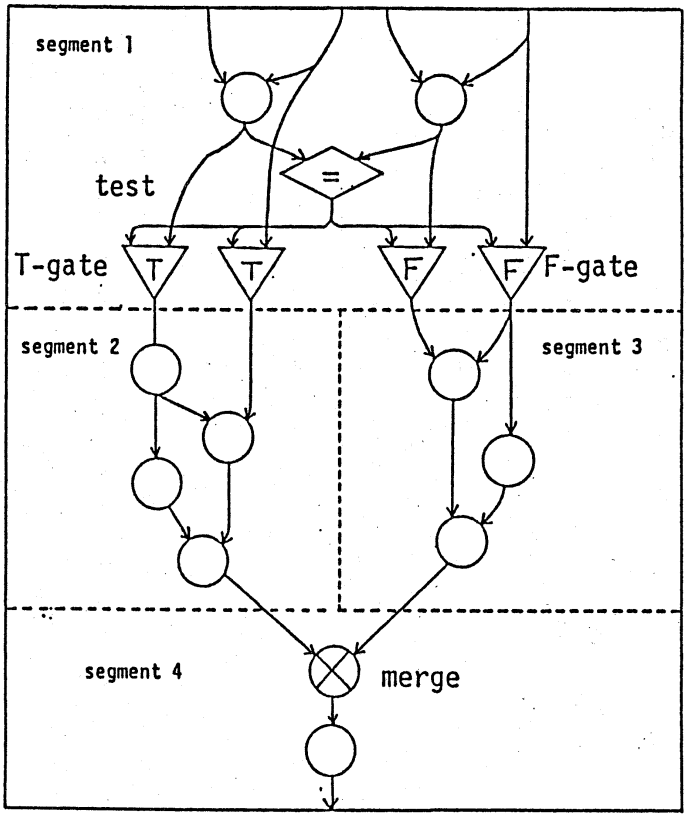


Fig. 8 The alternative segments

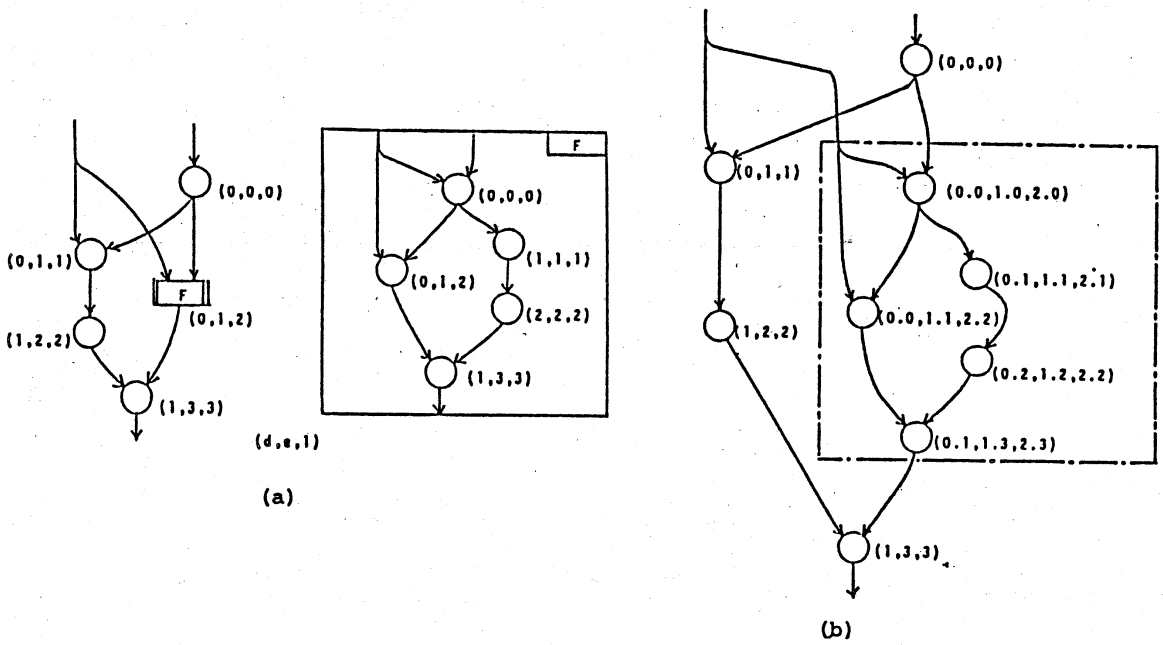
