12-1-1987

Robot Control Computation in Microprocessor systems with Multiple Arithmetic Processors

Bo Li
Luoyang Institute of Tracking and Telecommunications Technology, China

Shaheen Ahmad
Purdue University

Robot Control Computation in Microprocessor Systems with Multiple Arithmetic Processors

Bo Li
Shaheen Ahmad

TR-EE 87-47
December 1987

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

ROBOT CONTROL COMPUTATION IN MICROPROCESSOR SYSTEMS WITH MULTIPLE ARITHMETIC PROCESSORS

Bo Li[+], Shaheen Ahmad[++]

[+]Luoyang Institute of Tracking and Telecommunications Technology

Henan, Peoples Republic of China

[++]School of Electrical Engineering

Purdue University

West Lafayette, IN 47907

ABSTRACT

In this paper we address the problem of designing a high performance robot controller with multiple arithmetic processing units (APU's). One attractive feature about this controller is that a minimum number of special purpose hardware components are needed, and in fact off the shelf components can be used. In the controller described in this paper, one main processor (MPU) schedules a number of APU's to produce the computational throughput. In this design an efficient scheduling algorithm plays the most important role in the system performance.

DF/IHS* algorithm [8] is an efficient algorithm that solves "strong" NP-hard problems of scheduling a set of particularly ordered computational tasks onto a

*DF/IHS = Depth First/Initial Heuristic Search, this is a derivative of CP/MISF (critical path/Most Immediate Successor First) scheduling algorithm, see [8].

multiprocessor system. When interprocessor communication overheads are appreciable, it is not very effective in providing a practical near optimum schedule. It fails to consider the problem of contention for shared resources.

In this paper we present new multiprocessor scheduling algorithm, which minimizes the effect of overhead and by doing so it reduces the effect of contention.

We used this scheduling algorithm to derive the operational instructions of the APU's and the MPU for our multiple APU-based robot controller. Simulations show six Motorola MC 68881 APU's can be used to generate the robotic control computations in approximately 2.5 milliseconds. The control computations involve inverse dynamic calculations, forward kinematics, inverse kinematics, and trajectory computations.

1. INTRODUCTION

One of the bottlenecks in the control of industrial robots is that fast computers are necessary and they are not cheaply available. There are many calculations that need to be performed in a control loop time. These calculations include the *trajectory generation,* which is the calculation of the position of the hand in the next sample time, *inverse kinematic* operations to generate the positions of the joints, and the feedback control of the joints. Often linear feedback control is not adequate for trajectory tracking, a feedforward control signal (derived from *inverse dynamic computations)* is then added to the joint drive signal for improved trajectory performance. In many control schemes, the control signal is based on the manipulator hand position in cartesian space (as opposed to using joint positions) in such cases the forward kinematic or Jacobian computations are also needed every sample time. The order of computational demand each of these tasks make on the control computer are as listed:

(1) Inverse dynamics (generation of the feedforward torque signal),

(2) Inverse kinematics,

(3) Forward kinematics,

(4) Trajectory generation,

(5) Feedback control.

The order in the above may change depending on the number of joints and type of computational scheme. Generally the inverse dynamics computation is by far the most intensive, the other four computation are equally important although they require less of the robot controller.

Simpler computation algorithms for inverse dynamics (task one) has received much attention [13] [17] [23] [24]. Likewise the design of special purpose computers for task one has also received much attention. A number of researchers have proposed the developments of special purpose computers with parallel architectures.

Luh and Lin [12] were the first to consider the parallel computation of the inverse dynamics via Newton-Euler[+] [13] techniques. No specific detail of their computer system is given, apart from the fact a branch and bound algorithm is used for scheduling the processors. Niagam and Lee [16] considered the same problem and proposed some nominal architecture for various commercial microprocessors. They assumed the processors can be appropriately interconnected for the particular computation. Kashara and Narita [10] utilized their DF/IHS algorithm [8] to schedule number of microprocessors connected by a common bus to compute the inverse dynamics. Similar problems have been addressed by Watanalee et al. [18] also by Zheng and Hemami [19]. Lathrop [20] has shown systolic architecture and recursive doubling may be used to exploit the parallelism of inverse dynamics, and the computation may be performed in $O(\lceil \log_2 n \rceil)$

+ Newton-Euler computation scheme allows the inverse dynamic computation in O(n) steps given n is the number of robot joints. It is the least computationally intensive scheme in sequential (non parallel) machines.

time steps, where n is the number of robot joints. Nash [21] has designed a processor to perform linear matrix computations; such a processor is useful in kinematics and in many inverse dynamic operations. Several systolic algorithms and pipelined architectures were proposed by Orin, et al. for Jacobian and dynamic inverse computations [22]. Lee and Chang [25] have shown that by using a parallel pipelined single instruction multiple data stream machines, they are able to perform the inverse dynamic computation in $O(K_1 \lceil n/p \rceil + K_2 \lceil \log_2 p \rceil)$ time steps, where p is the number of processors and n is the number of joints of the robot. VLSI implementation of their algorithm was also proposed.

In this paper we also address the problem of robot control computer design. Such a control computer should be able to perform the inverse and forward kinematics, inverse dynamic and trajectory computations in approximately one to five milliseconds. It should be architecturally simple with few components, easy to program and *as new control methodology evolves we should be able to implement them without alteration of the existing hardware system.* Prefferably when we construct such a system we would like to utilize existing off-the-shelf hardware.

Recently in the market a number of 32 bit microprocessor and coprocessors for arithmetic processing (APU's) have appeared. APU's generally are only able to execute arithmetic operations and have a very limited storage eg Motorola's MC 68881 [15]. Such APU's may be loaded by a host with an instruction and their respective operand's. Usually, the APU will interrupt the host once the operation is complete. At that time the host is required to offload the results.

In our proposed robot controller we connect a number of APU's to one host processor through a 32 bit bus (see Figure 1). In this design we select an optimal number of APU's to exploit the task parallelism. One simplicity of this architecture is that on a double size Eurocard board (measuring approximately 9'×8' in^2) it is possible to accommodate a host (eg a Motorola 68000 or 68020 microprocessor) with all necessary peripherals and in excess of ten or more MC 68881 APU's. This system is also upgradable as APU's become significantly faster we may simply exploit this by directly replacing the APU's in our controller.

In this paper we will address the issues involved in generating the instructions to *run concurrently* on the APU's for the basic robotic tasks one through five. We will also address the question of how many APU's do we need and how fast we can carry out this computation for a particular APU (MC 68881).

In order to take full advantage of parallel processing, an efficient scheduling algorithm must be developed to obtain a minimum computation time with a minimum number of processors. Numerous scheduling algorithms* have been developed [3] [4] [8] [12] [13], etc. Among them DF/IHS [8] [9] [10] is one of the most efficient. When data transfer time among tasks are not negligible and other overheads exist, even DF/IHS algorithm becomes inefficient. This is because it assumes that data transfer times are negligible in comparison with the processor computation time. Thus if all the processors are identical, a task can be assigned to any one of the processors without increasing or decreasing the execution time. In fact when the interprocessor communication overheads are considered, a task may have a different execution time in two separate processors. This is why DF/IHS and some branch-and-bound methods become inefficient.

Another drawback of DF/IHS is that it fails to consider the possible contention problem. As is often the case, the contention for shared resources cannot be neglected, it has to be reduced as much as possible in a multiprocessor system, such that resources are efficiently utilized and maximum parallel processing is obtained. Contention problem has been analyzed as a markovian process in [7] [14] etc. However effect of

*We do not cite all reference, but a few relevant to this paper.

contention on a schedule of a set of known tasks has not been extensively analyzed.

In order to obtain maximum throughput from our parallel processing system we developed a new scheduling algorithm DF/MOHS*. The algorithm assumes that: (1) interprocessor communication overheads (including data transfer) and other necessary overheads are not insignificant, (2) contention for host processor (MPU) service exists and has to be considered. As a result (in our scheduling algorithm) not only the relation among tasks, but also the assignment relation between tasks and processors is important and is considered during the scheduling process.

2. THE SCHEDULING ALGORITHM

In order to allocate tasks to processors efficiently, some assumptions are essential. Every APU is assumed to be identical, i.e. they have the same processing capability. The time needed to transfer the same data packet between two processors are also the same, and both data and instructions are transferred between the main processor and coprocessors through a shared bus (see Figure 1). Hence, the execution time of a task* i is T_i and it can be viewed as a computational time t_{ai} and a overhead time t_{oi}. The computational time is the time needed for an APU to compute the task, whereas the overhead time may include the times to fetch the task operational code, task operands, retrieve the results from the APU, and store the results appropriately. Therefore, the overhead time can be further represented as: initiation overhead[1] t_{bi}, data and operand fetch time t_{fi}; task termination overhead[2] t_{ei}, and data storage time t_{si}. The overhead times t_{bi} and t_{fi} are accumulated before a task is executed in a APU and t_{ei} and t_{si} are

*DF/MOHS = Depth First/Minimized Overhead Heuristic Search.
*A mathematical operation executed in an APU.
1: Initiation times may include effective address computation etc.
2: Termination overhead will include such operations as interrupt processing or coprocessor polling.

the overhead times accumulated after a task has been executed in the APU. Four possible situations may arise:

$$t_{1oi} = t_{bi} + t_{ei} \quad \text{(if data transfer is unnecessary)} \tag{1}$$

otherwise,

$$t_{2oi} = t_{bi} + t_{fi} + t_{ei;} \quad \text{(with data fetch only)} \tag{2}$$

$$t_{3oi} = t_{bi} + t_{ei} + t_{si;} \quad \text{(with data storage only)} \tag{3}$$

$$t_{4oi} = t_{bi} + t_{fi} + t_{ei} + t_{si;} \text{ (with full data transfer)} \tag{4}$$

and the total execution time of task i, t_i is then accumulated as:

$$t_i = t_{koi} + t_{ai;} \quad (i = 1,...,n) \quad \text{and} \quad k = 1,2,3,4. \tag{5}$$

If the processing system has one APU, then we may find the overall computation time by adding the prefetch[1] and termination[2] times into the task execution time. In a multi-APU system, if there are more than one APU being serviced by the MPU (Main Processor), then the MPU is required to perform the appropriate prefetch and termination operations, appropriately interleaved with other APU operations so as to minimize the effect of the overhead on the overall execution time. Kasihara and Narita's optimal scheduling algorithm DF/IHS neglected the fact that task initiation and termination may be as large as the actual APU execution time, eg. a fast floating point APU may take approximately 500ns to perform an arithmetic operation, whereas the prefetch and the termination may require more than 100ns each. Kasihara and Narita's scheduling algorithms in this particular case would not select an appropriately efficient solution (as the overhead processing is not addressed). Additionally, the host processor might be requested to service multiple APU's simultaneously and as only one bus exists

[1]Prefetch will now be taken to mean operations related to time t_{bi} and t_{fi}.
[2]Termination will now be taken to mean operations related to t_{ei} and t_{si}. These are loose terms used for easy explanation of the problem.

(restrictions of our problem) to service the APU's, a contention for the MPU service would exist. If the prefetch and the termination operations can be interleaved at instances when the service request for the APU is zero, then the optimal schedule may be obtained by the DF/IHS method. If this is not possible, an additional delay time will be inserted in to the overall computation time. We wish to minimize the effect of this delay time.

Task Representation

Given a set of n computational tasks $T = \{T_1, \ldots, T_n\}$, the relationship between each task may be represented in a finite acyclic task graph G. In general, data transfer only occurs between tasks and their immediate successors. The graph G (see Figure 2) is a multiple weighted as multiple packets of data may be transferred between a particular parent and different children tasks. In G the task i, T_i, is represented as a node, two extra nodes are included in G, one for the beginning of the computation and one for the termination. Both of these nodes have zero processing time, and all nodes can be reached from the entry and exit nodes.

We now describe the scheduling algorithm, it is based around the DF/IHS algorithm except that additional steps are included to minimize the time delay due to overhead operations that cause contention for MPU services. The algorithm is divided into eight steps, each of which are explained in the below.

STEP 1: Determine the level of each task in G. The level l_i of task T_i is defined as the longest path from the exit node to the node of T_i:

$$l_i = \max_k \sum_{j \in \pi_k} (t_{4oj} + t_{aj}) \tag{8}$$

where π_k is the kth path from the exit node to task node T_i. The time $(t_{4oj} + t_{aj})$ is the maximum execution time of task T_j in the worst case without contention. If contention exists, i.e. other APU's requests the service of the MPU this time might increase further. This time is dependent on the selected schedule, and therefore l_i is an approximation.

