NASA Contractor Report 4534

Node Assignment in Heterogeneous Computing

Sukhamoy Som
CTA INCORPORATED
Hampton, Virginia

Prepared for
Langley Research Center
under Contract NAS1-18936

National Aeronautics and
Space Administration

Office of Management

Scientific and Technical
Information Program

1993

Table of Contents

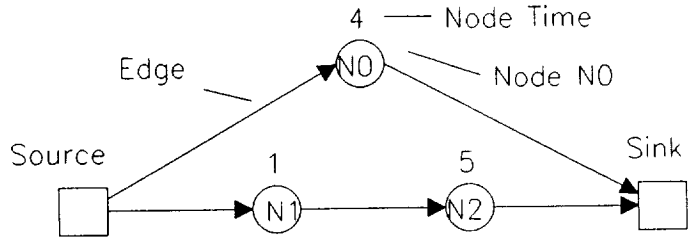
1. Introduction

The Algorithm To Architecture Mapping Model (ATAMM) is a dataflow model for real-time computing in multiprocessor architectures [1, 2]. The model has been successfully implemented in two Very High Speed Integrated Circuit (VHSIC) homogeneous architectures and has been shown to achieve predictable steady-state performance [2]. It is intended that the ATAMM should be extended to include a larger problem domain such as heterogeneous computing. The primary objective of this report is to extend the theoretical foundation of the ATAMM so that the problem domain can be increased. In this section, the present rules of the ATAMM, the purpose of the research, and the organization of the report are described. Some familiarity with the ATAMM is assumed.

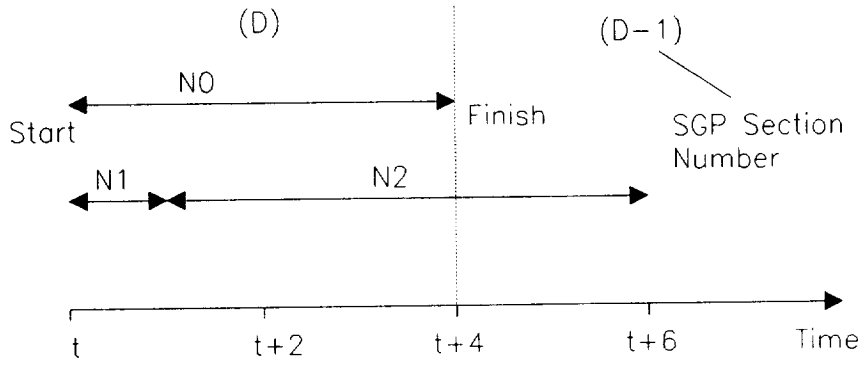
1.1 The ATAMM Rules

A computational problem is represented by a dataflow graph called an algorithm marked graph (AMG) such as that shown in Figure 1(a). Nodes represent computations or operations and directed edges represent precedence constraints or data dependency. A token on an edge indicates the availability of data for the successor node. All edges have a pool of buffers and can accommodate more than one token at a time. Two special nodes are a source and a sink. The algorithm marked graph is executed periodically with data packets injected from the source and the corresponding outputs being removed by the sink. The injected data packets are numbered from 1 and up in the order they are injected. All tokens are tagged with their corresponding data packet number. A node is enabled when all its incoming edges have a token of matching tag. When an enabled node is fired for a data packet, the corresponding token is removed from each incoming edge and a tagged token is deposited on each outgoing edge at the end of node execution. It is possible to execute multiple instances of a node concurrently on different processors [3].

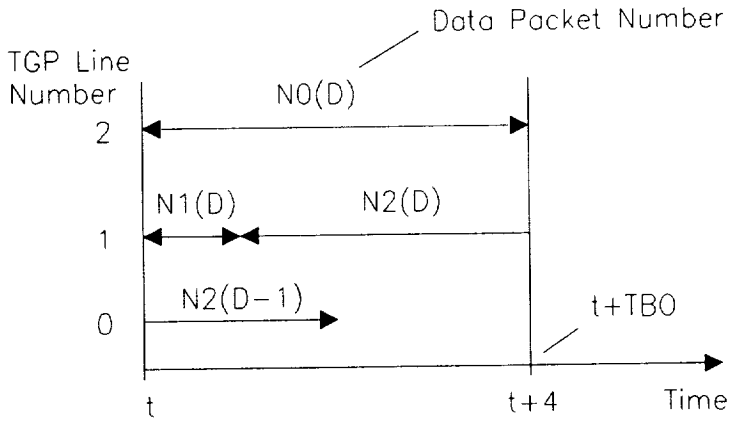
Two other dataflow graphs, called the node marked graph (NMG) and computational marked graph (CMG) are used to describe the execution of an AMG on a



(a)



(b)



(c)

Figure 1. (a) An Algorithm Marked Graph.

(b) Single Graph Play for data packet D.

(c) One iteration period of Total Graph Play.

dataflow architecture. As these graphs are not needed for the contents of this report, the description of the NMG and CMG are omitted.

The execution of a single data packet is known as an iteration and is described by a diagram called the single graph play (SGP)[1]. Total graph play (TGP) is a drawing depicting the beginning, duration, and end of execution of each node of each data packet when the AMG is executed periodically. Two performance measures are defined next. The performance measure TBIO, or time between input and output, is the elapsed computing time between the injection of a data packet and the corresponding output. The time between successive outputs is known as the iteration period and denoted as the time between outputs (TBO). The SGP for data packet D and one iteration period of TGP following the injection of packet D are shown in Figures 1 (b) and (c) respectively. Time t is the injection time for data packet D which is the left vertical line in the TGP diagram. Arrows are used to show the beginning and the end of execution of a node. All node executions are shown on horizontal lines which will be referred to as TGP node execution lines, or simply TGP lines. Execution of source and sink are assumed to take zero time and are not depicted. It is proved in [1] that one iteration period of the TGP can be derived by overlapping sections of the SGP as shown in Figure 1. The SGP for packet D is divided from the beginning in sections of TBO time units, and are numbered from the left with data packets D, D-1, etc. These sections are the contributions from the concurrent data packets towards one iteration period of TGP beginning with the injection of packet D as described in Figure 1.

1.2 Purpose

In the present ATAMM implementations on homogeneous architectures [2], it is assumed that all processors are identical and are capable of executing any node of a given algorithm marked graph. Hence, the nodes are assigned to processors dynamically at run-

[1]The SGP represents steady-state performance. Further analysis to determine the SGP may be required if initial tokens are present in the graph.

time and no effort is made to enforce a fixed or static mapping between nodes and processors. Such node-to-processor mapping where any node can be assigned to any processor in any order is referred to as dynamic assignment in this report. The model determines processor requirements with dynamic assignment and is always able to achieve predictable periodic graph execution. In heterogeneous architectures, however, some nodes may only be executed in a few specific processors. This happens because processors are of different types. In addition to computations, the nodes may represent other operations such as input/output, access to data base, etc. Also, the size of code or data may restrict the execution of certain nodes onto certain processors only. Hence, it may be desired that the set of nodes and processors are partitioned in blocks so that a block of nodes can be mapped onto a distinct block of processors which are capable of executing all nodes from the assigned block.

Another important aspect of a static node to processor assignment scheme is that the memory requirement of a processor is expected to decrease significantly because a processor is not required to execute all nodes of all algorithm marked graphs. Therefore even in homogeneous architectures, there is an advantage in introducing a static node-to-processor assignment scheme in the ATAMM model. Two well known static node-to-processor assignment schemes are fully-static and cyclo-static assignments [4, 5, 6]. In a fully-static scheme, a node of the algorithm marked graph is assigned to a specific processor for all iterations. In a cyclo-static scheme, a node is assigned to a group of processors cyclically. A more detailed definition of static assignments with an extension to suit the ATAMM problem domain, is given in Section 2.2.

The purpose of this report is to develop the necessary foundation to include static assignment schemes for both homogeneous and heterogeneous architectures within the ATAMM context[2]. The dynamic assignment scheme can still be used for homogeneous

[2]Private communication with Paul J. Hayes of NASA Langley Research Center, Hampton, Virginia and Mahyar R. Malekpour of Lockheed Engineering & Sciences Company, Hampton, Virginia.

architectures but a static partitioning of nodes-to-processors is necessary for heterogeneous architectures. Within a partition either dynamic or static assignment scheme could be used[3].

1.3 Organization

In Section 2, the necessary terminology is defined. The algorithms, theorems, and proofs are developed in Section 3. In Section 4, an heterogeneous application is discussed. Conclusions are drawn and future research topics are discussed in Sections 5 and 6, respectively.

2. Terminology

In order to address the issues of node assignment schemes and heterogeneous computing, certain formal terms need to be defined. These terms are derived from Set Theory [7] and various assignment schemes from the literature [4, 5, 6].

2.1 Set Theory Definitions

Set: A set is a collection of distinct objects.

Examples of sets in the ATAMM context are as follows. The processors of a multicomputer form a set, denoted as P. All nodes of an algorithm marked graph form another set, denoted as N.

Hence, N = {N0, N1, N2, ... N(k-1)} and P = {P0, P1, P2, ... P(m-1)}

where the algorithm marked graph has k nodes and the architecture has m processors. N1 and P1 are members or elements of sets N and P, respectively.

Subset: A set S1 is said to be a subset of another set S2 if every element of S1 is also an element of S2.

A set S1 = {N0, N1, N2} is a subset of N and is expressed as S1 $\subseteq$ N. However, a set {N1, N3} is not a subset of {N1, N2}.

---

[3]Private communication with Robert L. Jones, NASA Langley Research Center, Hampton, Virginia, August 1992.

Union of Sets: The union or sum of two sets S1 and S2, designated S1 + S2 or S1 U S2 is the set containing all elements which are members of either S1 or S2 or both.