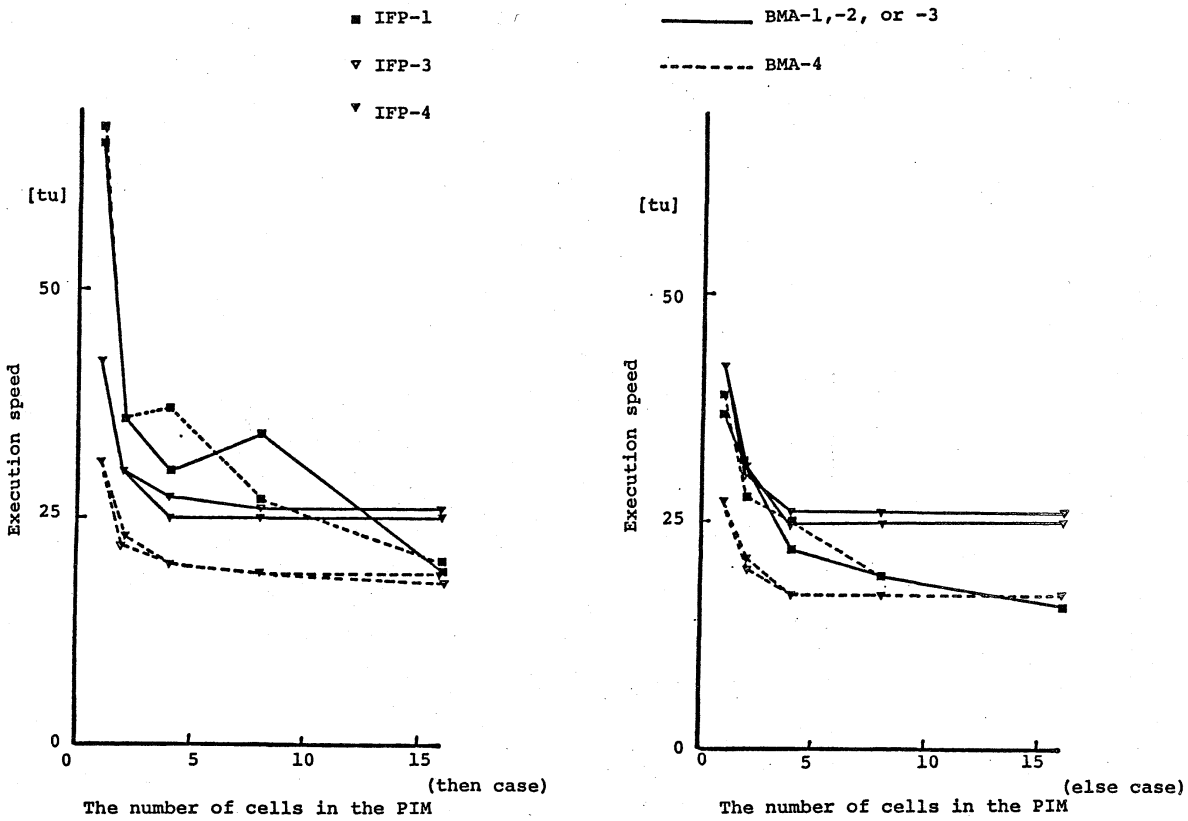
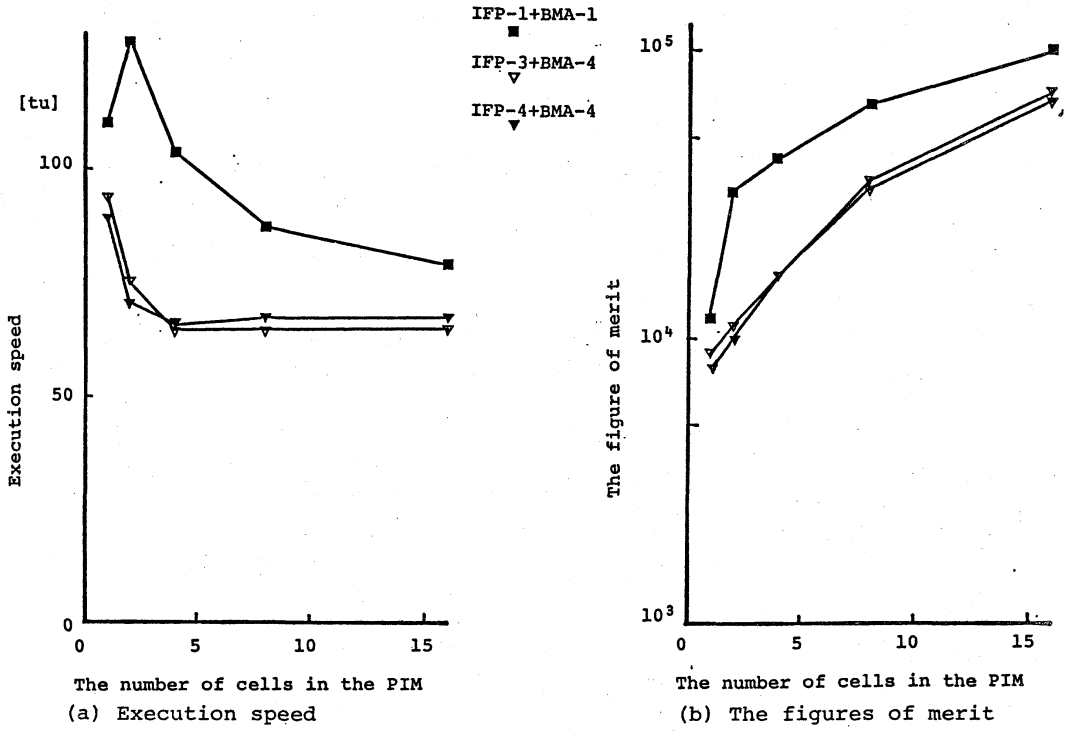