STEP 2: We next form a list for each task T_i $\{l_i, n_i, t_{4oi}\}$, where n_i is the number of immediate successors. From this list, we form a priority table for each task T_i. Task's with the higher priority are those with larger l_i and n_i, and in that order. That is, if $l_i < l_j$, task j kas higher priority, if $l_i = l_j$, then the one with largest number of children has higher priority. If, $l_i = l_j$, and $n_i = n_j$ then the task having the smallest overhead time t_{4oi} has the higher priority. This is chosen because the smaller overhead implies the MPU may begin servicing other APU's at an earlier time. Here we are making an assumption that t_{4oi} is composed mainly of the task initiation, eg prefetch operations as opposed to termination tasks. If this is not the case, then those tasks with the smallest initiation time should be considered first, before other tasks are scheduled. Note in robotic computations, dyadic and monadic operations are usual with one or two data fetches and one operator code fetch, and one resultant word is output. In robotic computations initiation time is almost equal for all tasks.

STEP 3: At each scheduling step a list of tasks available for immediate execution $\{afe(t)\}$ is next formed. A task is assigned "afe" status, if its parents have been executed:

$$afe(t) = \{T_{r1} ... T_{rn_r}\}$$

where n_r is the number of tasks in afe(t).

STEP 4: If m_a is the number of processors *available for computation* at this scheduling stage, select *as many tasks as possible* from the afe(t) in the sequence of priority (as proposed earlier) to form an execution list {fe(t)}. This forms a branching node, for example: $fe(t) = \{T_{r1}, T_{r2}, \ldots, T_{r\ell}\}$; $\ell \leqq \min(m_a, n_r)$.

STEP 5: Assign tasks in {fe(t)} to the available processors and compute delay time as in the below:

<a> Set an incremental variable t_d denoting time delay caused by contention for MPU services equal to zero.

<b> For every available processor, check if any of the tasks in {fe(t)} are children of the task T_{ej} (a task already executed in the available processors). If task T_{ri} is the child of T_{ej}, assign the processor on which T_{ej} was executed on to the task T_{ri} and remove the task from the fe(t) list and the processor from the m_a list*.

If the finished task in the processor has other children, other than T_{ri} the MPU schedule is organized to store data first so that other tasks may access them. If no other children other than T_{r_i} exists then this is unnecessary.

Time delay caused by the transfer operations can now be accumulated as:

$$t_d = t_d + t_{sjk} + t_{bi} + t_{fio}$$

If the executed task j has a child, task i and it is in the {fe(t)} list and the task i is chosen to execute on the same processor x (see Figure 3). Then time t_{fio} is the time taken to fetch all the data for task i from the parent tasks l, m and n. The time t_{bi} is the initiation time for task i, e.g. this may include the time needed to fetch the instruction code and perform the address computations. The time t_{skj} represents a storage

* The exact method of processor and task assignment is complex and is described in the below, but it may be thought as being simple as that.
Note l is the number of tasks in {fe(t)} list and $l = \min(m_a, n_r)$.

operation for task k not in the {fe(t)} list. Note this time delay t_d is not the delay of the entire computation but it is a delay in this computation step with respect to that of the no overhead case. This delay is used to calculate the real-computation time.

We wish to minimize the data transfer from APU to storage. In order to achieve this we may assign tasks to processors (APU's) from the {fe(t)} list in an appropriate manner. If a task r_1 has a parent in APU P1, then the APU P1 is assigned task r_1. This simplistic algorithm does not produce the optimal results as we illustrate by the below example, assume $\{fe(t)\} = \{r_1, r_3, r_4\ r_5, r_2\}$ and $m_a = \{P1, P2, P3, P4, P5\}$ also the part of the task graph which is currently being executed is shown in Figure 4. If we assign APU's on a priority basis then we may select the following assignment $r_1 \longrightarrow P1$; $r_3 \longrightarrow P2$ therefore we need to transfer data from P2 to P3 to execute r_4. Clearly from the task graph this is nonoptimal.

If a task has m-parents we may reduce the data transfer to (m-1) data fetches by assigning an appropriate APU. This is easily accomplished by the below assignment steps. First we form a table to illustrate the relationship between processors and tasks interms of their parents. For the example in Figure 4, a table may be formed as per below:

		r_1	r_2	r_3	r_4	r_5
	$\Sigma_y(\cdot) \setminus \Sigma_x(\cdot)$	3	0	2	1	0
P1	2	X		X		
P2	3	X		X	X	
P3	1	X				
P4	0					
P5	0					

Table 1

Processor Assignment Map

A cross is placed in the table to indicate if a processor generated a parent, e.g. P1 contains the parents of task r1 and r3. Row $\Sigma_x(\bullet)$ is used to indicate the number of parents of a task generated by the processors in $\{m_a\}$ list e.g. $\Sigma_x(r1) = 3$. Column $\Sigma_y(\cdot)$ indicates the number of children of task M, where task M is generated in a particular processor eg $\Sigma_y(P3) = 1$. Clearly those tasks which have one link to the processor list $(\Sigma_x(\cdot) = 1)$ or one processor link to the task list $(\Sigma_y(\cdot) = 1)$, must be assigned first before assigning other tasks. Since these tasks are those with one parent as the case with r_4 or alternatively the parents have only one child as is the case with r_1 and P3.

Therefore we schedule $r_4 \rightarrow P2$ and $r_1 \rightarrow P3$, such an assignment reduces data transfer operations. Following these assignments, these tasks and processors are removed from the assignment map and the map is changed accordingly. The next set of processor assignments is similar, as $\Sigma_x(r_3) = 1$, we must assign $r_3 \rightarrow P1$. These assignment steps reduce the data transfer. Remaining tasks are arbitrarily assigned to processors on priority basis, as $\Sigma_x(\cdot) = 0$, and $\Sigma_y(\cdot) = 0$, the case with r_5 and r_2.

Consider a second example, with the task graph as shown in Figure 5. The resulting processor assignment map is then as given by Table 2:

		r_2	r_1	r_3	r_4
	$\Sigma_y(\cdot) \setminus \Sigma_x(\cdot)$	3	3	3	1
P1	2	X		X	
P2	3	X	X	X	
P3	2		X	X	
P4	3	X	X		X

Table 2

Processor Assignment Map for Example 2

In this example we select $r_4 \rightarrow P4$, as $\Sigma_x(r_4) = 1$. Next we remove P4 row and r_4 column from the assignment map, and adjust $\Sigma_x(\cdot)$, $\Sigma_y(\cdot)$ values accordingly. We now consider r_2, if the time to transfer data for execution of r_2 in processor P1 or in processor P2 is exactly the same, the task may be chosen to execute in any of the two processors. Otherwise the processor chosen is based on minimum data transfer time. If for example P1 is chosen, then P1 and r_2 must be removed from the assignment map, and $\Sigma_x(\cdot)$, $\Sigma_y(\cdot)$ must be adjusted accordingly. We continue to assign processors in this manner, if at any step the $\Sigma_x(\cdot)$ or $\Sigma_y(\cdot)$ value should become equal to one, the task is assigned the corresponding processor on which one of its parents was generated. The process is repeated down the priority list until all the processors are assigned. These assignment rules can be recursively programmed in a compact manner.

Reflecting on the above algorithm, the processor assignment method will yield optimal results, if all the data transfer operations have approximately the same transfer time. In robot dynamics, kinematics and trajectory computations most of the tasks require the same data transfer time. In such a case, the algorithm will yield optimal results. If however, the operations have varying unit data transfer times the algorithm must be adjusted to select assignments which minimize the time of the transfer, as opposed to minimizing the number of transfers.

<c> In the above we assigned all the tasks that have parents which executed on the processors in the $\{m_a\}$ list. We now assign the remaining tasks with the available processors in their respective priority, until all the tasks in {fe(t)} list are assigned. The time delay t_d is accumulated appropriately for each assignments.

Note that if two tasks in the {fe(t)} list are of the same level then task with the lower overhead may be assigned first.

STEP 6: The time to poll the processors to see if the task has finished may now accumulated, this time is:

$$t_d = \sum_{i \in f^+} t_{ei}$$

where f^+ is the set of tasks which have completed execution.

Accumulation of Time-Delay into Computation Time

The computation time is determined on an incremental basis and it is illustrated in Figure 6. Consider at time t_0, MPU schedules task j in APU1 and task k in APU2 then time delay of t_{d1} and t_{d2} is associated with each of these operations respectively. Assume also task m is in execution in APU X. Then after APU1 is scheduled, time will read $t_0^1 = t_0 + t_{d1}$. After APU2 is scheduled it will read $t_0^2 = t_0 + t_{d1} + t_{d2}$. The next time processor will be required to service an APU will be at $t_0^3 = t_0^2 + \min\{(t_{aj} - t_{d2}), t_{ak}, (t_{am}^* - t_{d1} - t_{d2})\}$. The computation time can be further determined in this stepwise manner.

STEP 7: If the entire task graph is executed, i.e. the exit node is reached, go on to Step 8, otherwise go to Step 3.

STEP 8: At this point one schedule is obtained. If the task graph is complicated or if the graph is large we can stop. Assuming the solution is satisfactory. When the data transfer and contention problems are negligible, this solution is similar to CP/MISF. If interprocessor communication is lengthy and contention problem is appreciable, a better solution than CP/MISF can be obtained by the above steps. This is because the above method of assigning tasks to processors reduces delay due to contention. If the generated solution is not satisfactory or it is not very close to the optimal one backtracking may be employed to search for a better solution. In those cases (in which optimal solution time is unknown) an estimate of the ideal lower bound on the computation time can be used to compare with the present solution and if a better solution is desired, then the present solution can be used as an initial solution. Next, utilizing branch-and-bound method, we can backtrack and search for other possible solutions that have a shorter length. For that purpose, the present solution can be used as an initial upper bound. Thus by continually backtracking from the terminal node to other possible branches closer to the entry node, a desirable computation time may be obtained.

The procedure for backtracking and the determination of new branching nodes is now explained as follows: Assuming there are m coprocessors altogether, at a certain branching stage, if there are n_r ready tasks and m_a idle processors available for execution, and if $n_r \geqq m_a < m$, then the number of branching nodes (possible choice of local schedules) are:

$$n_b = \sum_{k=0}^{m^*} {}^{n_r}C_k = {}^{n_r}C_0 + {}^{n_r}C_1 + \ldots + {}^{n_r}C_{m^*}$$

where $m^* = m_a$ and ${}^{n_r}C_0$ corresponds to selecting (m_a) processors idle, and the $(m - m_a)$ processors would currently be active. If $m_a = m$, the sum is carried over the range $k = 1 \ldots m$, and $m^* = m$. This would correspond to having at least one processor being active. If $m = m_a > n_r$, then we reassign the above summation to be carried over the range, $k = 1 \ldots n_r$ and $m^* = n_r = \min(m_a, n_r)$. If $m > m_a > n_r$ then $k = 0 \ldots n_r$ as this corresponds to having $(m - m_a)$ processors being active.

We seek to eliminate those nodes which will not yield a better solution than the one selected by the present schedule. The way we achieve this is by the following elimination rule. If the selected node will not lead to a solution which will be better than the current solution within an approximation of the lower bound, we delete it from the search list. The lower bound on the computation time of a new node is given in the below:

$$t_{\text{lbound}} = \left[\max(t_{\ell 1},\, t_{\ell 2})\right](1 + e)$$

where,

$$t_{\ell 1} = \max_i \{l_i\} + t_0$$

where $\max\{l_i\}$ is the critical path length from current node to the terminal node in the task graph.

and,
$$t_{\ell 2} = \frac{1}{m} \sum_{i \in U} (t_{ai} + t_{oi}) + t_o$$

where the set of unassigned tasks from the current node is U and $i \in U$. The time t_o is the time taken to arrive at the current branch node from the entry node utilizing the current schedule. The constant e is an arbitrary number and it is chosen as $0 < e < 1$.

After the initial solution is obtained and it is not within t_{lbound} for the graph, we backtrack from the lower level branching node to higher level nodes, using the selection rule and elimination rule to select a new branching node which appears yields a better schedule. Next we branch to STEP 5 to generate another schedule and proceed to check if it is within t_{lbound} of the graph. If it is not, we select another node to backtrack for a solution within the t_{lbound} of the graph. We repeat this process until the entire tree is searched or the desired schedule is obtained.

The scheduling algorithm is represented in the flow charts shown in Figure 7.

A Simple Example of the Scheduling Algorithm

An example task graph is shown in Figure 8. It consists of nine tasks including an entry node and an exit node. Two schedules were generated one by our algorithm (DF/MOHS) and one by DF/IHS for varying number of processors. The results are summerized in Figure 10 for DF/MOHS. It is seen that the computation can be carried out in 17.33 time units for one processor. The results for the DF/IHS algorithm are summerized in Figure 9. The computation time for the DF/IHS algorithm is 19.00 units of time for one processor. This because DF/IHS does not minimize data transfers, eg in steps involving Task 2⟶Task 5, Task 5⟶Task 7 and Task 4⟶Task 6.

Note that overhead transfer time has been added to the task execution time in the DF/IHS algorithm. The effect of contention and data transfer is also clear if we consider the schedules with two processors, DF/MOHS has a processing time of 10.08, whereas DF/IHS has a processing time of 12. With three processors DF/MOHS results in a processing time of 8.08 and for DF/IHS this is 10.5.

