N94- 35053

Modelling Heterogeneous Processor Scheduling for Real Time Systems

J. F. Leathrum, R. R. Mielke, and J. W. Stoughton
Old Dominion University

Abstract

A new model is presented to describe data-flow algorithms implemented in a multiprocessing system. Called the resource/data flow graph (RDFG), the model explicitly represents cyclo-static processor schedules as circuits of processor arcs which reflect the order that processors execute graph nodes. The model also allows the guarantee of meeting hard real-time deadlines. When unfolded, the model identifies statically the processor schedule. The model therefore is useful for determining the throughput and latency of systems with heterogeneous processors. The applicability of the model is demonstrated using a space surveillance algorithm.

Introduction

Improvement in throughput and latency in hard real-time algorithms increasingly is realized through the use of parallel constructs. However, it is known that the characterization of performance in parallel systems is particularly difficult, and is compounded when a heterogeneous processing environment is introduced. The processing time of a particular piece of code is dependent on the processor type scheduled to execute it. This may complicate the analysis of throughput and latency. Current strategies focus on the use of data-flow graphs to describe algorithm play, and then an external processor scheduling scheme is imposed for graph execution. A method for the development of processor scheduling within the data-flow model is presented. It provides a deterministic method of predicting the effects of various schedules, both in homogeneous and heterogeneous processor environments.

Scheduling tasks on parallel computers can be divided into two categories, static scheduling and dynamic scheduling. Static scheduling methods [1,2,3,4,5,6] allocate tasks to processors during compile time. The time required to schedule is incurred only once, independent of the number of times the application is executed. Dynamic scheduling methods [7,8] allocate tasks to processors at run time, taking advantage of current knowledge of the state of the system. Dynamic scheduling methods generally provide better resource utilization, but at the penalty of real-time overhead to complete the scheduling. Dynamic scheduling also may incur a degradation in expected performance resulting from slight variations in task time, even when the task time decreases. Heterogeneous systems generally employ dynamic schedulers [7,8].

The applications under consideration in this paper have hard real-time deadlines. Once an input arrives, the corresponding output must be generated by some maximum time deadline. Inputs must be accepted at some maximum rate while still meeting performance deadlines. Consequently, completing tasks as early as possible is not as important as avoiding a late finish.

The paper starts by reviewing the data-flow graph model implemented on a homogeneous processor system [9]. Methods of analyzing performance are described, and are followed by a discussion of how the graph plays in steady state in order to establish the

requirements for schedules.

The paper then introduces a new model, based on the data-flow paradigm, which allows a schedule to be represented in the model. The schedules used are cyclo-static [1]. These schedules are characterized by processors being scheduled to nodes in a cycle so that each processor periodically revisits the nodes in its cycle. Once the schedule is included in the model, the resulting graph can be implemented on dynamic scheduling systems with full guarantee of achieving hard deadlines, even in the presence of time varying tasks.

The model is then extended to heterogeneous processor systems. This extension allows the same deterministic analysis of system performance. In this manner different schedules can be compared for their ability to meet deadlines. Finally the system is applied to an example of a space surveillance algorithm [9] to demonstrate the analytical ability of the model.

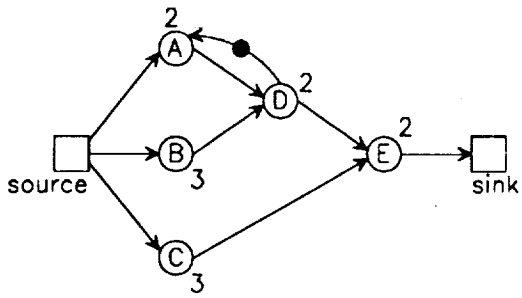

Figure 1. Data-flow graph (DFG)

Data-flow Model

The ensuing discussion is based on a data-flow model which employs a data-flow graph (DFG) [10]. Nodes in the graph represent tasks or modules of code. Each module has a maximum execution time, but times may vary to smaller values. Arcs in the graph represent data dependencies between the code modules which must be satisfied prior to a module executing. Tokens on an arc represent the communication of data from one module to another. The graph plays under the rule of earliest fire - as soon as a node is enabled by tokens on all incoming arcs, it fires. Figure 1 shows an example DFG with node times indicated next to each node.

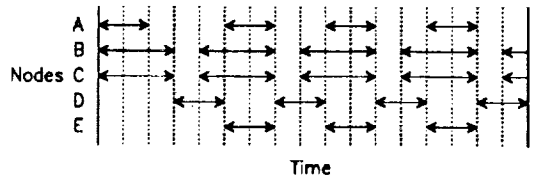

Figure 2. Graph play.

Execution of the graph is initiated by source nodes injecting data into the graph. The graph operates with repeated inputs which are periodic in nature. Data is consumed by sink nodes to signify completion of a computation. The rate of information flow through the graph is managed by injection control at the source. The injection rate is set to guarantee graph performance. The expected graph play if each node requires its maximum time is given in Figure 2 for the DFG in Figure 1.