An example is {N1, N2, N5} U {N2, N6} = {N1, N2, N5, N6}.

Intersection of Sets: The intersection or product of two sets S1 and S2, designated as S1 $\cap$ S2 or S1.S2 is the set containing precisely those elements which are members of both S1 and S2.

An example is {P1, P2, P3} $\cap$ {P2, P3} = {P2}.

Disjoint Sets: Two sets S1 and S2 are disjoint, or mutually exclusive, if they have no common element, i.e., S1 $\cap$ S2 or S1.S2 = ϕ, or a null set.

The set of nodes N and the set of processors P are disjoint. Hence, N $\cap$ P = ϕ, or a null set.

Partition: A partition π on a set S is a collection of disjoint subsets whose set union is S. The disjoint subsets are called the blocks of π. The #(π) is the number of blocks in π.

Suppose a computer has 5 processors named as P0, P1, P2, P3, and P4. Hence, P = {P0, P1, P2, P3, P4}. A possible partition of P is {P0, P1}, {P2}, and {P3, P4}. Then {P2} is a block of this partition and #(π) = 3. However, the partition specified by {P0, P1, P2, P3} and {P3, P4} is not a valid partition of P because the subsets contain the common element P3, and thus, are not disjoint.

2.2 Assignment Definitions

A block of nodes is assigned to a block of processors. This assignment may be **dynamic** where a node may be assigned to any one of the processors in any order. A **static** assignment indicates that the nodes are mapped to the processors according to some fixed rules. A static assignment can be either periodic or aperiodic. A periodic static assignment is a scheme where the node-to-processor assignment is repetitive over a fixed number of iterations. Two periodic static node assignment schemes of interest are cyclo-static and fully-static assignments [4, 5, 6]. They are defined as follows.

Cyclo-Static Assignment: When successive data packets of a node are assigned cyclically within a block of processors with an equal displacement in processor number, the assignment is referred to as a cyclo-static assignment. Let there be m processors numbered as P0, P1, P2, P3, .. P(m-1) to which a node Ni is assigned in a cyclo-static manner. If packet D of the node Ni is assigned to a processor P(q), then packet D+1 of the same node Ni must be assigned to processor P(q+c) modulo m, where c is an integer and is known as the processor displacement. An example with c= 2 and m=5 is given in Figure 2 where the execution of a node Ni of an arbitrary algorithm marked graph is shown for five iteration periods. The data packet D of Ni is assigned to P1 and hence the data packet D+1 must be assigned to P{(1+2) modulo 5} or P3. Similarly, the data packet D+2 is assigned to P{(3+2) modulo 5} or P0.

Fully-Static Assignment: A fully-static assignment is defined as an assignment where all iterations of a node are assigned to the same processor. The successive data packets of a fully-static node is executed by an unique processor. An interesting point to note is that if the processor displacement of a cyclo-static assignment is zero, it results into a fully-static assignment. For example, if c = 0 in Figure 2, the node Ni would have been assigned to processor P1 for all six iteration periods. Hence, fully-static is a special case of cyclo-static assignment with processor displacement equal to zero.

If the assignment of a node is fully-static on a processor, the required code and data could be stored only in that processor. Therefore, a fully-static assignment is expected to require less memory in processors compared to a cyclo-static assignment and is a key objective in [4, 5]. On the other hand, a cyclo-static assignment is more flexible compared to a fully-static assignment and is expected to provide better processor utilization. An added advantage is the existence of multiple copies of code and data which provide the needed redundancy for fault tolerance.

Next, a unique characteristic of the ATAMM problem domain is discussed where a node may be executed concurrently for multiple data packets. In [4, 5], multiple

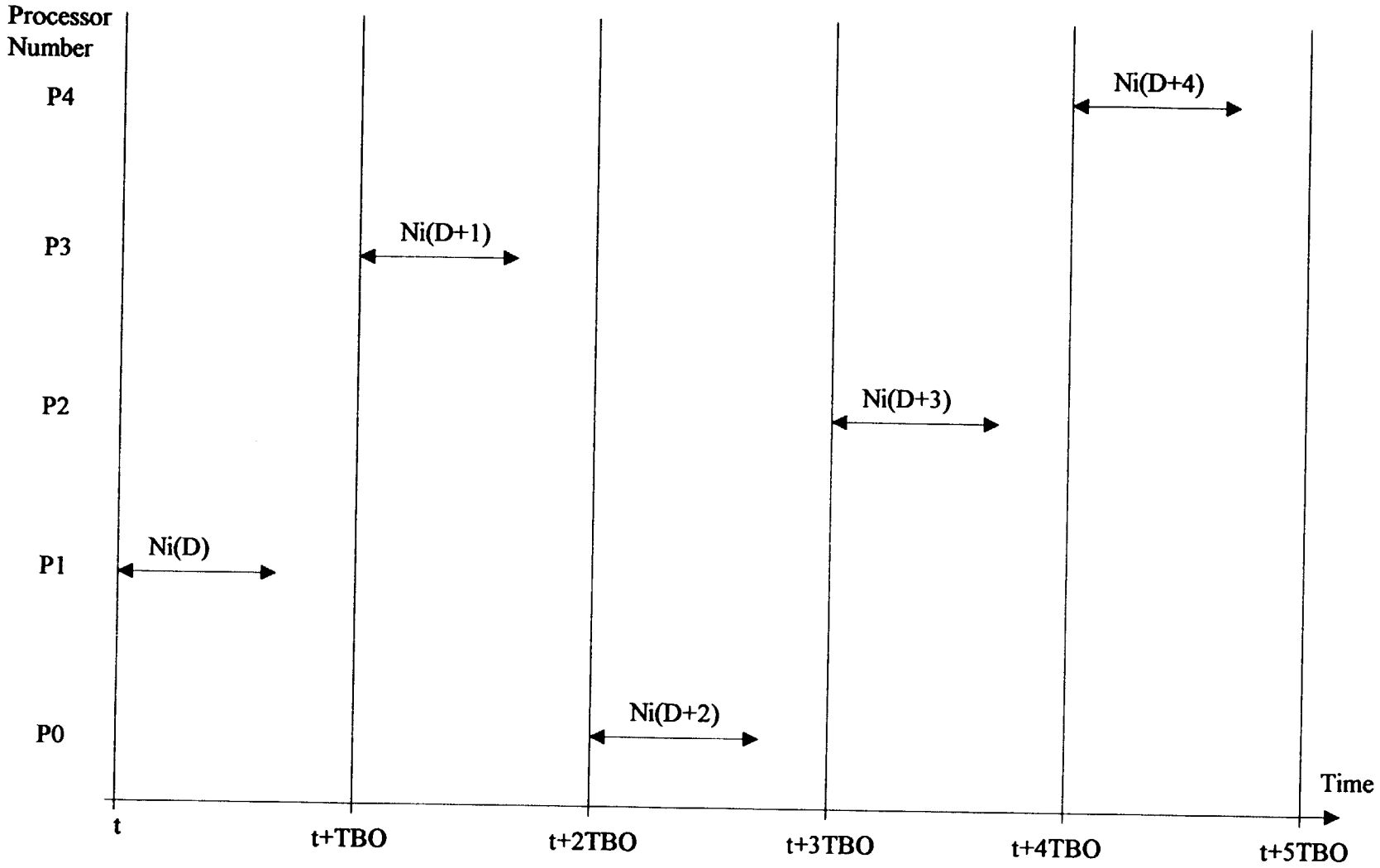


Figure 2. Node Ni in five iteration periods following the injection of packet D at time t.

instances of a node cannot be executed concurrently, but this is not a restriction for the ATAMM. When the node time of a node Nj is larger than TBO, Nj is executed for ⌈(Node time of Nj)/TBO⌉ data packets concurrently. Hence, ⌈(Node time of Nj)/TBO⌉ processors must be assigned to this node Nj as a single processor can only execute one data packet of a node at a time. Hence, it is impossible to develop a fully-static assignment for the node Nj. As a fully-static assignment of all nodes may not be possible in the extended ATAMM problem domain, a new assignment scheme called an Optimally-Static assignment [8] is defined[4]. The objective is to achieve an assignment as fully-static as possible.

Optimally-Static Assignment: A block of nodes consisting of at least one node with node time greater than the iteration period is said to be optimally-static in a block of processors provided the following rules are satisfied. If the node time of a node is less than the iteration period, it is assigned to one unique processor as in the fully-static case. For a node with node time larger than the iteration period, the node is assigned cyclo-statically with a processor displacement of 1 to a disjoint subset of the block of processors consisting of ⌈node time / iteration period⌉ processors.

As an example, consider a node Nj = 7 time units and TBO = 3 time units. Hence, ⌈7/3⌉ or 3 processors are required to execute this node. The optimally-static assignment of Nj to three processors {P0, P1, P2} is shown in Figure 3 for four iteration periods. The data packet D+1 of Nj is assigned to P{(2+1) modulo 3} or P0 because the data packet D is assigned to P2 and processor displacement is 1. No other node can be assigned to {P0, P1, P2}.

The classes of assignment schemes are pictorially described in Figure 4(a). Any assignment is either static or dynamic. The static assignment can be either periodic or aperiodic. A class of periodic assignment is the cyclo-static assignment where the

[4]This new term is coined by the author jointly with Matthew Storch of University of Illinois, Urbana, Illinois and Paul J. Hayes and Robert L. Jones of NASA Langley Research Center.