On examination of task graph it is seen that at most three parallel execution paths exist for T_1, T_2 and T_3, for remaining part of the graph two such paths exists. Because of such a small parallelism the contention problem is not so significant. However, it does exist and it contributes to the relative increase in the computation time for the

DF/IHS schedule. By minimizing the data transfers in DF/MOHS we have reduced the MPU service request, therefore reduced the effect of contention. This will be further born out by the simulations of the robot control tasks.

Summary of Algorithm Advantages

It is difficult to rigorously prove that our algorithm DF/MOHS yields a near optimal schedule in presence of overhead and contention for MPU services. However, extensive simulation have shown the following:

(i) It can be executed in approximately the same time as DF/IHS and CP/MISF algorithms.

(ii) Our algorithm considers the overhead involved with data transfer and proceed to reduce it by generating a schedule which minimize it.

(iii) By minimizing the data transfer operations we reduce the contention for MPU services, and if contention occurs we accumulate its effect in the overall schedule.

3. The Number of APU's Needed for a Robot Controller

In order to determine the optimal number of APU's needed to perform the forward kinematics, the inverse kinematics, the inverse dynamics and trajectory computations we used the DF/MOHS algorithm to generate the computation time for varying number of APU's. From these set of times we were able to determine the optimal number of APU's and obtain the computation time, these times included overhead and effects of contention. In our calculations we used the data for the Motorola MC68881 APU, the computation times are summarized in Table 3.

Operation	Approximate Computation time (μs)	Approximate Overhead time (μs)			
		t_b	t_f	t_e	t_s
subtraction	4.66	0.24	0.96	0.18	0.48
addition	4.66	0.24	0.96	0.18	0.48
multiplication	5.87	0.24	0.96	0.18	0.48
division	7.78	0.24	0.96	0.18	0.48
sqrt	7.90	0.24	0.48	0.18	0.48
sincos	28.47	0.24	0.48	0.18	0.96
atan2	33.38	0.24	0.48	0.18	0.48
negate	3.59	0.24	0.48	0.18	0.48

Table 3

Computation time for the Motorola MC68881 APU used in our Simulations

Computation Tasks:

For our computation we used the dynamical and kinematic models of the PUMA manipulator. The forward kinematics of the PUMA arm is summarized in the Appendix-A1, the inverse kinematics is summerized in the task graph of Appendix-A2. Seventy four APU operations are necessary to compute the forward kinematics and 104 operations are necessary to compute the inverse kinematic operations. The Newton-Euler inverse dynamic computations of the PUMA arm are summerized in the Appendix-A3, 154 steps or approximately 400 APU computations are necessary to compute the joint feedforward torques given the joint position velocity and accelerations. The cartesian trajectory computation as described in Paul's book in terms of drive matrix are summerized in Appendix-A4. Two hundred and fifty five APU-operations are necessary to compute the drive matrix.

Summary of Simulation Results

The computation times and schedules were generated by DF/IHS and by the DF/MOHS algorithm. In all simulations overhead is included. In order to show the effect of contention, a set of simulations for DF/IHS and DF/MOHS were preformed without accumulating the effect of contention, and another set accumulates the effect due to contention.

The factor e was set to 0.05 for those simulations not accumulating the effect of contention, and e is set to 0.09 for those simulations accumulating the effect of contention. A computer time limit was imposed on the simulations by the UNIX 4.3 Operating System running on VAX 11/780. In those simulations in which contention was accumulated, this time limit was exceeded.

Figure 11 shows the simulation for the forward kinematics. It is seen from the Figure 11 that DF/MOHS without contention produces better results than DF/IHS as expected, as DF/MOHS minimizes data transfer. In the case with contention DF/MOHS also produces better solution than DF/IHS with contention, as minimizing data transfer reduces the effect of contention. Note also that in the DF/IHS simulation with contention a 'kink' is present, this is because in presence of contention it is difficult to get the near optimal solution. No such kink is present in DF/MOHS simulation, mainly because of the reasons indicated earlier i.e. with minimized data transfer, a smaller contention exists, therefore an acceptable solution is found quite quickly.

From Figure 11, it is apparent that approximately six APU's lead to an optimal processing time of $147\mu s$ for the forward kinematics.

Simulations for the inverse kinematic operations are shown in Figure 12. The optimal APU utilization occur for five APU's with a processing time of $416\mu s$.

Simulation results for the inverse dynamics is shown in Figure 13. Here we note that the optimal processing time of 1213 μs occurs for six APU's.

The drive matrix computations can be carried out in approximately 400μs by six APU's.

From our simulations it is seen that with six MC 68881 APU's we may perform the forward kinematic, inverse dynamic, inverse kinematic, drive matrix computations using floating point arithmetic in approximately $\sim$ 2500μs using a very simple parallel processing architecture.

It is interesting to note that different parallelism exist for each of the tasks and it is reflected in the way the computation time change with increasing number of APU's.

CONCLUSION

In this paper we presented an algorithm which extends the method of DF/IHS to include overhead and contention. The algorithm seeks to minimize overhead by reducing the number of data transfer operations between the processors and in this way reduces the effect due to contention for MPU services. This result is verified from simulations of robot control tasks for varying number of APU's.

We have also presented a simple multi-coprocessor (APU) robot controller which may be constructed utilizing the Motorola MC 68881. Such a device has optimal performance with six APU's. Such a controller is able to perform kinematics, inverse dynamics and trajectory computations using floating point arithmetic in approximately 2.5ms. It is sufficiently modular to allow adaptation for other computational purposes. A fundamental component of the design of this robot controller is an accurate schedule which not only produces an accurate estimate of the computation time but also produces an MPU schedule which has a minimum number of operations, allowing easy programming and implementation.

REFERENCES

[1] Ahmad, S. and Li, B. "Optimal Design of Multiple Arithmetic Processor-Based Robot Controllers," Proc. 1987 IEEE Robotics and Automation Conf. Raleigh, N.C.

[2] Ahmad, S., "On the Design of Special-Purpose Computational Structures for Robot Control: Design Constraints," Proc. 1986 Applied Motion Control Conf., Minneapolis, Minnesota.

[3] Arya, S. "An Optimal Instruction-Scheduling Model for a Class of Vector Processors," IEEE Trans. Comp., Vol. C-34, No. 11, Nov. 1985, pp. 981-994.

[4] Coffman, E.G. et al. Computer and Job Shop Scheduling Theory, New York, Wiley, 1976.

[5] Craig, J. J. Introduction to Robotics, Addison-Wesley Publishing Company, 1986.

[6] Fernandez, E. G. and Bussel, B., "Bounds on the Number of Processors and Time for Multiprocessor Optimal Scheduler, IEEE Trans. Comp., Vol. C-22, No. 8, Aug. 1973, pp. 745-751.

[7] Fung, K. T. and Torng, H. C. "On the Analysis of Memory Conflicts and Bus Contentions in a Multiple-Microprocessor System," IEEE Trans. Comp. Vol. C-27, No. 1, Jan. 1979, pp. 28-37.

[8] Kasahara, H., and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," IEEE Trans. Comput., Vol. C-33, 1984, No. 11.

[9] Kasahara, H., and S. Narita, "Load Distribution Among Real-Time Central Computers Connected via Communication Media," Proc. 9th IFAC World Cong., 1984, Oxford: Pergammon.

[10] Kasahara, H. and Narita, S., "Parallel Processing of Robot-Arm Control Computation on a Multimicroprocessor Systems," IEEE Journal of Robotics Automat.,

Vol. RA-1, No. 2, June 1985, pp. 104-113.

[11] Li, C. J. and Wah, B. W., "Computational Efficiency of Parallel Processing Approximate Branch-and-Bound Algorithm," Proc. Int. Conf. Parallel Processing, 1984, pp. 473-480.

[12] Luh, J. Y. S. and Lin, C. S., "Scheduling of Parallel Computation for a Computer-Controlled Mechanical Manipulator," IEEE Trans. Sys., Man and Cyber., Vol. SMC-12, No. 2, March/April 1982, pp. 214-234.

[13] Luh, J. Y. S., Walker, M. W., and Paul, R. P. C., "On-line Manipulator Scheme for Mechanical Robots," Journal of Dynamic Systems, Measurement and Control Trans. ASME, Vol. 102, June 1980, pp. 69-76.

[14] Marsan, M. A., Balbo, G. and Conte, G., "Comparative Performance Analysis of Single Bus Multiprocessor Architectures," IEEE Trans. Comp., Vol. C-31, No. 12, Dec. 1982, pp. 1179-1191.

[15] MC68881 Floating-Point Coprocessor User's Manual, Motorola, Inc., 1985.

[16] Nigam, R. and Lee, C. S. G., "A Multiprocessor-Based Controller for the Control of Mechanical Manipulators," IEEE J. Robotics Automat., Vol. RA-1, No. 4, Dec. 85, pp. 173-182.

[17] Paul, R. P., Robot Manipulator: Mathematics Programming, and Control. MIT Press, 1981.

[18] Watanalee, T., et. al., "Improvement in the Computing Time of Robot Manipulators Using a Multimicroprocessor," Trans. ASME, Vol. 108, Sept. 1986, pp. 190-197.

[19] Zheng, Y. F. and Hemami, H., "Computation of Multibody System Dynamics by a Multiprocessor Scheme," IEEE Trans. Sys. Man, and Cyber., SMC-16, No. 1, Jan./Feb. 1986, pp. 102-110.

[20] Lathrop, R.H., "Parallelism In Manipulator Dynamics," Int. Journal of Robotics Research, MIT Press, Vol. 4, 1985.

[21] Nash, G., "A Systolic/Cellular Computer Architecture for Linear Algebraic Operations," Proceedings of the 1985 IEEE Conference on Robotics and Automation, St. Louis, MO, April 1985.

[22] Orin, D.E., H.H. Chao, Olson, K.W., Schrader, W.W., "Pipeline/Parallel Algorithms for Jacobian and Inverse Dynamic Computations," Proceedings of the IEEE Conference on Robotics and Automation, St. Louis, MO, April 1985.

[23] A.K. Bejczy, "Robot Arm Dynamics and Control," JPL, Pasadena CA, memo 33-669, Feb. 1974.

[24] J.M. Hollerbach, "A Recursive Lagrangian Formulation of Manipulator Dynamics and a Comparative Study of Dynamics Formulation Complexity," IEEE Trans. Systems, Man, Cybernetics, Vol. SMC-10, No. 11, pp. 730-736, Nov. 1980.

[25] Lee, C.S.G., Chang, P.R., "Efficient Parallel Algorithm for Robot Inverse Dynamics Computation," IEEE Trans. Systems, Man Cybernetics, Vol. SMC-16, No. 4, pp. 532-542, July 1986.

APPENDIX A1

TN: Task number
FN: Function number
NC: Number of children
LC: List of children

TN	FN	NC	LC
1	5	12	24, 25, 27, 28, 37, 39, 42, 43, 52, 53, 58, 60
2	5	6	7, 8, 11, 10, 48, 49
3	5	4	7, 8, 10, 11
4	5	6	14, 15, 21, 22, 33, 38
5	5	6	13, 18, 33, 35, 38, 46
6	5	4	13, 15, 18, 21
7	2	1	9
8	2	1	9
9	0	6	19, 30, 35, 45, 51, 54
10	2	1	12
11	2	1	12
12	1	6	17, 31, 34, 46, 50, 55
13	2	2	14, 22
14	2	1	16
15	2	1	16
16	0	2	17, 30
17	2	1	20
18	2	2	19, 31
19	2	1	20
20	1	2	25, 27
21	2	1	23
22	2	1	23
23	1	2	24, 28
24	2	1	26
25	2	1	26
26	0	2	71, 69
27	2	1	29
28	2	1	29
29	1	2	65, 72
30	2	1	32
31	2	1	32
32	0	2	66, 68
33	2	2	34, 45
34	2	1	36
35	2	1	36
36	0	2	37, 42
37	2	1	40
38	2	2	39, 43
39	2	1	40
40	0	1	41
41	7	2	68, 72

TN: Task number
FN: Function number
NC: Number of children
LC: List of children

TN	FN	NC	LC
42	2	1	44
43	2	1	44
44	1	2	66, 71
45	2	1	47
46	2	1	47
47	1	2	65, 69
48	2	1	56
49	2	1	62
50	2	1	56
51	2	1	62
52	2	1	61
53	2	1	59
54	2	1	57
55	2	1	63
56	0	1	57
57	1	2	58, 60
58	2	1	59
59	1	1	74
60	2	1	61
61	0	1	74
62	0	1	63
63	0	1	64
64	7	1	74
65	2	1	67
66	2	1	67
67	1	1	74
68	2	1	70
69	2	1	70
70	1	1	74
71	2	1	73
72	2	1	73
73	1	1	74
74	8	0	