Fig. 9 Hierarchical D-, E-, and L-levels



(a) Execution speed in then case (b) Execution speed in else case

Fig. 10 Simulation result for the branch management algorithms



(a) Execution speed (b) The figures of merit

Fig. 12 Simulation result for the dynamic data-flow program (1fact)

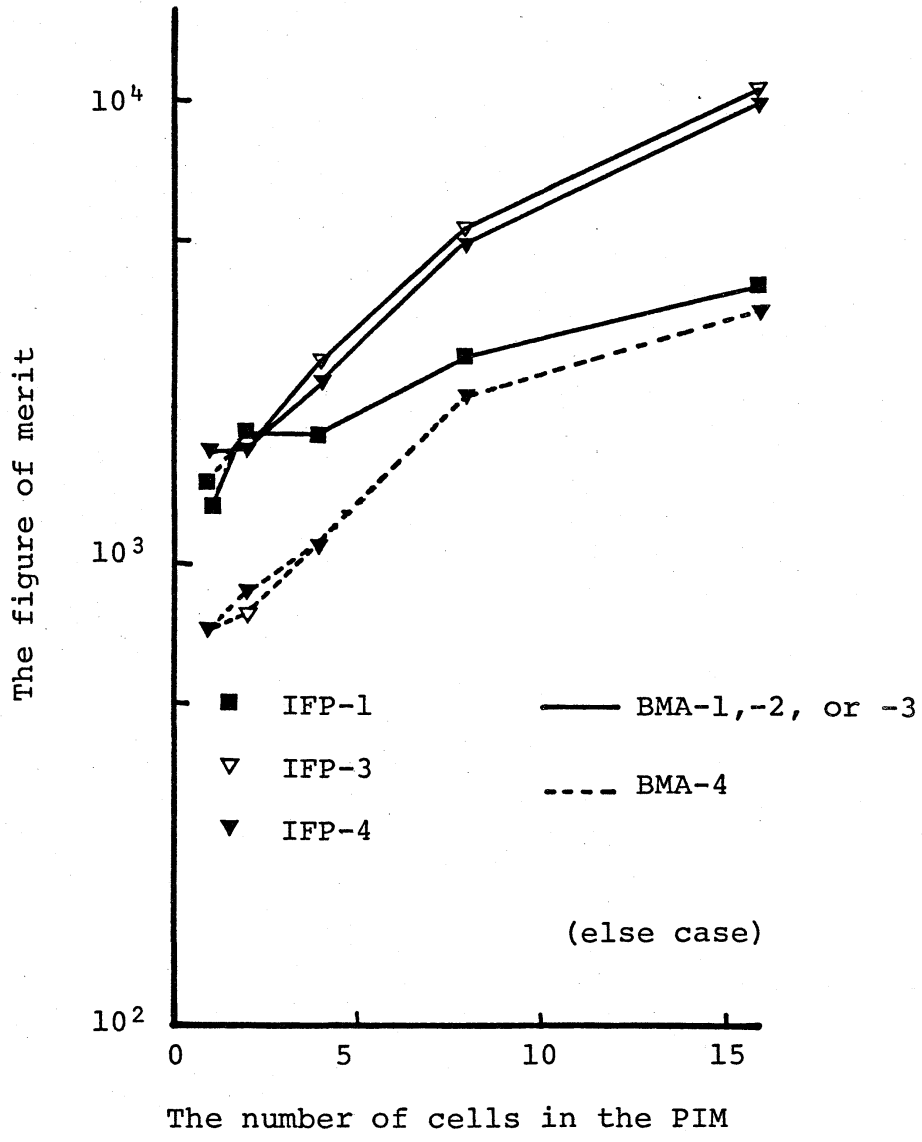


Fig. 11 The figures of merit for the BMA

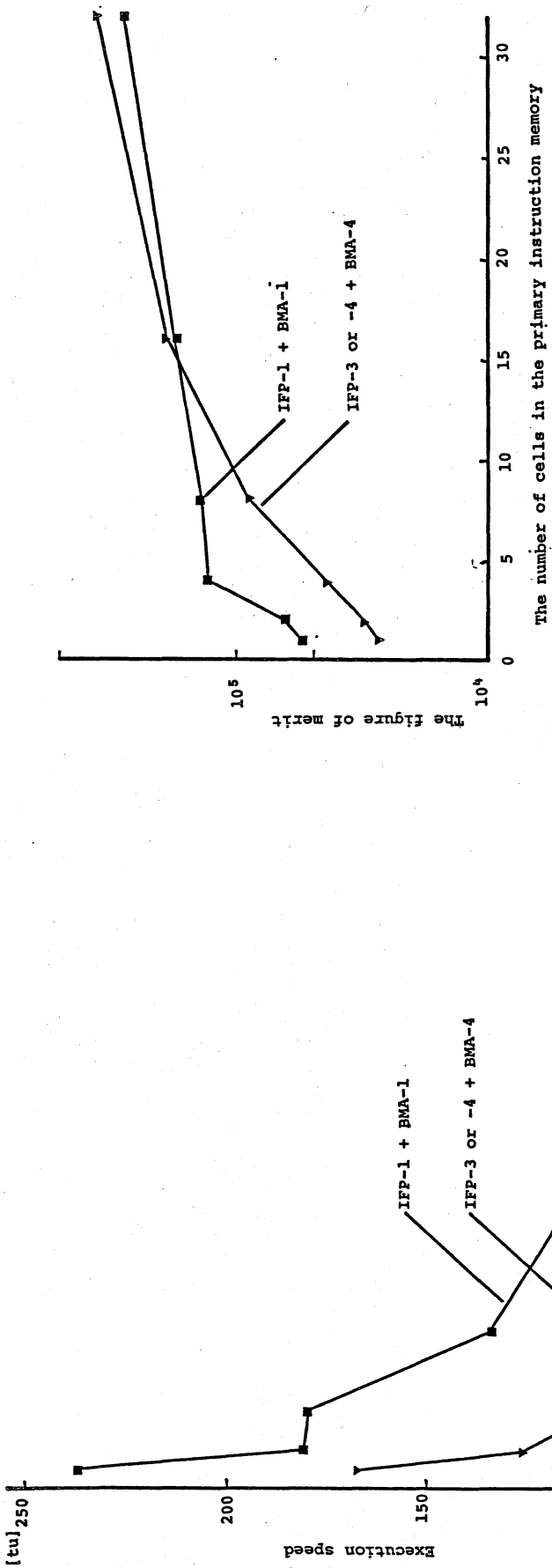


Fig. 13 Simulation result for the dynamic data-flow program (dcfact)