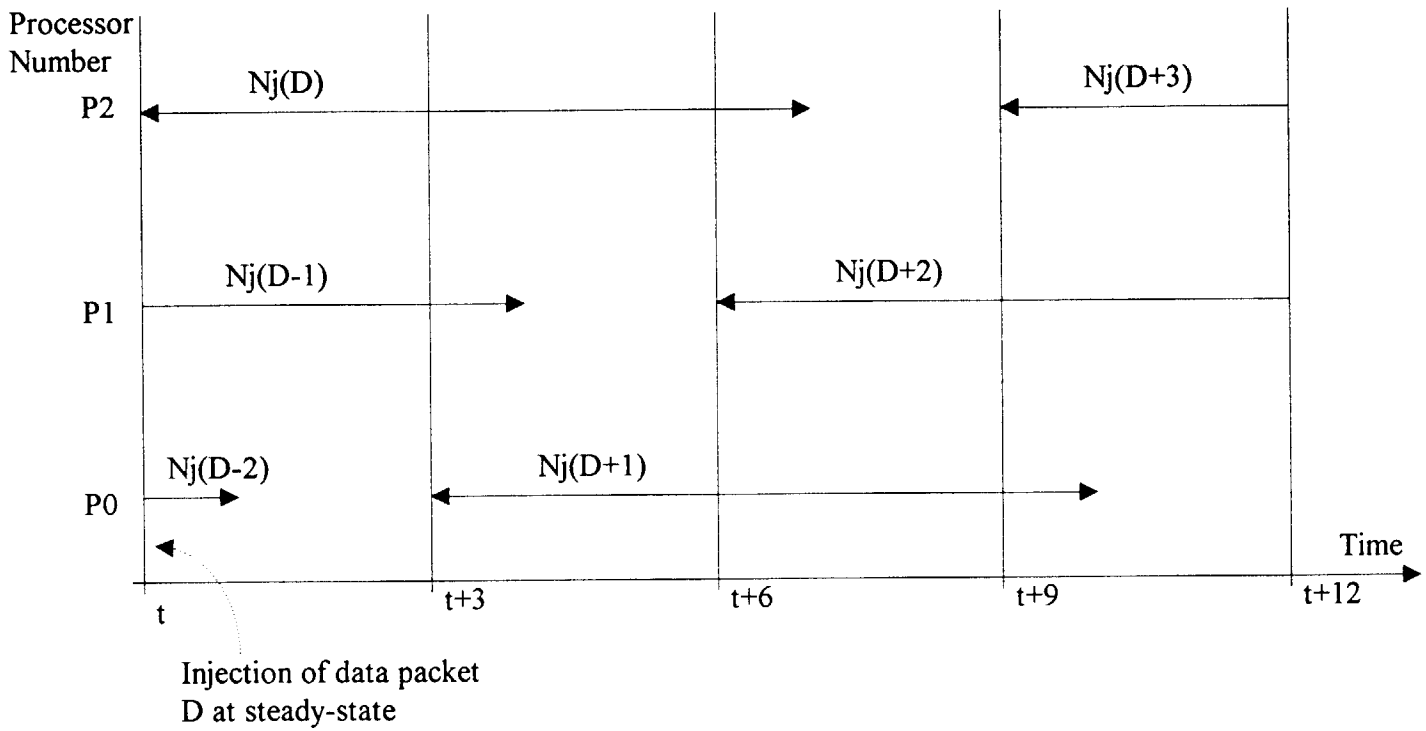
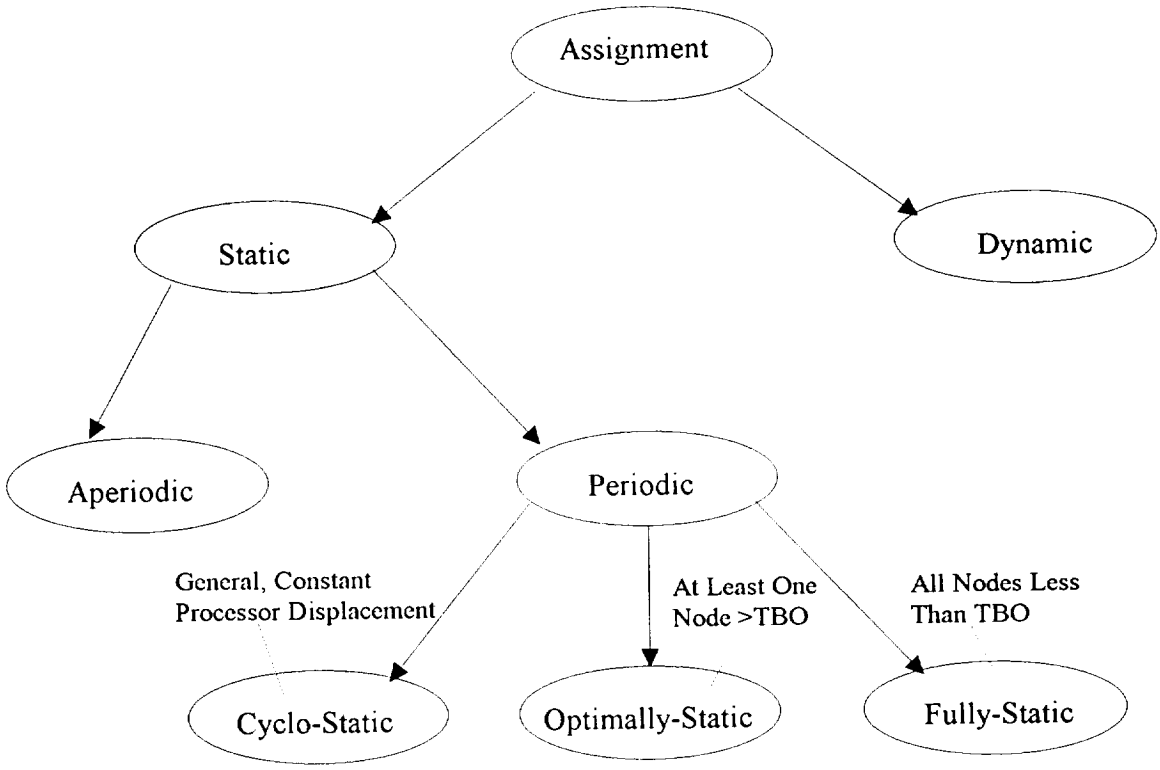
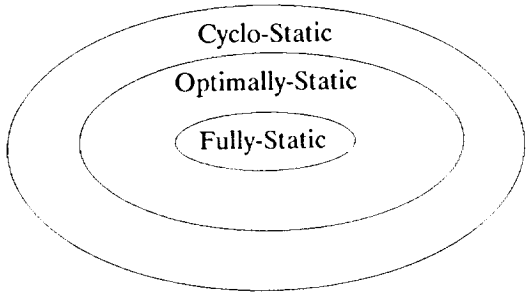


Figure 3. An optimally-static assignment of node Nj to three processors.



(a)



(b)

Figure 4. (a) Classification of assignment schemes.
(b) Relationships between Cyclo, Optimally, and Fully-Static assignments.

processor displacement from iteration to iteration are maintained to be constant. The optimally-static assignment is applicable to a block of nodes which contains at least one node with node time greater than TBO. If all nodes are less than TBO, a fully-static assignment is possible. Finally in Figure 4(b), the relationship between cyclo, optimally, and fully-static assignments is shown with respect to processor displacement. A special case of cyclo-static is optimally-static where the processor displacement is either 1 or 0. In a fully-static assignment, the processor displacement is always zero.

2.3 Token Classifications

Tokens on an AMG are classified into three categories: initial tokens, inter-iteration tokens, and intra-iteration tokens.

Initial Token: Before any data packets are injected from the source, the algorithm marked graph may contain tokens which are known as initial tokens. These tokens represent initial conditions (history) for the algorithm marked graph. A tag is attached to each initial token which identifies the data packet number for which it is to be used. The number of initial tokens on an edge is assumed to be zero or positive in this report. Also initial tokens can appear either on data or control edges and are identical to delays in signal processing literature [4].

The initial tokens on an edge also indicate inter-iteration dependency between the nodes and edges with initial tokens are referred to as inter-iteration edges. An initial token on an edge from node Ni to Nj indicates that the execution of node Nj for data packet D can only start after the finishing of execution of node Ni for data packet D-1.

Inter-Iteration Token: When the immediate predecessor of an inter-iteration edge is executed by a data packet, a token is deposited on the inter-iteration edge and the tag of the token is incremented by the number of initial tokens on the edge. This token is referred to as an inter-iteration token.

According to the ATAMM rules, a node is enabled when all incoming edges have tokens of a matching tag. Hence, for the edge from Ni to Nj as described above, data packet 1 fires Nj with the initial token. When data packet 1 finishes Ni, it produces an inter-iteration token of tag 2 on the edge from Ni to Nj. This token is used to fire Nj for packet 2.

Intra-Iteration Token: All tokens generated from a node solely based on input data packet D, are known as intra-iteration tokens of packet D. If an algorithm marked graph does not have any initial token, all tokens used in firing nodes are always intra-iteration tokens.

3. Theoretical Analysis

In this section, the ATAMM theoretical foundation is extended to include an arbitrary number of initial tokens, heterogeneous computing, and static node assignment schemes. It is proved that the periodic execution of the AMG is achieved. The characteristics of periodic graph execution are established. Algorithms are developed and proved for generating static node assignments in the ATAMM context.

3.1 Periodic Graph Execution

The periodic execution of a graph refers to a condition when each node of the AMG is fired at intervals of TBO. The single graph play for each data packet is identical. Inputs are injected and outputs are generated at intervals of TBO. For algorithm marked graphs with no initial token, periodic execution of the AMG can be achieved by injection control, insertion of buffers, and providing sufficient resources as specified by the total graph play [1]. However, for AMGs with initial tokens on either data or control edges [1], a transient condition may exist for a few packets in which periodic execution for all nodes is not possible.

When the AMG is executed periodically, it is said to be in a steady-state. With initial tokens on AMG edges, the single graph play and hence the total graph play at steady-state may be different from that during transient conditions.