Function Type and Computation Time

FN = Function number
FT = Function type
CT = Computation Time

FN	FT	CT	t_b	t_f	t_e	t_s
0	tfadd	4.66	0.24	0.96	0.18	0.48
1	tfsub	4.66	0.24	0.96	0.18	0.48
2	tfmul	5.87	0.24	0.96	0.18	0.48
3	tfdiv	7.78	0.24	0.96	0.18	0.48
4	tfsqrt	7.90	0.24	0.48	0.18	0.48
5	tfsincos	28.47	0.24	0.48	0.18	0.96
6	tfatan2	33.38	0.24	0.48	0.18	0.48
7	tfneg	3.59	0.24	0.48	0.18	0.48
8	exit	0.0	0.0	0.0	0.0	0.0

Function type key

0 = Floating addition
1 = Floating subtraction
2 = Floating multiplication
3 = Floating division
4 = Floating square root
5 = Floating sine cosine
6 = Floating arc-tangent
7 = Floating negate
8 = Null function

APPENDIX A2

TN: Task number
FN: Function number
NC: Number of children
LC: List of children

TN	FN	NC	LC
1	6	1	9
2	2	1	5
3	2	1	5
4	2	1	6
5	0	1	6
6	1	2	7, 17
7	4	1	8
8	6	1	9
9	1	1	26
10	2	1	16
11	2	1	16
12	2	1	15
13	2	1	15
14	0	1	19
15	0	2	17, 22
16	1	1	18
17	1	1	18
18	0	1	19
19	3	2	20, 24
20	2	1	22
21	6	1	25
22	1	1	23
23	4	1	24
24	6	1	25
25	1	2	27, 42
26	5	12	30, 31, 44, 45, 47, 49, 57, 61, 68, 70, 78, 82
27	5	2	28, 34
28	2	1	29
29	1	2	33, 38
30	2	1	32
31	2	1	32
32	0	2	33, 39
33	2	1	37
34	2	1	35
35	0	2	36, 39
36	2	1	37
37	1	1	41
38	2	1	40
39	2	1	40
40	0	1	41

TN: Task number
FN: Function number
NC: Number of children
LC: List of children

TN	FN	NC	LC
41	6	2	42, 43
42	1	1	104
43	5	9	47, 49, 51, 64, 68, 70, 72, 85, 98
44	2	1	46
45	2	1	46
46	1	1	54
47	2	3	48, 56, 77
48	2	1	52
49	2	3	50, 60, 81
50	2	1	53
51	2	1	52
52	1	1	53
53	1	1	54
54	6	1	55
55	5	10	56, 57, 60, 61, 64, 77, 78, 81, 82, 85
56	2	1	58
57	2	1	58
58	0	2	59, 89
59	2	1	66
60	2	1	62
61	2	1	62
62	1	2	63, 93
63	2	1	67
64	2	2	65, 97
65	2	1	66
66	1	1	67
67	1	4	76, 90, 94, 98
68	2	2	69, 90
69	2	1	73
70	2	2	71, 94
72	2	1	74
73	0	1	75
74	7	1	75
75	1	4	76, 89, 93, 97
76	6	1	104
77	2	1	79
78	2	1	79
79	1	1	80
80	2	1	87
81	2	1	83
82	2	1	83
83	0	1	84

TN: Task number
FN: Function number
NC: Number of children
LC: List of children

TN	FN	NC	LC
84	2	1	88
85	2	1	86
86	2	1	87
87	1	1	88
88	1	1	103
89	2	1	91
90	2	1	91
91	1	1	92
92	2	1	101
93	2	1	95
94	2	1	95
95	1	1	96
96	2	1	101
97	2	1	99
98	2	1	99
99	0	1	100
100	2	1	102
101	0	1	102
102	1	1	103
103	6	1	104
104	8	0	

Function Type and Computation Time

FN = Function number
FT = Function type
CT = Computation Time

FN	FT	CT	t_b	t_f	t_c	t_s
0	tfadd	4.66	0.24	0.96	0.18	0.48
1	tfsub	4.66	0.24	0.96	0.18	0.48
2	tfmul	5.87	0.24	0.96	0.18	0.48
3	tfdiv	7.78	0.24	0.96	0.18	0.48
4	tfsqrt	7.90	0.24	0.48	0.18	0.48
5	tfsincos	28.47	0.24	0.48	0.18	0.96
6	tfatan2	33.38	0.24	0.48	0.18	0.48
7	tfneg	3,59	0.24	0.48	0.18	0.48
8	exit	0.0	0.0	0.0	0.0	

Function type key

0 = Floating addition
1 = Floating subtraction
2 = Floating multiplication
3 = Floating division
4 = Floating square root
5 = Floating sine cosine
6 = Floating arc-tangent
7 = Floating negate
8 = Null function

APPENDIX A3

TN: Task number
FN: Function number
NC: Number of children
LC: List of children

TN	FN	NC	LC
1	10	7	4, 5, 10, 11, 14, 19, 20
2	10	4	6, 12, 16, 21
3	10	2	7, 22
4	5	1	5
5	5	1	8
6	5	1	7
7	10	1	8
8	2	1	9
9	6	2	146, 148
10	6	1	11
11	5	1	13
12	5	1	13
13	2	1	150
14	5	2	15, 17
15	10	7	25, 26, 31, 32, 35, 40, 41
16	5	1	18
17	11	1	18
18	2	4	27, 33, 37, 42
19	5	1	20
20	5	1	23
21	5	1	22
22	10	1	23
23	2	1	24
24	7	2	28, 43
25	8	1	26
26	8	1	29
27	8	1	28
28	3	1	29
29	3	1	30
30	6	2	137, 139
31	9	1	32
32	8	1	34
33	9	1	34
34	3	1	141
35	7	2	36, 38
36	1	7	46, 47, 52, 53, 56, 61, 62
37	7	1	39
38	11	1	39
39	3	4	48, 54, 58, 63
40	8	1	41

TN: Task number
FN: Function number
NC: Number of children
LC: List of children

TN	FN	NC	LC
41	8	1	44
42	8	1	43
43	3	1	44
44	3	1	45
45	7	2	49, 64
46	8	1	47
47	8	1	50
48	8	1	49
49	3	1	50
50	3	1	51
51	6	2	128, 130
52	9	1	53
53	8	1	55
54	9	1	55
55	3	1	132
56	7	2	57, 59
57	1	5	67, 68, 73, 74, 77
58	7	1	60
59	11	1	60
60	3	3	69, 75, 79
61	8	1	62
62	8	1	65
63	8	1	64
64	3	1	65
65	3	1	66
66	7	2	70, 82
67	8	1	68
68	8	1	71
69	8	1	70
70	3	1	71
71	3	1	72
72	6	2	121, 123
73	9	1	74
74	8	1	76
75	9	1	76
76	3	1	124
77	7	2	78, 80
78	1	5	83, 84, 89, 90, 93
79	7	1	81
80	11	1	81
81	3	3	85, 91, 95
82	7	2	86, 98
83	8	1	84

TN: Task number
FN: Function number
NC: Number of children
LC: List of children

TN	FN	NC	LC
84	8	1	87
85	8	1	86
86	3	1	87
87	3	1	88
88	6	2	114, 116
89	9	1	90
90	8	1	92
91	9	1	92
92	3	1	117
93	7	2	94, 96
94	1	4	99, 100, 105, 106
95	7	1	97
96	11	1	97
97	3	2	101, 107
98	7	1, 102	
99	8	1	100
100	8	1	103
101	8	1	102
102	3	1	103
103	3	1	104
104	6	2	109, 110
105	9	1	106
106	8	1	108
107	9	1	108
108	3	1	111
109	12	1	113
110	8	1	111
111	3	2	112, 115
112	1	1	154
113	7	1	114
114	3	1	120
115	7	1	117
116	8	1	118
117	3	1	118
118	3	2	119, 122
119	1	1	154
120	7	1	121
121	3	1	127
122	7	1	124
123	8	1	125
124	3	1	125
125	3	2	126, 129
126	1	154	

TN: Task number
FN: Function number
NC: Number of children
LC: List of children

TN	FN	NC	LC
127	7	2	128, 131
128	3	1	136
129	7	1	132
130	8	1	133
131	8	1	133
132	3	1	134
133	3	1	134
134	3	2	135, 138
135	1	1	154
136	7	2	137, 140
137	3	1	145
138	7	1	141
139	8	1	142
140	8	1	142
141	3	1	143
142	3	1	143
143	3	2	144, 147
144	1	1	154
145	7	2	146, 149
146	3	1	154
147	7	1	150
148	8	1	151
149	8	1	151
150	3	1	152
151	3	1	152
152	3	1	153
153	1	1	154
154	0	0	

Function Type and Computation Time

FN = Function number
FT = Function type
CT = Computation Time

FN	FT	CT	t_b	t_f	t_e	t_s
0	exit	0.0	0.0	0.0	0.0	0.0
1	tfadd	4.66	0.24	0.96	0.18	0.48
2	tf2add	9.32	0.48	1.92	0.36	0.96
3	tf3add	13.98	0.72	2.88	0.54	1.44
4	tfmul	5.87	0.24	0.96	0.18	0.48
5	tf2mul	11.74	0.48	1.92	0.36	0.96
6	tf3mul	17.61	0.72	2.88	0.54	1.44
7	tf4m2a	32.80	1.44	5.76	1.08	2.88
8	tf6m3a	49.20	2.16	8.64	1.62	4.32
9	tf9m6a	80.80	3.6	14.4	2.7	7.2
10	tfload	4.66	0.24	0.96	0.18	0.48
11	tf2mil	16.40	0.72	2.88	0.54	1.44
12	tf3load	13.98	0.72	2.88	0.54	1.44

Function type key

0 = Null function
1 = Floating add
2 = Floating (2x1) vector addition
3 = Floating (3x1) vector addition
4 = Floating multiplication
5 = Floating vector x scalar (one element = 0)
6 = Floating vector x scalar
7 = Floating (3x3) matrix x vector (three elements = 0)
8 = Floating (3x3) matrix x vector (four elements = 0)
9 = Floating (3x3) matrix x vector
11 = Floating 3x3 matrix (four element = 0) x (3x1) vector
12 = Floating (3x1) vector addition

APPENDIX A4

TN: Task number
FN: Function number
NC: Number of children
LC: List of children

TN	FN	NC	LC
1	0	2	4, 15
2	0	2	5, 16
3	0	2	6, 17
4	1	1	7
5	1	1	7
6	1	1	8
7	0	1	8
8	0	1	9
9	3	4	13, 15, 16, 17
10	0	1	12
11	0	1	12
12	0	1	13
13	2	1	14
14	4	3	24, 25, 26
15	2	2	18, 24
16	2	2	19, 25
17	2	2	20, 26
18	1	1	21
19	1	1	21
20	1	1	22
21	0	1	22
22	0	1	23
23	3	3	27, 28, 29
24	1	1	27
25	1	1	28
26	1	1	29
27	2	1	135
28	2	1	148
29	2	1	161
30	0	1	34
31	0	1	34
32	0	1	35
33	1	1	36
34	1	1	36
35	1	1	37
36	0	1	37
37	0	1	38
38	3	1	42
39	0	1	41
40	0	1	41

TN: Task number
FN: Function number
NC: Number of children
LC: List of children

TN	FN	NC	LC
41	0	2	42, 44
42	2	1	43
43	4	3	62, 63, 64
44	1	2	45, 46
45	0	2	47, 51
46	0	1	47
47	2	1	48
48	3	1	49
49	6	3	50, 56, 62
50	0	1	51
51	1	2	53, 55
52	0	1	53
53	2	2	57, 63
54	0	1	55
55	2	2	58, 64
56	1	1	58
57	1	1	59
58	1	1	60
59	0	1	60
60	0	1	61
61	3	3	65, 66, 67
62	1	1	65
63	1	1	66
64	1	1	67
65	2	2	135, 138
66	2	2	148, 151
67	2	2	161, 164
68	0	2	71, 82
69	0	2	72, 83
70	0	2	72, 83
71	1	1	74
72	1	1	74
73	1	1	75
74	0	1	75
75	0	1	76
76	3	4	80, 82, 83, 84
77	0	1	79
78	0	1	79
79	0	1	80
80	2	1	81
81	4	3	91, 92, 93
82	2	2	85, 91
83	2	2	86, 92

TN: Task number
FN: Function number
NC: Number of children
LC: List of children

TN	FN	NC	LC
84	2	2	87, 93
85	1	1	88
86	1	1	88
87	1	1	89
88	0	1	89
89	0	1	90
90	3	3	94, 95, 96
91	1	1	94
92	1	1	95
93	1	1	96
94	2	2	138, 142
95	2	2	151, 155
96	2	2	164, 168
97	0	1	100
98	0	1	101
99	0	1	102
100	1	1	103
101	1	1	103
102	1	1	104
103	0	1	104
104	0	1	105
105	3	1	109
106	0	1	108
107	0	1	108
108	0	2	109, 111
109	2	1	110
110	4	3	129, 130, 131
111	1	2	112, 113
112	0	2	114, 118
113	0	1	114
114	2	1	115
115	3	1	116
116	6	3	117, 123, 129
117	0	1	118
118	1	2	120, 122
119	0	1	120
120	2	2	124, 130
121	0	1	122
122	2	2	125
123	1	1	126
124	1	1	126
125	1	1	127
126	0	1	127
127	0	1	128

TN: Task number
FN: Function number
NC: Number of children
LC: List of children

TN	FN	NC	LC
128	3	3	132, 133, 134
129	1	1	132
130	1	1	133
131	1	1	134
132	2	1	142
133	2	1	155
134	2	1	168
135	0	1	136
136	2	2	137, 140
137	2	1	225
138	0	1	139
139	2	2	140, 146
140	0	1	141
141	2	1	225
142	0	1	143
143	2	2	144, 146
144	2	1	145
145	6	1	225
146	0	1	147
147	2	1	225
148	0	1	149
149	2	2	150, 153
150	2	1	225
151	0	1	152
152	2	2	153. 159
153	0	1	154
154	2	1	225
155	0	1	156
156	2	2	157, 159
157	2	1	158
158	6	1	225
159	0	1	160
160	2	1	225
161	0	1	162
162	2	2	163, 166
163	2	1	225
164	0	1	165
165	2	2	166, 172
166	0	1	167
167	2	1	225
168	0	1	169
169	2	2	170, 172
170	2	1	171
171	6	1	225

TN: Task number
FN: Function number
NC: Number of children
LC: List of children

TN	FN	NC	LC
172	0	1	173
173	2	1	225
174	0	1	175
175	2	2	176, 179
176	2	1	225
177	0	1	178
178	2	2	179, 185
179	0	1	180
180	2	1	225
181	0	1	182
182	2	2	183, 185
183	2	1	184
184	6	1	225
185	0	1	186
186	2	1	225
187	0	1	188
188	2	2	189, 192
189	2	1	225
190	0	1	191
191	2	2	192, 198
192	0	1	193
193	2	1	225
194	0	1	195
195	2	2	196, 198
196	2	1	197
197	6	1	225
198	0	1	199
199	2	1	225
200	0	1	201
201	2	2	202, 205
202	2	1	225
203	0	1	204
204	2	2	205, 211
205	0	1	206
206	2	1	225
207	0	1	208
208	2	2	209, 211
209	2	1	210
210	6	1	225
211	0	1	212
212	2	1	225
213	1	1	223
214	1	2	218, 219

TN: Task number
FN: Function number
NC: Number of children
LC: List of children

TN	FN	NC	LC
215	1	2	220, 222
216	1	1	224
217	0	1	218
218	0	1	225
219	0	1	220
220	0	1	225
221	0	1	222
222	0	1	225
223	0	6	136, 149, 162, 175, 188, 201
224	0	6	143, 156, 169, 182, 195, 208
225	7	0	

Function Type and Computation Time

FN = Function number
FT = Function type
CT = Computation Time

FN	FT	CT	t_b	t_f	t_e	t_s
0	tfadd	4.66	0.24	0.96	0.18	0.48
1	tf,ul	5.87	0.24	0.96	0.18	0.48
2	tfdiv	7.78	0.24	0.96	0.18	0.48
3	tfsqrt	7.90	0.24	0.48	0.18	0.48
4	tfatan	25.6	0.24	0.48	0.18	0.48
5	tfvmul	26.93	1.2	4.8	0.9	2.4
6	tfneg	3.59	0.24	0.48	0.18	0.48
7	exit	0.0	0.0	0.0	0.0	0.0

Function type key

0 = Floating addition
1 = Floating subtraction
2 = Floating multiplication
3 = Floating division
4 = Floating square root
5 = Floating sine cosine
6 = Floating arc-tangent
7 = Floating negate
8 = Null function

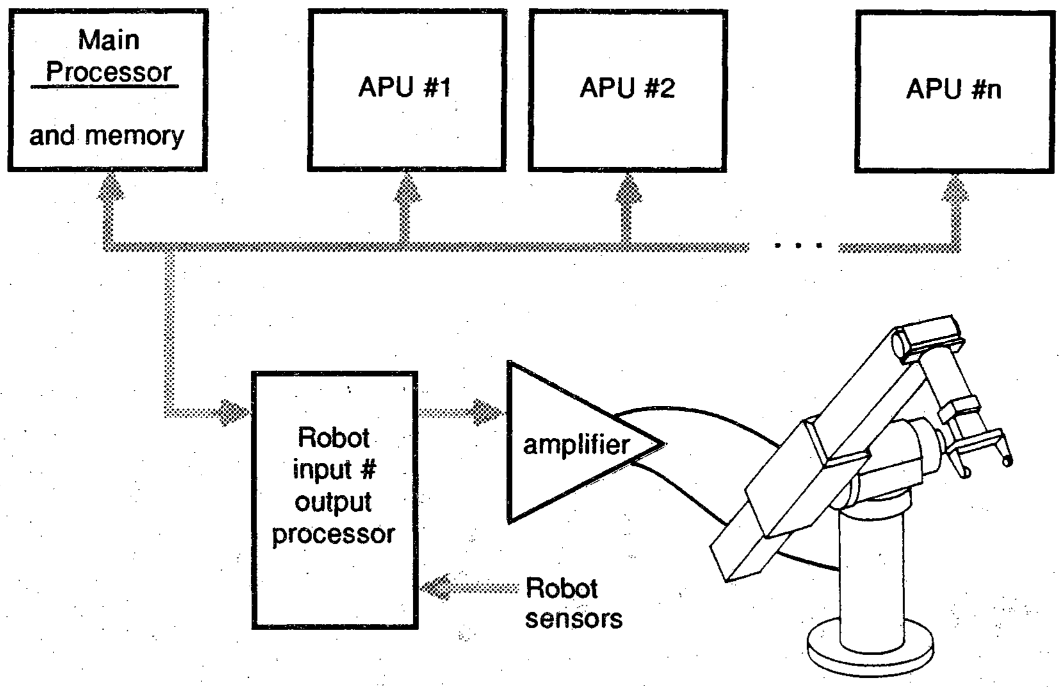


Figure 1

Multiple APU-Based Robot Controller

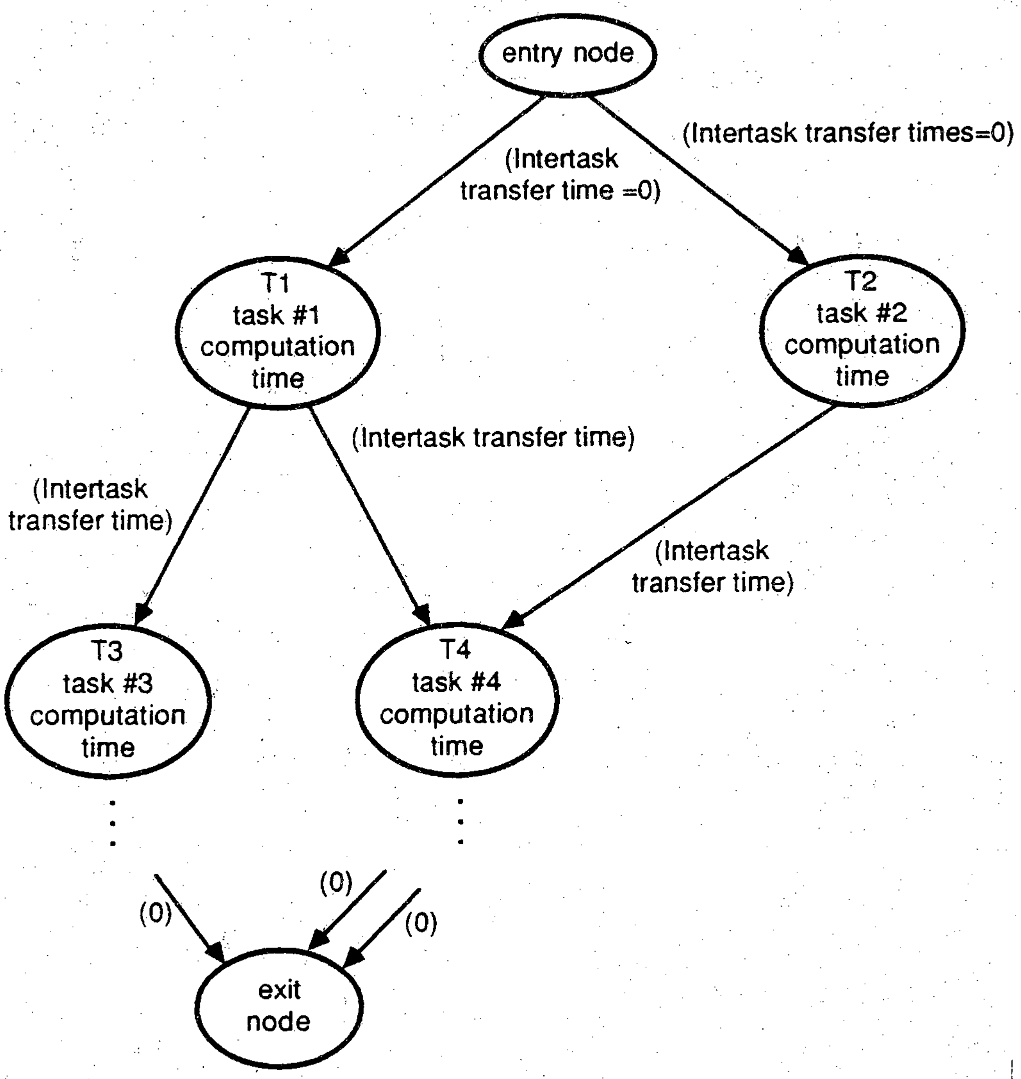

entry node
(Intertask transfer time =0)
(Intertask transfer times=0)
T1 task #1 computation time
T2 task #2 computation time
(Intertask transfer time)
(Intertask transfer time)
(Intertask transfer time)
T3 task #3 computation time
T4 task #4 computation time
(0)
(0)
(0)
exit node

Figure 2

An Acyclic Graph used for Representing a Computation

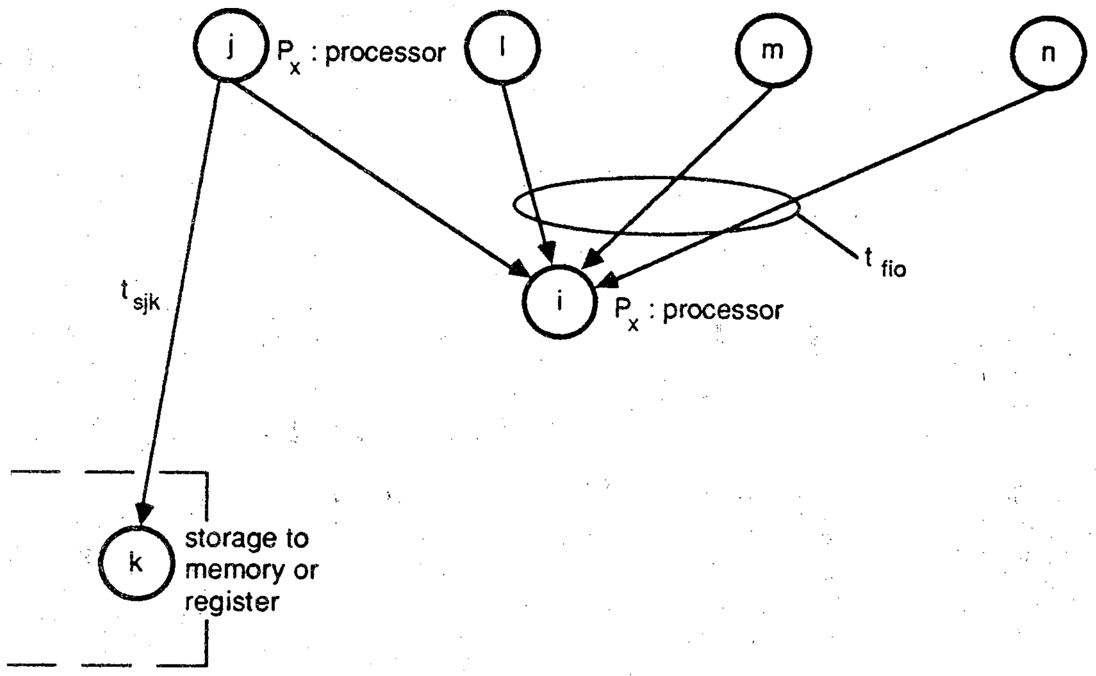


Figure 3

Computing time delay t_d

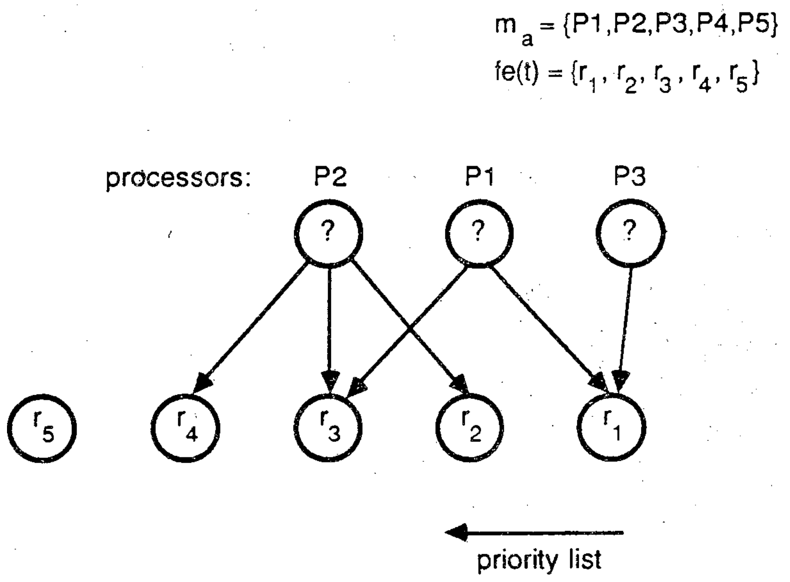

$m_a = \{P1, P2, P3, P4, P5\}$
$fe(t) = \{r_1, r_2, r_3, r_4, r_5\}$
processors:
P2
P1
P3
?
?
?
r_5
r_4
r_3
r_2
r_1
priority list