Theorem 1: The execution of the AMG according to the ATAMM rules always reaches a steady-state when the single graph play for each data packet is identical.

Proof: By the ATAMM rules, a node in the AMG must be able to fire as soon as all input edges have required tokens. A node may be fired by either initial tokens, intra-iteration tokens, inter-iteration tokens, or by a combination of all of them. Let the injection time for the first data packet be denoted as t = 0. The second data packet is injected at t = TBO and a new data packet is injected after every TBO time interval. The nodes for a data packet D which are fired solely by intra-iteration tokens are always enabled after the longest path time from the source to the node. Firing by intra-iteration tokens means that all tokens used to fire this node or all its predecessors are generated from the data packet D. With sufficient resources and output buffers, these nodes will be executed after equal time intervals from the injection of the data packet D. Hence the single graph play for these nodes will look identical for any data packet. Next the nodes for the data packet D which require either initial tokens or inter-iteration tokens directly or indirectly are considered. The indirect use of initial tokens or inter-iteration tokens refers to the condition when a node does not require such tokens but its predecessors are fired with these tokens. An inter-iteration token for packet D is generated from the finishing of a node in iteration (D-h) where h is a positive number. This inter-iteration token is always produced after a fixed time interval from the injection of the data packet (D-h) regardless of the value of D. Hence, when only inter-iteration and/or intra-iteration tokens are used to fire a node of packet D, the node is always executed after equal time intervals from the injection of packet D. With a finite number of initial tokens, all tokens will be either intra-

iteration or inter-iteration after executing a finite number of data packets. Hence the AMG will reach a steady-state when all SGPs will look identical for all data packets. However, for small values of D, initial tokens will be used instead of inter-iteration tokens. All the initial tokens are present in the AMG at time t = 0. Therefore, when an initial token is involved in firing a node for data packet D either directly or indirectly, the node may be fired earlier than when inter-iteration tokens are used for large values of D. Hence, there may be a transient condition during the execution of the first few data packets in which some nodes are fired earlier than that specified by the steady-state SGP. However, execution of any AMG according to AMG rules must reach a steady-state. **This completes the proof of Theorem 1.**

As an example, consider an edge E from Ni to Nj with one initial token. Let Ni be fired with only inter-iteration tokens. Also, let E be the only incoming edge for Nj. For data packet 1, Nj can fire at t = 0 itself. Let data packet 1 finish node Ni at t = T1. Hence for data packet 2, node Nj is fired at t = T1. As data packet 2 is injected TBO time units after data packet 1, node Ni for data packet 2 is finished at time T1+TBO. This occurs because all data packets must take equal time to fire nodes which depend solely on intra-iteration tokens. The node Nj for data packet 2 is fired at T1+TBO which is T1 time units after injection of packet 2. Similarly, data packet 3 fires node Nj at T1+2TBO which is T1 time units after injection of data packet 2. Hence, Nj for a packet D is fired after fixed time intervals T1 from the injection of packet D provided D is greater than 1. Hence, the steady-state SGP will always show execution of Nj in a fixed time interval.

Lemma 1: In steady-state, each node of the AMG is executed at TBO intervals.

Proof: The single graph play for each data packet is identical at steady-state. Data packets are injected in intervals of TBO. Every node is executed exactly once for each data packet. Therefore, the firing of each node of the AMG must occur at the interval of TBO. **This completes the proof of Lemma 1.**

Lemma 2: In one iteration period TBO at steady-state, a node may execute multiple data packets. The total execution time of a node for all data packets in one iteration period must equal the node time.

Proof: As the TGP is constructed with sections of the SGP, the complete execution of each node will always appear exactly once in one iteration period of the TGP. The execution of a node in the SGP may be divided in more than one section. As each section carries an unique data packet number, multiple data packets may execute the same node in one iteration period. **This completes the proof of Lemma 2.**

Lemma 3: The total graph play for every iteration period at steady-state is identical in the sense that all node execution lines are the same but the data packet number on a node execution line is incremented by 1 from current to next iteration period.

Proof: At steady-state, the single graph play for all data packets is identical. Hence, the TGP must always have identical node execution lines. Consider two successive iteration periods of TGP beginning at the injection of data packets D (at time t) and D+1 (at time t+TBO). The TGP for these iteration periods are derived from sections of SGPs for data packets D and D+1, respectively. Hence by the construction process, the data packet number for a node execution line segment in iteration period beginning (t+TBO) must be one higher than that in the previous iteration period. **This completes the proof of Lemma 3.**

The original method of generating the SGP by firing every node as early as possible needs to be modified for graphs with initial tokens. An analytical method for generating the steady-state SGP for any graph has been developed[5]. An alternative method based on simulation is to execute the AMG until it reaches a steady-state. When the SGP does not change with increasing data packet numbers, the AMG has reached

[5]Developed by Robert L. Jones, NASA Langley Research Center, Hampton, Virginia. A software tool called the ATAMM Design Tool can automatically generate the steady-state SGP and TGP.

steady-state. The simulation method of generating steady-state SGP and TGP is illustrated below by an example.

Consider the AMG in Figure 5 which is taken from [4]. The largest time per token ratio from all loops in the AMG is 4. Let TBO be 4. The execution of data packets 1 through 5 are shown in Figure 6. The node A is always repeated at the interval of TBO. However, the execution of node B reaches steady-state from the third data packet. The execution of B(1) and B(2) are triggered by initial tokens on the edge from A to B and hence do not reflect the steady-state periodic nature. B(3) is fired by the inter-iteration token from A(1). Similarly, B(4) and B(5) are fired by inter-iteration tokens from A(2) and A(3). As node A always starts and finishes at the interval of TBO, the execution of node B is also periodic with TBO beginning with the third packet. The steady-state TGP is then obtained from the steady-state SGP and is described in Figure 7(a).

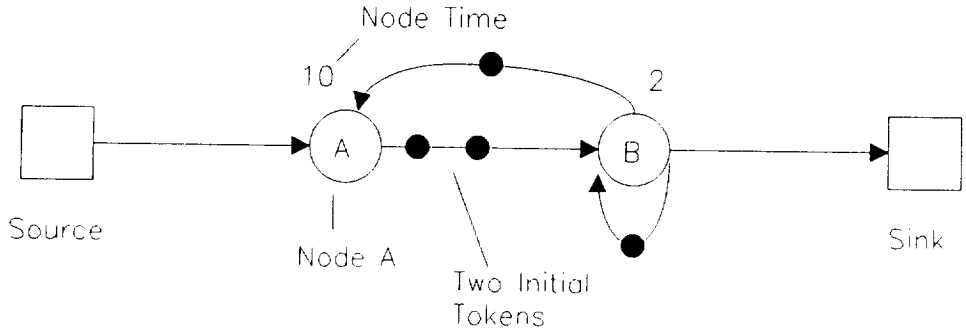


Figure 5. An Algorithm Marked Graph with initial tokens.

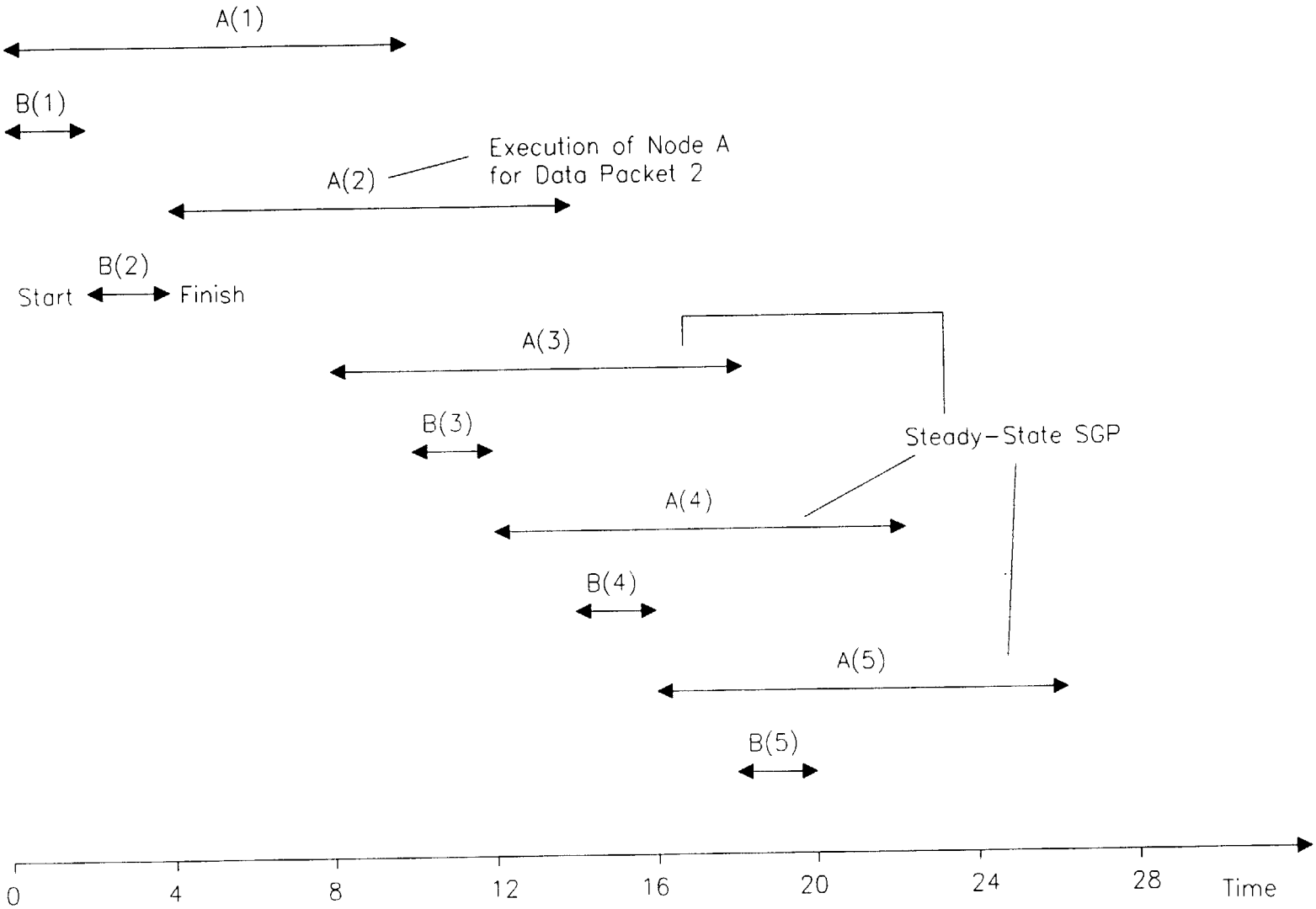
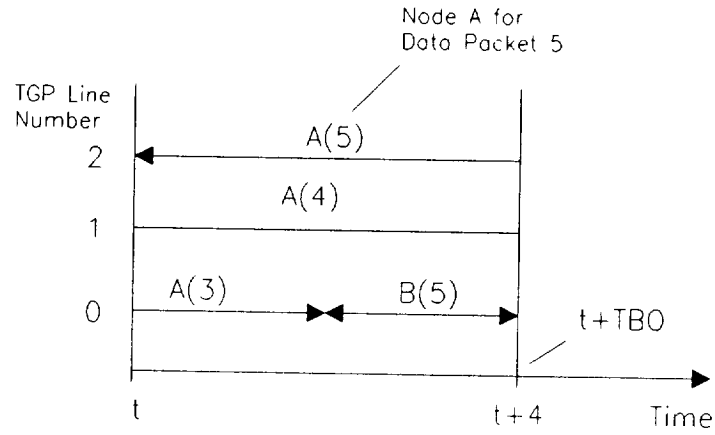
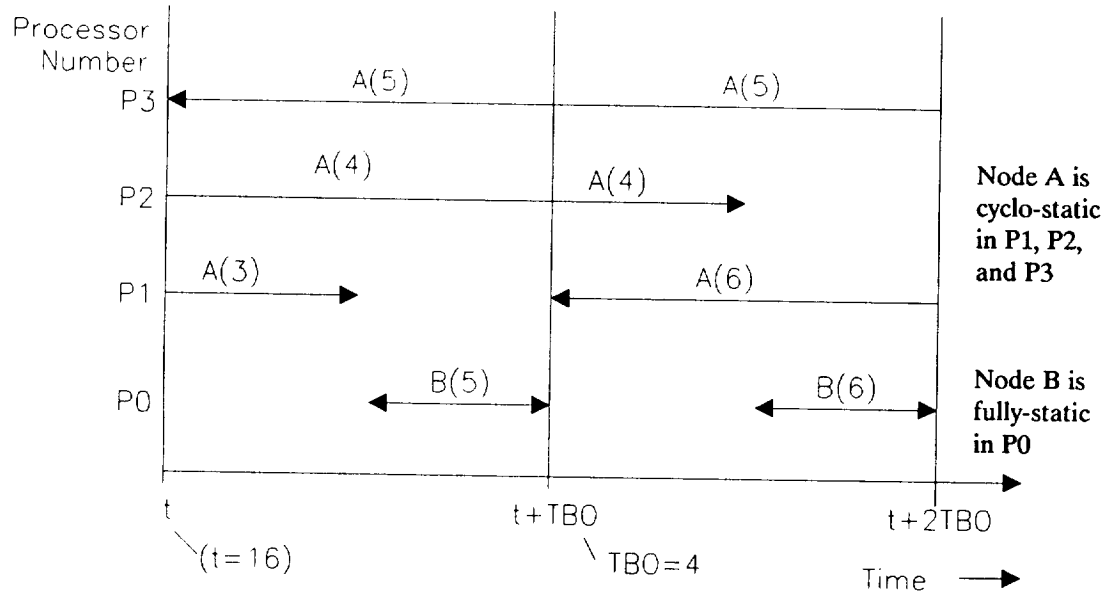


Figure 6. Execution of data packets 1 through 5 with injection interval = 4.



(a)



(b)

Figure 7. (a) Total Graph Play for TBO = 4.
(b) Two iteration periods of TGP with optimally-static assignment.

3.2 Cyclo-Static Assignment

In the following sub-section, it is shown that a cyclo-static assignment can always be achieved in the ATAMM. An algorithm, called Algorithm 1, is presented and proved to generate a cyclo-static assignment with a processor displacement of either one or zero[6]. Algorithm 1 is used to partition nodes of a given AMG into blocks. Then each block of nodes is assigned to a block of processors. It is assumed that all nodes of the AMG execute on the same type of processors.

Algorithm 1:

1) Construct one iteration period of the steady-state TGP for the AMG. Let t be the beginning of the iteration period.

2) To form a block, select a node whose execution begins at time t.

3) If no such node is available, select a node from the TGP whose execution begins immediately following the end of execution of an already selected node[7].

4) If the node selected does not end at (t+TBO), select a new node which begins immediately following the end of the last selected node. Repeat until all nodes are assigned or the last node selected ends at (t+TBO). The block is completed at this point.

5) Redraw the TGP to show the execution of the selected node(s) for the block together. If a selected node is executed for two or more data packets in one iteration period, show the complete node execution, from start to finish, in consecutive lines of the redrawn TGP.

---

[6] The cyclo-static nature of graph execution in the ATAMM was initially observed by Mahyar R. Malekpour of Lockheed Engineering & Sciences Company, Hampton, Virginia and Robert L. Jones of NASA Langley Research Center, Hampton, Virginia.

[7] For the formation of the first block, a node is guaranteed to begin at time t as a new data packet is injected at time t. Hence, Step 2 will select a node and Step 3 will be skipped. From the second block onwards, Step 2 alone may not be able to select a new node. However, as every AMG node is fired at the completion of another AMG node or source, Steps 2 and 3 together must be able to select a node for a new block.

6) A new node block is formed by the selected nodes. Assign processors to this block equal to the number of concurrent lines[8] in the corresponding portion of the TGP.

7) If all nodes of the TGP are not yet selected, go back to Step 2 to form another block. Each node is selected only once in the algorithm.

Theorem 2: Each block of nodes is mapped with a cyclo-static assignment by Algorithm 1. The processor displacement is 1 for two or more processors and 0 for one processor.

Proof: Consider a block of nodes created by Algorithm 1. The number of assigned processors is equal to the number of node execution lines in the TGP segment corresponding to this block. For the rest of the proof, the term TGP refers to this TGP segment only. Number the TGP node execution lines as 0, 1, 2 ... (q-1) and the corresponding processors as P0, P1, P2, ... P(q-1), from the bottom to top line of the TGP segment. Lines (q-1) and 0 are the beginning and the end of node executions for this block, respectively. Let q > 1 such that at least two processors are assigned to this block. By the construction process of a block, any two consecutive lines of a block have a common node with two different data packet numbers. If the execution of a selected node Ns for data packet D is not completed by the end of the current iteration period on line j (j not equal to 0), there must also be execution of Ns in line (j-1) with data packet D-1 in the current TGP. As this is a non-preemptive schedule, processor P(j-1) and Pj execute data packets (D-1) and D of Ns, respectively. Also in the next iteration period, the execution of node Ns for packet D will continue on line j and processor j. Hence the TGP line (j-1) in the current iteration period is shifted to line j in the next iteration period with an increase of 1 in the data packet number. Hence a node which begins in line (j-1) with

[8]These are the TGP lines which overlap in time.

packet Dp and is assigned to processor (j-1) in the current iteration period, will be assigned to processor j with data packet (Dp+1) in the next iteration period. This is a cyclic assignment with processor displacement of 1. For line 0 of this TGP segment, P0 will always complete execution of all nodes by the current iteration period. Hence, a node with packet D on line (q-1) can be assigned to P0 in the next iteration period with packet D+1. This is a displacement of 1 if processor numbers are calculated in modulo q. Hence Algorithm 1 always produces a cyclo-static assignment with processor displacement of unity. If q = 1, only one processor is assigned to this block. Hence, all iterations of all nodes in this block must be executed by the same processor. This is a cyclo-static assignment with processor displacement of 0 and is the special case referred to as fully-static. **This completes the proof of Theorem 2.**

As an example, consider the AMG of Figure 1(a). N0 = 4, N1 = 1 and N2 = 5 time units and the corresponding steady-state TGP for TBO = 4 is shown in Figure 1(c). Let a cyclo-static assignment be desired. By Step 2, N0 is selected as the first node of a block. As N0 ends right on t+TBO, no new node is selected. Hence, {N0} is a single node block and 1 processor, P2, is assigned to this block. P2 will always execute N0 and therefore mapping of N0 to P2 is both cyclo-static as well as fully-static. As all nodes are not yet selected, Step 2 is repeated and N1 is selected for a new block. By Step 4, N2 is selected. A new block is formed as {N1, N2} in Step 6. From the TGP, two processors, P0 and P1, are assigned to {N1, N2}. The cyclo-static nature of scheduling is shown in Figure 8 by displaying the TGP for multiple iteration periods. N1 for data packets 2, 3, and 4 are executed by P1, P0, and P1 respectively. Similarly, node N2 is assigned to P1 and P0 alternately. Hence the mapping of {N1, N2} to {P0, P1} is cyclo-static with a processor displacement of 1.