Figure 4

Example Computation with Processors and Priority Lists

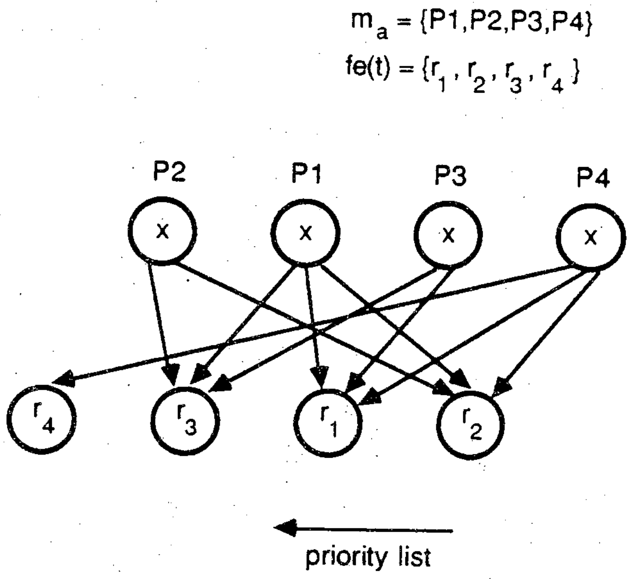
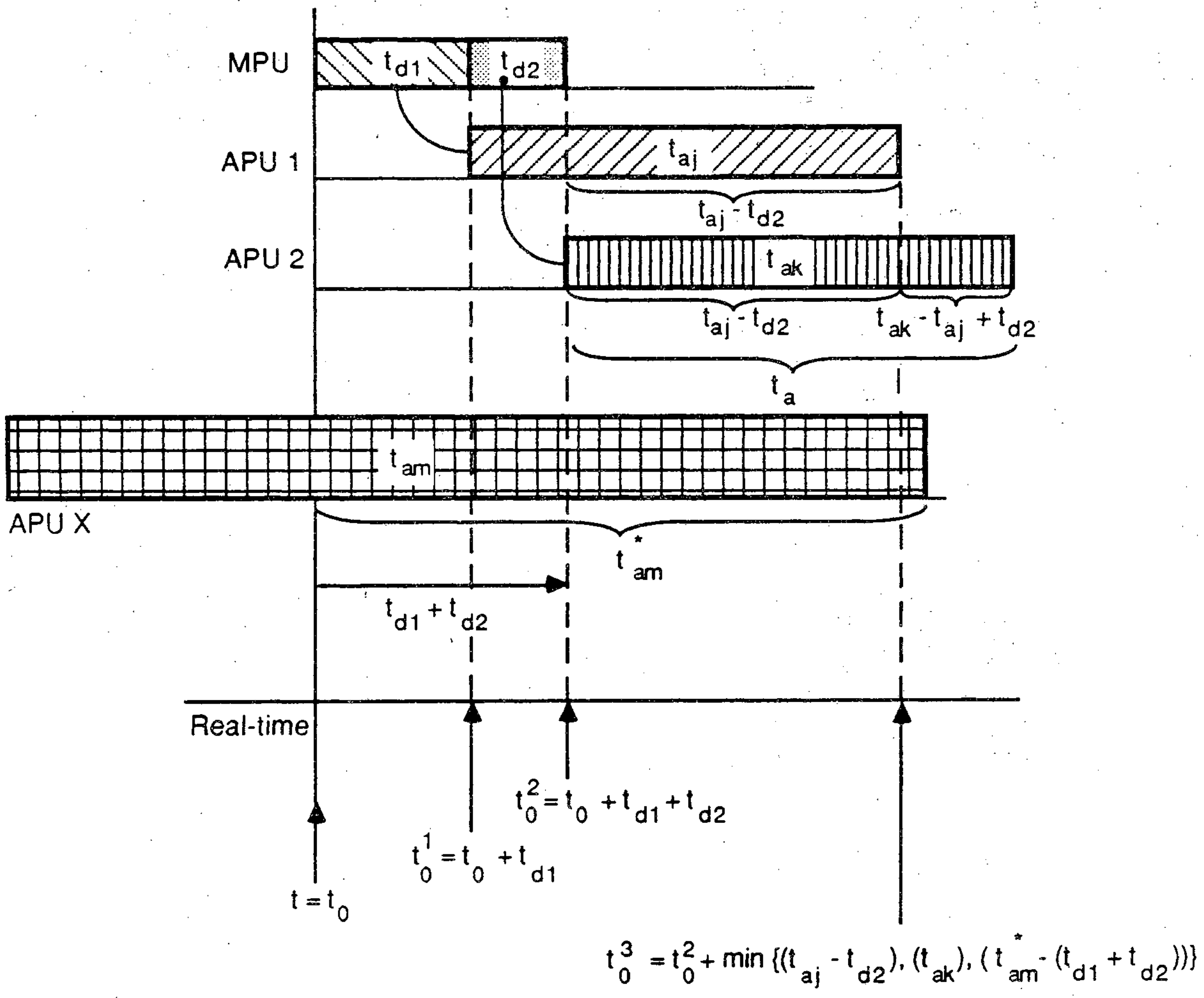


Figure 5

Second Example on Processor Assignment and Task Graph



In this figure: $t^3_0 = t^2_0 + t_{aj} - t_{d2}$

Figure 6

Computation time with accumulation of time-delay

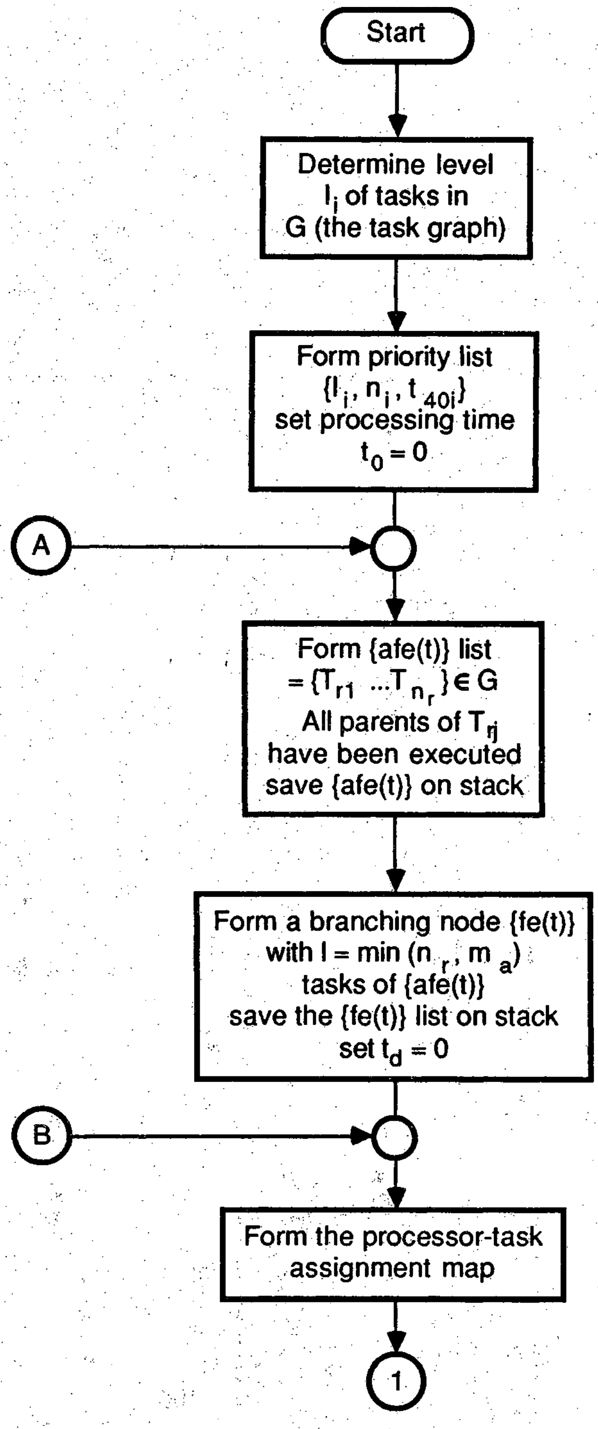

Start
Determine level l_i of tasks in G (the task graph)
Form priority list $\{l_i, n_i, t_{40i}\}$ set processing time $t_0 = 0$
A
Form {afe(t)} list $= \{T_{r1} \ldots T_{n_r}\} \in G$ All parents of T_{rj} have been executed save {afe(t)} on stack
Form a branching node {fe(t)} with $l = \min(n_r, m_a)$ tasks of {afe(t)} save the {fe(t)} list on stack set $t_d = 0$
B
Form the processor-task assignment map
1

Figure 7a
Flowchart of Scheduling Algorithm

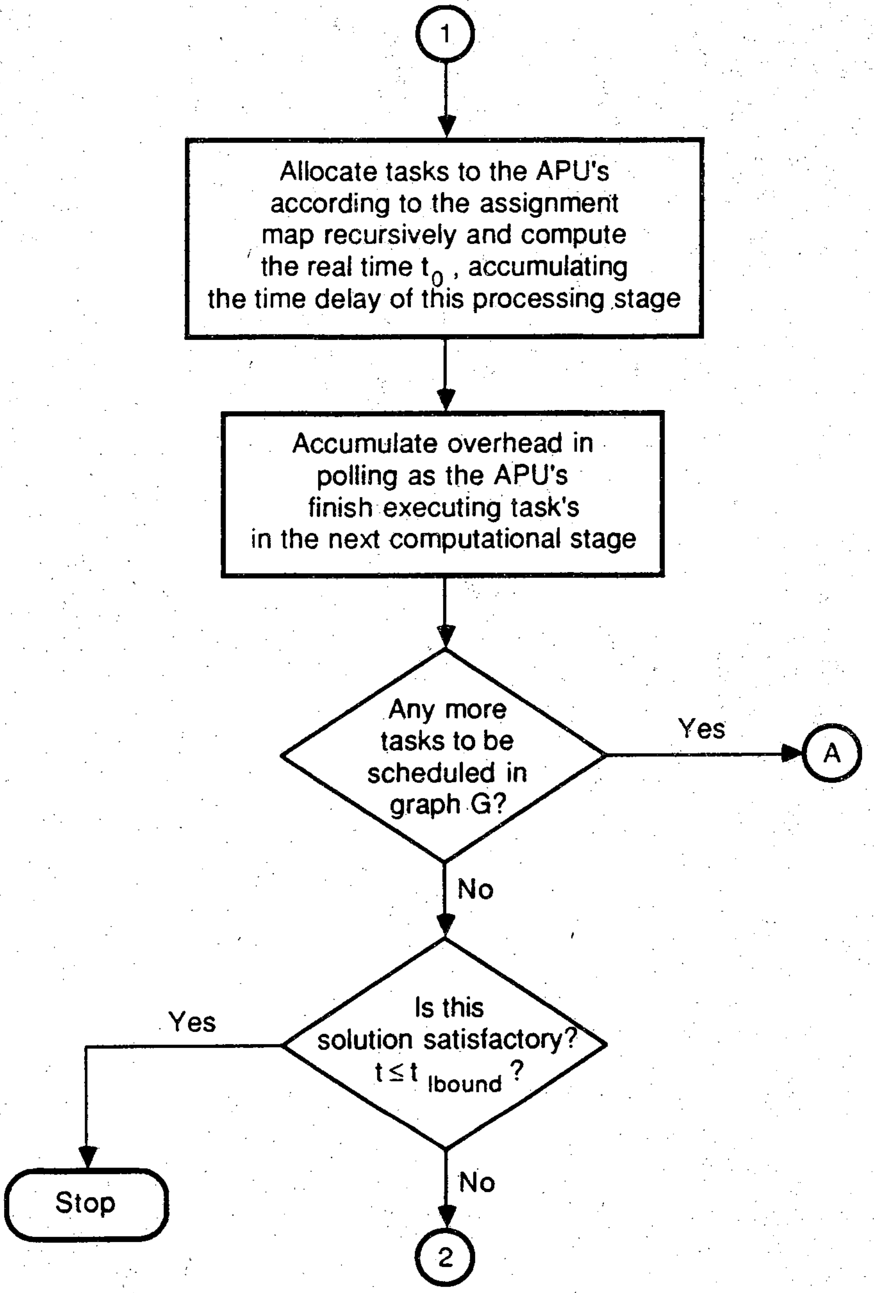

1
Allocate tasks to the APU's according to the assignment map recursively and compute the real time t_0, accumulating the time delay of this processing stage
Accumulate overhead in polling as the APU's finish executing task's in the next computational stage
Any more tasks to be scheduled in graph G?
Yes
A
No
Is this solution satisfactory? $t \leq t_{lbound}$?
Yes
Stop
No
2