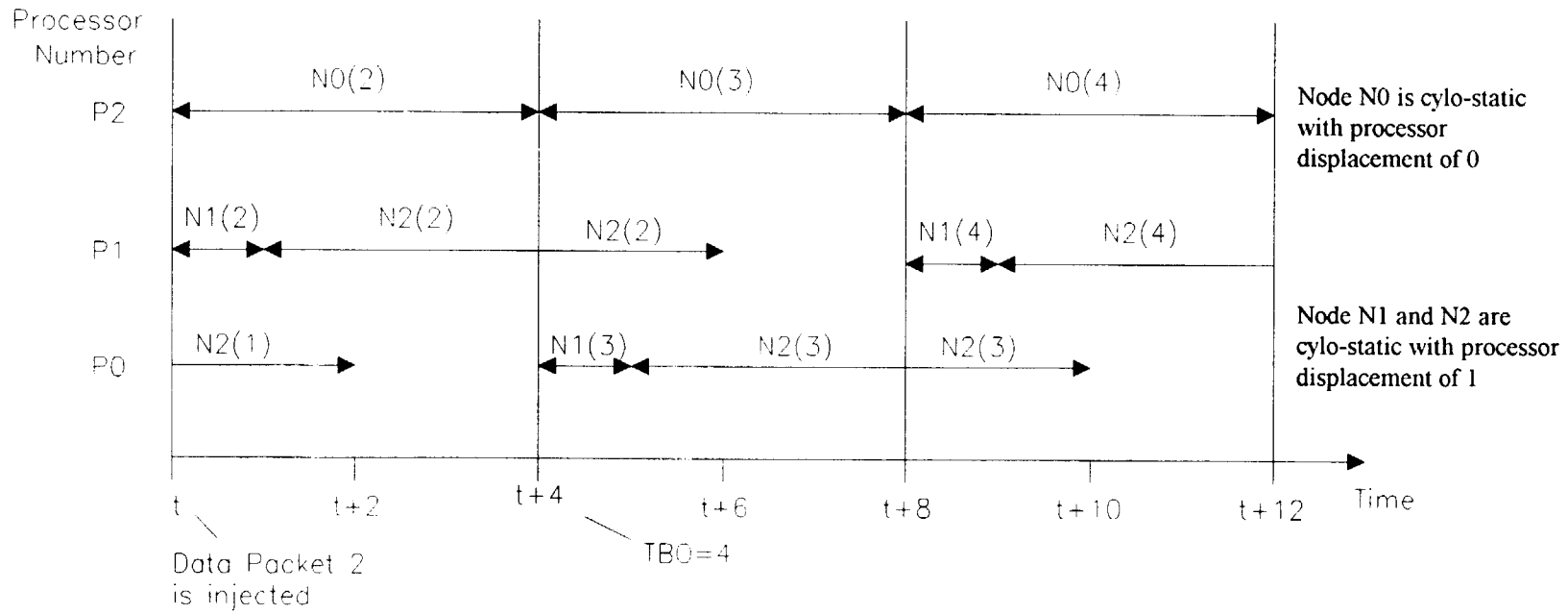


Figure 8. Three iteration periods of Total Graph Play of the AMG of Figure 1(a), following the injection of data packet 2.

Lemma 4: In a cyclo-static assignment by Algorithm 1, each processor must either have the code or access to the code for all nodes of the assigned block of nodes.

Proof: As a processor is required to execute every line of TGP and hence any node of the assigned block of nodes, the processor must have necessary codes. Otherwise, the processor must be able to fetch the required codes. **This completes the proof of Lemma 4.**

3.3 Optimally-Static And Fully-Static Assignments

It may be desirable that all nodes are assigned in a fully or optimally-static manner, depending on the node sizes with respect to TBO. A heuristic algorithm called Algorithm 2 is developed for generating such an assignment for a given AMG. It is assumed that all nodes of the AMG execute on the same processor type. If all node times are less than or equal to TBO, a fully-static assignment is generated. If the block of nodes contain at least one node larger than TBO, an optimally-static assignment is generated. This corresponds to AMGs with multiple concurrent instantiations of nodes. It is to be noted that optimally or fully-static assignments may result in low processor utilization.

Algorithm 2:

1) Construct one iteration period of the steady-state TGP for the AMG. Redraw the TGP with the following rules.
2) For all nodes larger than TBO, show the node execution in consecutive lines. Form a block of processors equal to the number of concurrent lines[9] and assign processors to this node.
3) Follow Steps 4 through 6 for nodes with node time less than or equal to TBO.
4) Depict the execution of a node always in one line of the TGP.
5) Combine non-overlapping node executions on TGP lines while satisfying Step 4 and precedence constraints[10].

[9]This corresponds to parallel execution of multiple data packets of the same node.

[10]This is to reduce the number of TGP lines and hence required processors.

6) Form a block of nodes consisting of all nodes on one TGP node execution line. Assign one processor to this block of nodes.

7) Calculate the total number of processors from all blocks of processors.

Theorem 3: Algorithm 2 produces a fully-static assignment if all nodes are less than TBO and an optimally-static assignment if at least one node is larger than TBO.

Proof. For nodes less than TBO, Step 4 ensures that the execution of a node Ns, regardless of data packet numbers, is always on one line in the TGP. Hence, only one processor which is assigned to nodes on this line of TGP will execute Ns in every iteration period. Therefore, the node Ns is fully-static on this processor. By Steps 5 and 6, a number of nodes may be assigned to the same processor but a node always appears in only one line of TGP. Hence, a node will always belong to one block of nodes which is always assigned to the same processor. When a node Ns is larger than TBO, the execution of Ns will be shown in multiple lines in the TGP and data packet numbers on these lines decrease by 1 from top to bottom. Then Ns is assigned to a block of processors as in Algorithm 1. By Theorem 2, Ns is assigned to the block of processors cyclo-statically with a processor displacement of 1. **This completes the proof of Theorem 3.**

An optimally-static assignment is generated for the AMG of Figure 5. The TGP is redrawn in Figure 7(b) according to Algorithm 2. Three processors are required for node A. One processor is required for node B. While node B is fully static on P0, node A is assigned cyclically to P1, P2, and P3. Note that processor utilization is less than 100% as indicated by the gaps in processor activity in the TGP, for this assignment.

4. Heterogeneous Computing

The objective of this section is to use the extended theoretical foundation in heterogeneous multiprocessor scheduling and design. For this purpose, a design objective is defined. A methodology for scheduling is developed and illustrated through an example.

4.1 Problem Formulation

An algorithm marked graph is to be mapped on a heterogeneous architecture. Nodes of the algorithm marked graph and processors of the architecture are to be partitioned in blocks based on type. A block of the node set N is to be assigned to a block of the processor set P. This assignment of a type of nodes to a type of processors is one-to-one and fully-static. However, the assignment of nodes within one type may be either static or dynamic.

The mapping of nodes onto processors is done on the basis of an objective function while satisfying some constraints.

Objective Function: The iteration period or TBO is specified. The computing time or TBIO can be increased up to a specified deadline. The objective is to minimize the number of required processors of each type and generate a schedule and node-to-processor assignment scheme based on the constraints.

The constraints can be classified into three categories: 1) Type Constraint, 2) Memory Constraint, and 3) Precedence Constraint.

Type Constraint: Each node and processor has unique characteristics. A node must be mapped onto a processor which is capable of executing the node.

Memory Constraint: The total amount of memory needed in a processor due to the code and data of assigned nodes must be less than a specified percentage of the total available memory in that processor.

Precedence Constraint: Any valid schedule of the AMG must satisfy both inter-iteration and intra-iteration precedence constraints. Both of these precedence constraints are

depicted in the total graph play (TGP). New precedence constraints of both kinds can be created by the insertion of control edges [1].

The following observations can be made. As there may be different types of nodes, processor requirements have to be calculated separately for each type of node. Hence, the TGP should be segmented into different portions corresponding to node types. Control edges can still be inserted between any two nodes of the AMG in order to modify a TGP segment. However, the processor requirements for a particular node type are determined only from the corresponding segment of the TGP. Let TCE_S denote the total computing effort for a TGP segment.

4.2 Design Method

A heuristic design procedure is described below.

1) Partition the set of nodes in blocks based on type. Identify the processor type for each block of N.
2) Generate the steady-state SGP and TGP. Divide the TGP into segments corresponding to each type.
3) If $\lceil TCE_S / TBO \rceil$ is less than the number of concurrent lines in a TGP segment, modify the TGP segment with control edges, if possible, to reduce the processor requirements while not violating any specified deadline.
4) Based on the memory constraint and node sizes, select either a dynamic, cyclo-static, optimally-static, or a fully-static assignment scheme for each type.
5) Use Algorithm 1 for a cyclo-static assignment.
6) If a fully-static or an optimally-static assignment is required, use Algorithm 2.
7) Repeat Steps 4 through 6 for each type of processor.
8) Determine the number of processors required for each processor type.

Consider the AMG of Figure 9 which is a combination of the AMGs of Figures 1(a) and 2. By the time per token ratio, the lower bound on TBO is 4. Let nodes A and B be of type 1 and nodes N0, N1, and N2 be of type 2. By Step 1, nodes are partitioned into

blocks {A, B} and {N0, N1, N2}. The steady-state SGP and segmented TGP are shown in Figure 10. For block {N0, N1, N2}, $TCE_S = 10$ and hence, at least 3 processors are required. Also, block {A, B} requires at least 3 processors for TBO = 4. Hence, control edges cannot reduce processor requirements and Step 3 is skipped. If dynamic assignment is desired for both type of nodes, the TGP of Figure 10(b) is achieved by assigning three processors of type 1 to {A, B} and three processors of type 2 to {N0, N1, N2}. No node-to-processor assignment is necessary within a block. Now let both blocks {A, B} and {N0, N1, N2} be assigned as cyclo-static with processor displacement of 1 or 0. Hence, Algorithm 1 is used in Step 5. The resulting total graph play and processor assignment are shown for three iteration periods in Figure 11. Three processors are required for each type.

As an alternative, suppose both blocks {A, B} and {N0, N1, N2} are to be assigned as optimally-static. The construction of such an assignment by Algorithm 2 is shown in Figure 12 which requires four processors of each type. It is clear that the optimally-static assignment requires more processors compared to cyclo-static (and dynamic assignment) and thereby will have a lower processor utilization.

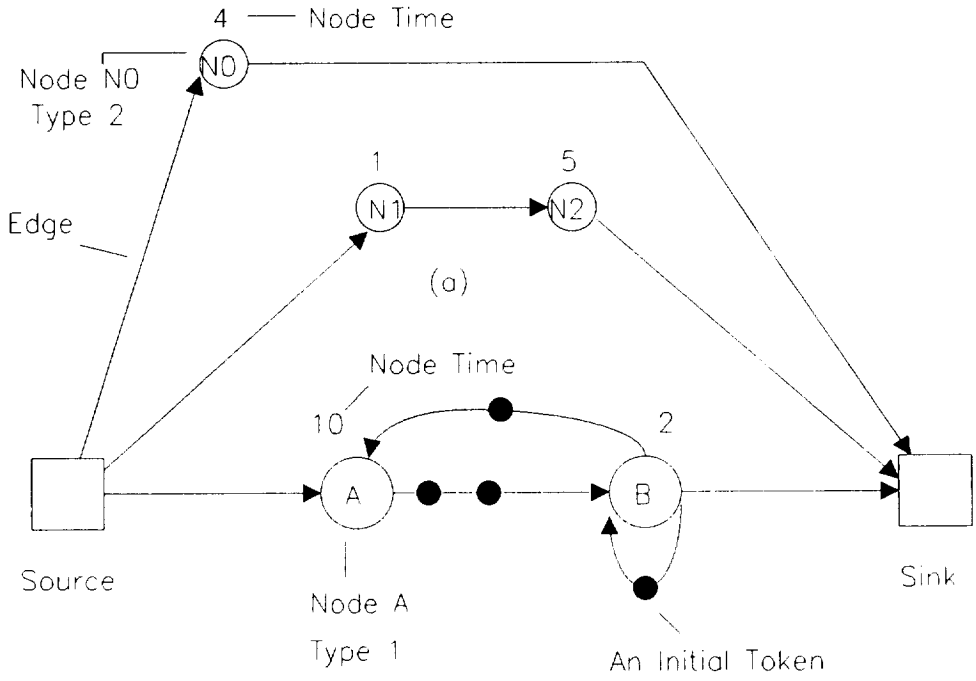
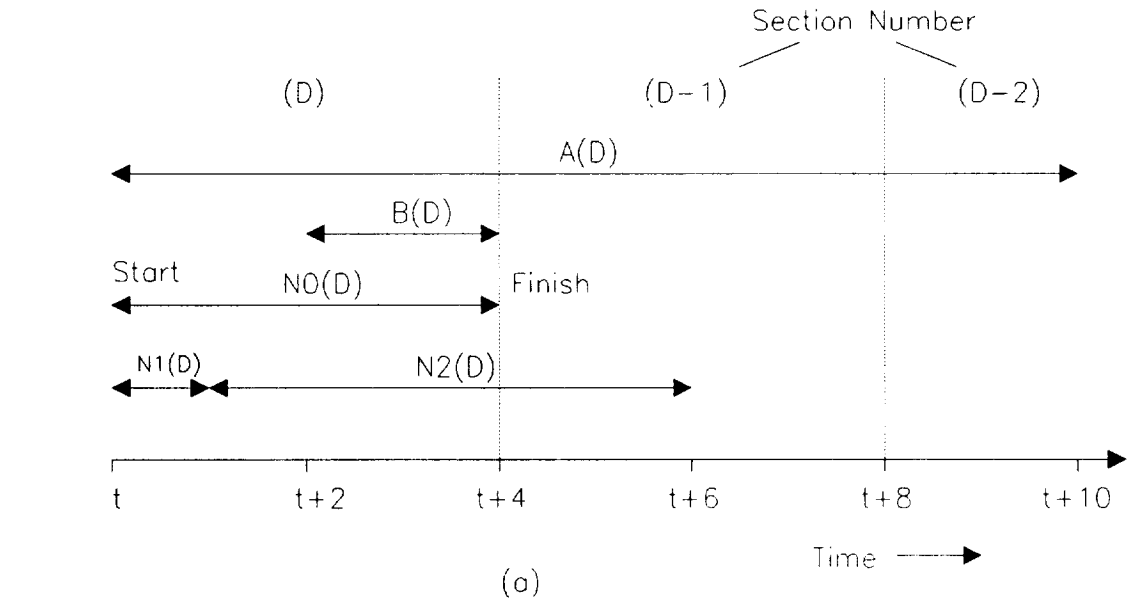


Figure 9. A heterogeneous Algorithm Marked Graph.



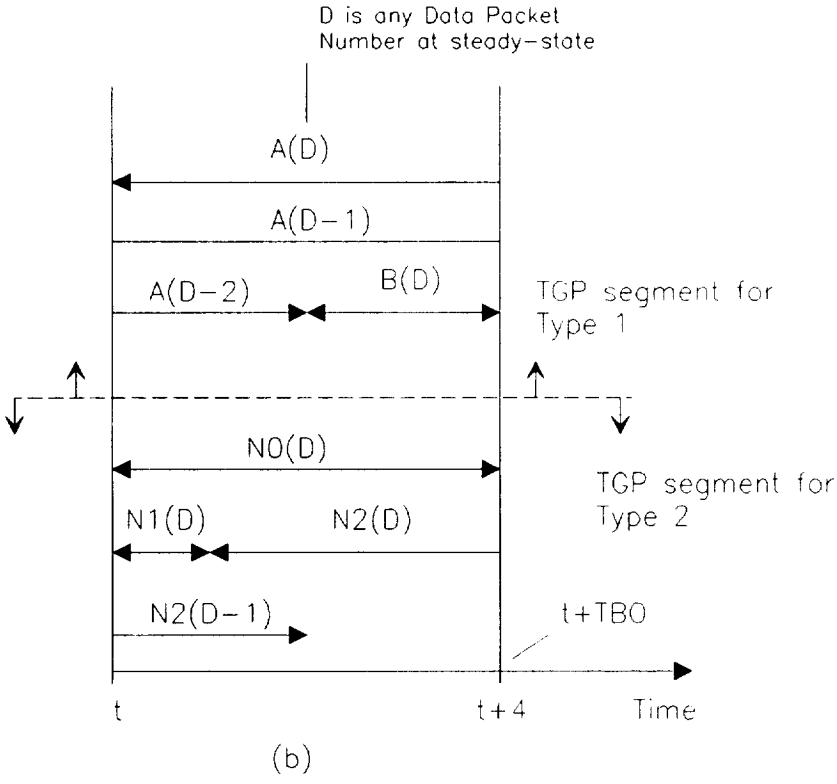


Figure 10. (a) Steady-state Single Graph Play.

(b) One iteration period of steady-state Total Graph Play for TBO=4.

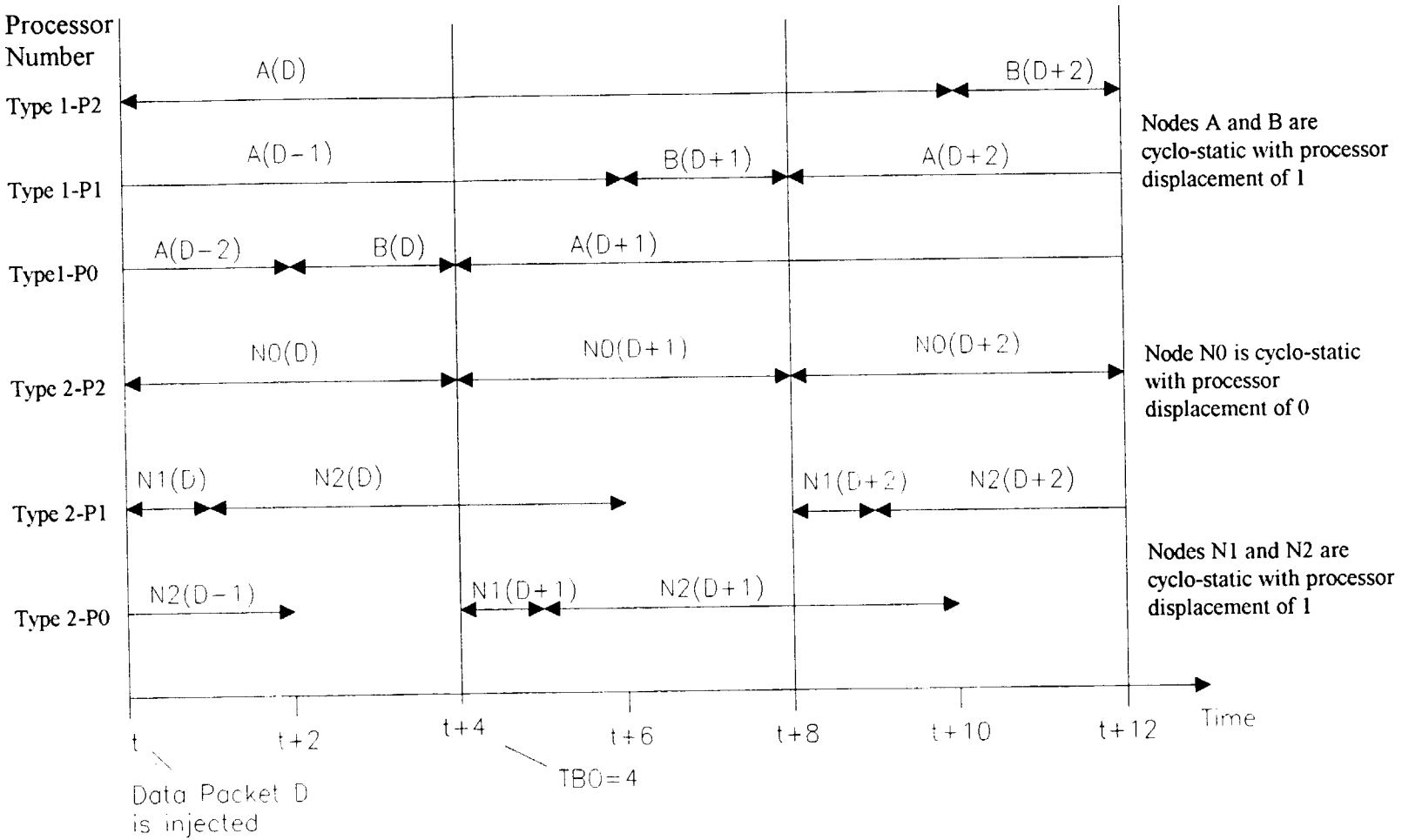


Figure 11. Cyclo-static assignments for both type 1 and type 2 nodes.