Figure 7b
Flowchart of Scheduling Algorithm

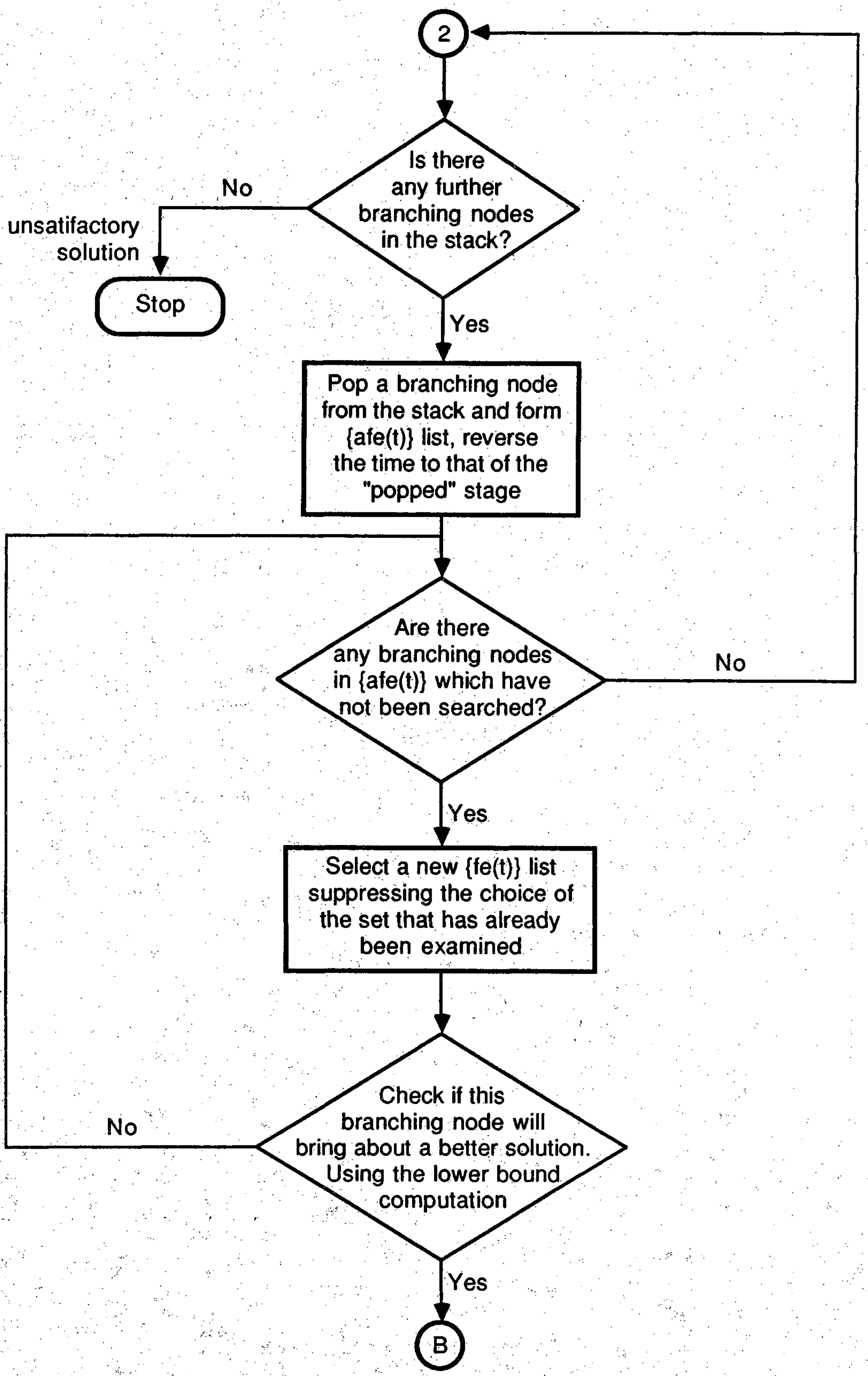
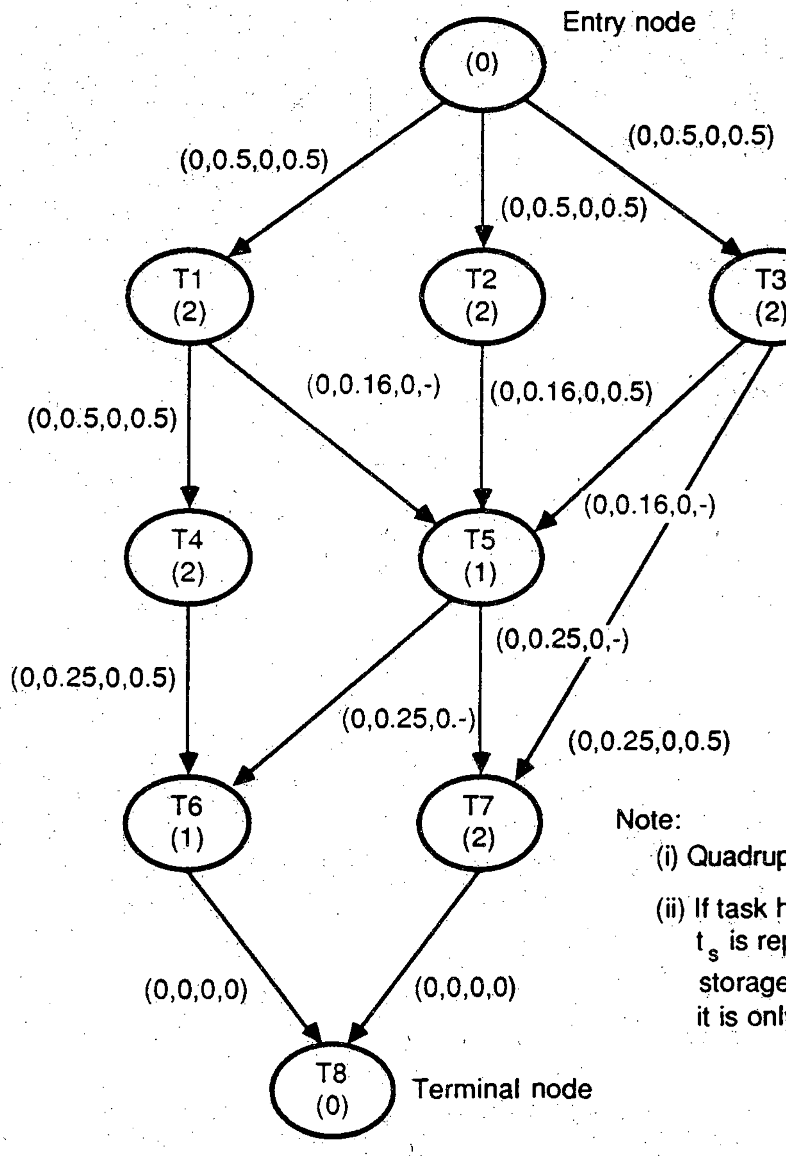

2
Is there any further branching nodes in the stack?
No
unsatifactory solution
Stop
Yes
Pop a branching node from the stack and form {afe(t)} list, reverse the time to that of the "popped" stage
Are there any branching nodes in {afe(t)} which have not been searched?
No
Yes
Select a new {fe(t)} list suppressing the choice of the set that has already been examined
Check if this branching node will bring about a better solution. Using the lower bound computation
No
Yes
B

Figure 7c
Flowchart of Scheduling Algorithm



Note:
(i) Quadruplet weight is (t_b, t_f, t_e, t_s)
(ii) If task has more than one parent, t_s is replaced with (-), meaning only one storage to main memory is necessary, and it is only counted once for all the children.

Figure 8

An Example Schedule with Data Transfer Overhead

Number of processors (APU)=1

Time	m_a	*Task in execution*
0.00	1	1
3.00	1	3
6.00	1	2
9.00	1	5
11.00	1	4
14.00	1	7
17.00	1	6
19.00	1	8

Number of processors (APU)=2

Time	m_a	*Task in execution*	
0.00	2	1	3
3.50	2	2	0
6.50	2	5	4
8.50	1	7	4
10.00	1	7	6
12.00	2	8	0

Number of processors (APU)=3

Time	m_a	*Task in execution*		
0.00	3	1	3	2
4.50	3	5	4	0
6.50	2	7	4	0
8.00	2	7	6	0
10.50	3	8	0	0

Number of processors	*Processing Time*	*Total Idle Time*
1	19.00	0.00
2	12.00	3.00
3	10.50	6.00

Figure 9. Simulation Results for DF/IHS

Number of processors (APU)=1

Time t_o	m_a	*Task in execution*
0.00	1	1
2.50	1	3
5.50	1	2
8.50	1	5
9.83	1	7
12.58	1	4
15.58	1	6
16.83	1	8

Number of processors (APU)=2

Time	m_a	*Task in execution*	
0.00	2	1	3
2.50	1	2	3
3.50	1	2	4
5.50	1	5	4
6.50	1	5	0
6.83	2	7	6
9.08	1	7	0
9.58	2	8	0

Number of processors (APU)=3

Time	m_a	*Task in execution*		
0.00	3	1	3	2
2.50	1	4	3	2
3.00	1	4	0	2
3.50	2	4	0	5
4.83	2	4	0	7
5.58	2	6	0	7
6.83	2	0	0	7
7.58	3	0	0	8

Number of processors	*Processing time*	*Lower bound time*	*APU idle time*
1	17.33	19.00	0.00
2	10.08	9.50	1.33
3	8.08	8.00	6.33