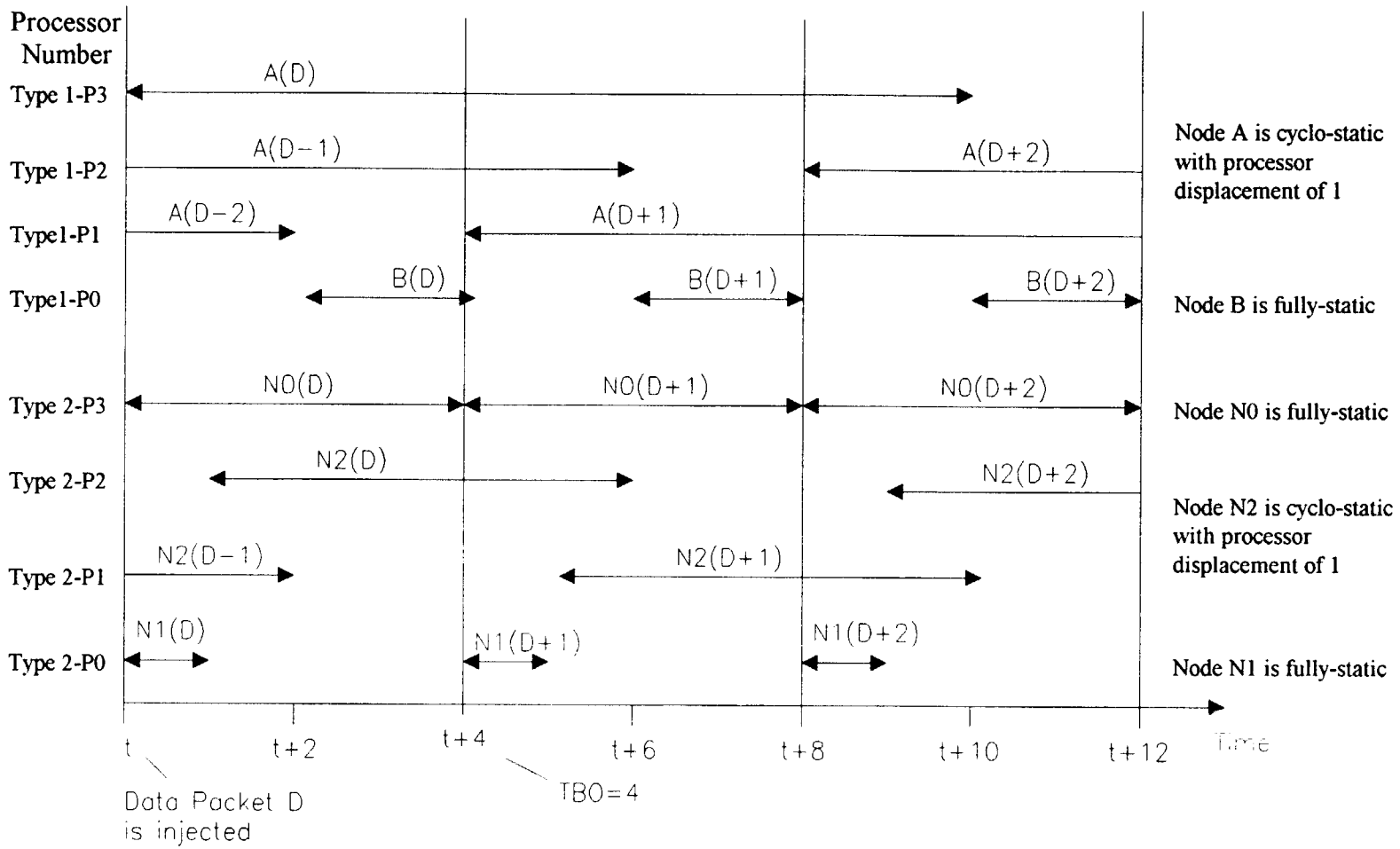


Figure 12. Optimally-static assignments for both type 1 and type 2 nodes.

5. Conclusions

The primary objective of this research has been to investigate an extended ATAMM approach for node assignments in heterogeneous architectures. Several key results are achieved in that respect. It is established that ATAMM can incorporate both dynamic and static node assignment schemes for both heterogeneous and homogeneous architectures. A new optimally-static assignment scheme is defined to determine the closest possible fully-static node assignment in the ATAMM dataflow architectures. Also, periodic execution can still be maintained at steady-state for the extended class of problems. Included in this report is a formalism for the partitioning of nodes and the assignment of nodes in multiprocessor architectures.

6. Future Research

In the context of this report, the following topics are identified as subjects of current and future research.

- Heterogeneous architectures should be studied with a finite number of processors and processor types.
- A methodology needs to be developed for comparing the performance of heterogeneous architectures.
- The feasibility of reducing communication bottlenecks by static node assignment should be explored [9, 10].
- The initial transient nature of graph execution requires further investigation for effects on overall predictability and duration of transients.
- Heuristic and extensive search techniques should be explored for selecting control edges.
- Finally, non-causal precedence constraints created with a negative number of initial tokens should be examined.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1993	Contractor Report, May-August 1992

4. TITLE AND SUBTITLE
Node Assignment in Heterogeneous Computing

5. FUNDING NUMBERS
C NAS1-18936, Task 15
WU 586-03-11

6. AUTHOR(S)
Sukhamoy Som

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
CTA INCORPORATED
1 Enterprise Parkway, Suite 390
Hampton, VA 23666

8. PERFORMING ORGANIZATION REPORT NUMBER
038-3524-150-93-1

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-4534

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Paul J. Hayes. The author is currently employed by Lockheed Engineering & Sciences Company, Hampton, VA 23666.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 33

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A number of node assignment schemes, both static and dynamic, are explored for the Algorithm to Architecture Mapping Model (ATAMM). The architecture under consideration consists of heterogeneous processors and implements dataflow models of real-time applications. Terminology is developed for heterogeneous computing. New definitions are added to the ATAMM for token and assignment classifications. It is proved that a periodic execution is possible for dataflow graphs. Assignment algorithms are developed and proved. A design procedure is described for satisfying an objective function in an heterogeneous architecture. Several examples are provided for illustration.

14. SUBJECT TERMS
Iterative dataflow, multiprocessing, static scheduling, dynamic scheduling

15. NUMBER OF PAGES
40

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

References

1. S. Som, J. W. Stoughton, and R. R. Mielke, "Strategies for Concurrent Processing of Complex Algorithms in Data Driven Architectures," *NASA Contractor Report 187450*, Grant NAG1-683, October 1990.

2. R. Mielke, J. Stoughton, S. Som, R. Obando, M. Malekpour, and B. Mandala, "ATAMM Multicomputer Operating System Functional Specification," *NASA Contractor Report 4339*, Grant NCC1-136, November 1990.

3. S. Som, R. Mielke, R. Obando, J. Stoughton, P. J. Hayes, and R. L. Jones, "Throughput Enhancement by Multiple Concurrent Instantiations in the ATAMM Data Flow Architecture," *Proceedings of the ISMM International Symposium on Computer Applications in Design, Simulation, and Analysis*, pp. 71-74, Las Vegas, Nevada, March 19-21, 1991.

4. Kesab. K. Parhi and David G. Messerschmitt, "Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding," *IEEE Transactions on Computers*, pp. 178-195, Vol. 40, No. 2. February 1991.

5. Sonia M. Heemstra de Groot and Sabih H. Gerez, and Otto E. Herrman, "Range-Chart-Guided Iterative Data-Flow Graph Scheduling," *IEEE Transactions on Circuits and Systems*, pp. 351-364, Vol. 39, No.5. May 1992.

6. D. A. Schwartz and T. P. Barnwell, III, "Cyclostatic Multiprocessor Scheduling for the Optimal Implementation of Shift Invariant Flow Graphs," in *Proceedings of the ICASSP-85*, Tampa, FL, March 1985.

7. Zvi Kohavi, "*Switching and Finite Automata Theory*," Second Edition, Mcgraw-Hill, New York, 1978.

8. Matthew Storch, "A Comparison of Multiprocessor Scheduling Methods for Iterative Data Flow Architectures," *NASA Contractor Report 189730*, Grant NAG1-613, February 1993.

9. Nicholas S. Bowen, Christos N. Nikolaou, and Arif Ghafoor, "On the Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computer Systems." *IEEE Transactions on Computers*, pp. 257-273, Vol. 41, No, 3, March 1992.

10. B. Narahari and Hyeong-Ah-Choi, "Allocating Partitions To Task Precedence Graphs," *Proceedings of the IEEE 1991 International Conference on Parallel Processing*, vol. 1, pp. 621-624, August 1991.