Figure 10. Simulation Results for DF/MOHS

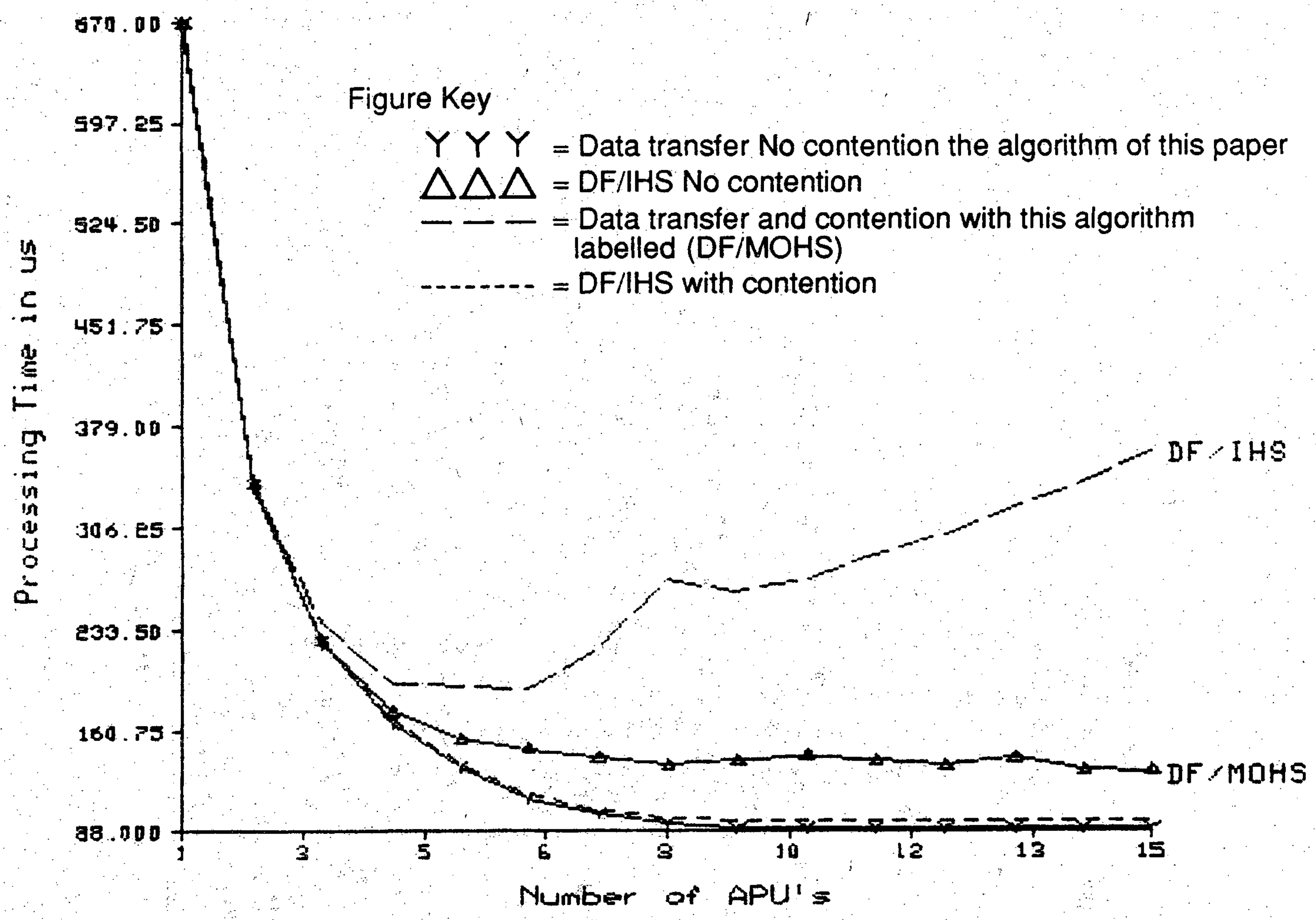


Figure 10

Forward Kinematics Computation Time

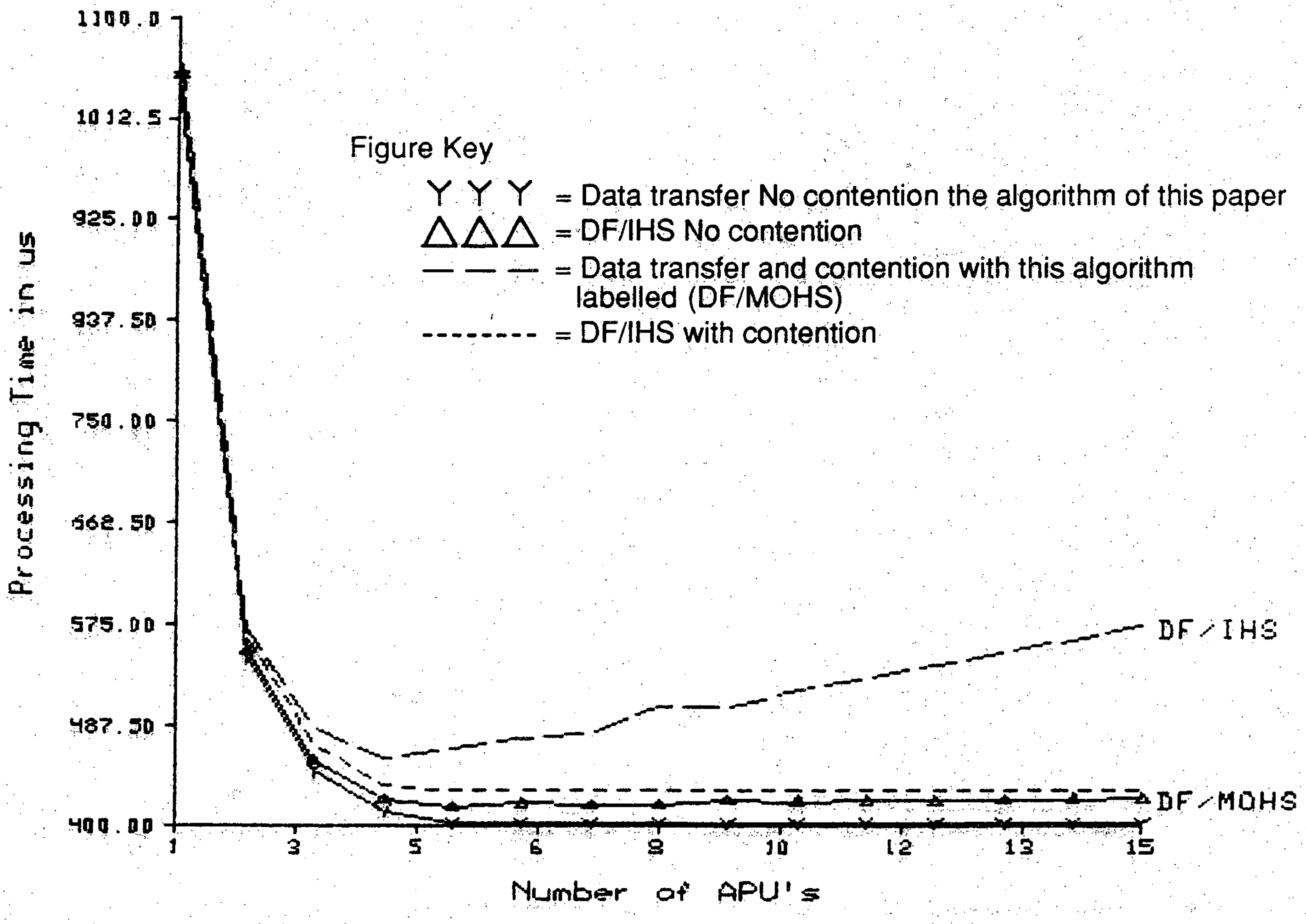


Figure 11

Inverse Kinematics Computation Time

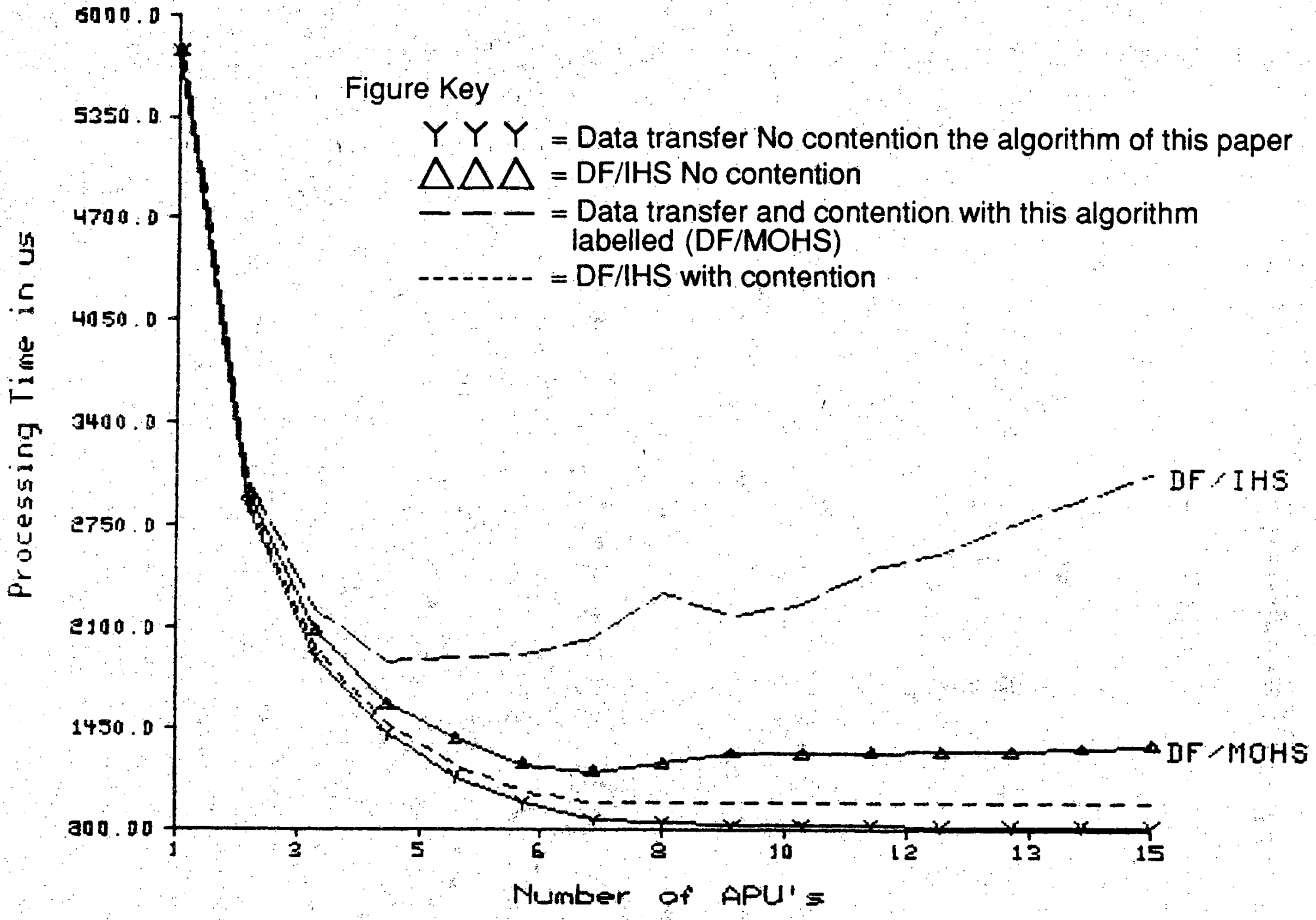


Figure 12

Inverse Dynamics Computation Time for PUMA

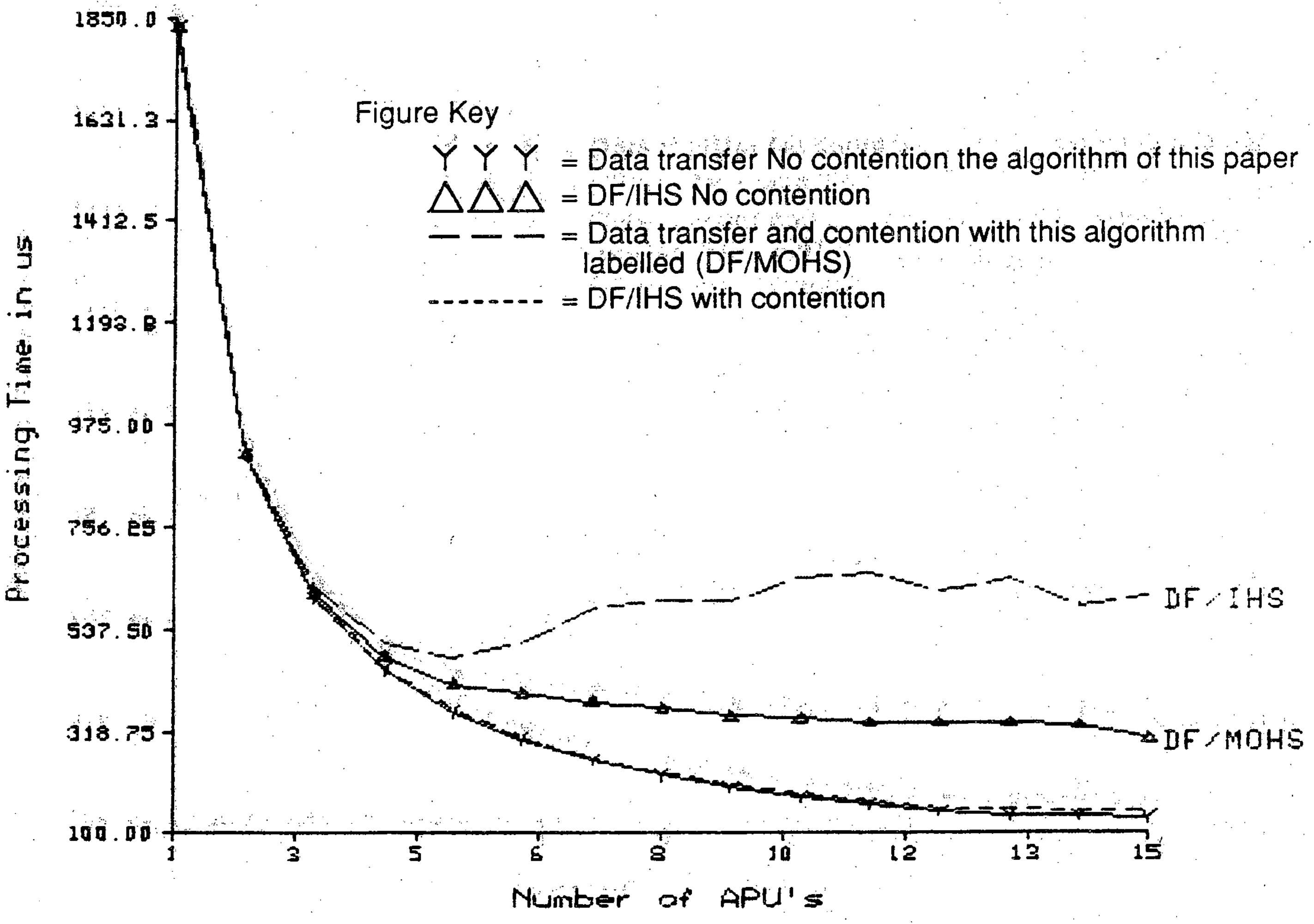


Figure 13

Cartesian Space Path Planning Computation Time