The goal of the graph is to guarantee hard real-time deadlines. These guarantees are made based on steady state graph play. Steady state is defined by operation with maximum node times under repeated inputs. Any deviation in node times should result in no worse than steady state completion times for nodes.

This section defines the system performance and the associated steady state graph play. Then the requirements a cyclo-static

schedule must meet to satisfy steady state play are defined. This establishes the data-flow model on which current work is based.

Performance

Performance is based on the assumption that there are sufficient processor resources to execute the graph in steady state. Performance is determined by steady state graph play and is characterized by the throughput and latency [1,9]. There is no guarantee that the resource requirement is a minimum bound, but it is sufficient to play the graph.

Throughput is bounded by the length of the critical circuit where the length of a circuit is the time per token in the circuit. The critical circuit length (CL_{cr}) is defined by

$$CL_{cr} = \max\left(\frac{NT_i}{IT_i}\right) \quad \text{for } i=1...C \quad (1)$$

for a graph with C circuits, where NT_i is the total node time in circuit i and IT_i is the number of tokens in circuit i. Throughput is then bound by

$$TP \leq \frac{1}{CL_{cr}} \quad (2)$$

The maximum rate that data can be injected into the graph is defined in terms of the time between inputs (*TBI*) which is equivalent to CL_{cr}.

Latency is bounded by the length of the critical path, or the longest path from source to sink. Latency (L) is then defined as

$$L \geq \max(PL_i) \quad \text{for } i=1..P \quad (3)$$

for P paths from source to sink, where PL_i is the length of path i.

With sufficient resources, throughput can be maximized, and latency minimized, for a given graph. The number of resources (R) is bounded to meet performance requirements and can be found as the number required to meet steady state requirements as defined later.

Steady State Graph Play

The purpose of the proposed model is to predict performance of DFGs operating in steady state with repeated inputs and where the assigned node times are worst case values. Steady state graph play is completely characterized by the node firings which occur in one time period of length TBI [9]. This is demonstrated by a Gantt chart termed the Total Graph Play (TGP) diagram.

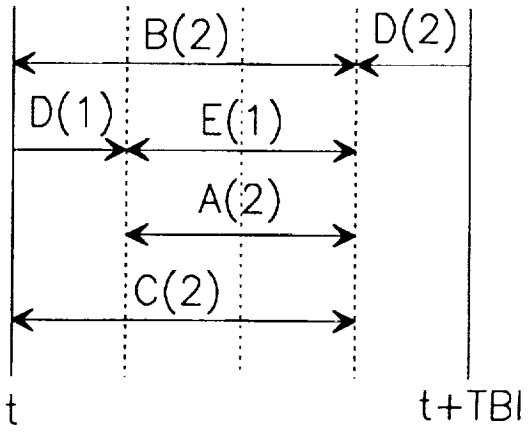

Figure 3. Total Graph Play.

The TGP diagram is constructed by drawing a Gantt chart of length TBI, with time ranging from t to $t+TBI$. The nodes are inserted into the TGP diagram using their worst case values. The nodes in the critical path are placed in the TGP diagram first. Node firing times are wrapped around from time $t+TBI$ to time t as needed to indicate a change in data packet number. Then the remaining graph nodes are placed in the

TGP based on their precedence relationship with existing nodes. Each node in the TGP diagram is associated with an iteration number which identifies the input data packet that produced a firing of the node. The iteration number is decremented by one during a wrap around the TGP diagram. This is a result of the times t and $t+TBI$ being equivalent in steady state, but for different input data packets. Thus all node firings in a TGP diagram are not necessarily associated with the same input. The TGP diagram for the graph in Figure 1 is shown in Figure 3.

Steady State Scheduling Requirements

The goal of steady state scheduling is to provide a schedule which will guarantee graph play with performance no worse than steady state play. This requires that processors be available to graph nodes no later than worst case steady state graph play dictates. It is not the purpose of this paper to determine a schedule, but rather to model and analyze schedules. However, some important criteria which schedules must meet are presented.

The first criterion for a schedule concerns the use of processor idle time. Different schedules cause different distributions of processor idle time. However, all schedules must meet the constraints established by the TGP diagram and data-flow graph precedence relations. Cyclo-static schedules can be represented by circuits of nodes over which processors traverse during the play of the graph. There may be one or more circuits for a given graph and schedule, and each processor may reside on only one circuit. As a processor traverses from one node to another on this circuit, it accumulates a non-negative idle time while waiting for the next node firing. Each transition appears in the TGP diagram once since each node fires exactly once in the interval TBI. Thus, the total time processors spend in graph play during the interval of TGP diagram is the sum of the total computing effort (TCE), which is the sum of all node times, and the idle time (T_{idle}). This must equal the total computing time available in the TGP diagram which is given by the number of available processors (R) times TBI resulting in the requirement

$$R \cdot TBI = TCE + T_{idle}. \qquad (4)$$

This equation represents a necessary requirement for a schedule, where the schedule determines the value for T_{idle}.

An example schedule for the TGP diagram in Figure 3 consists of two processor circuits, one containing nodes A, D, E, C and the other containing only node B. The idle times that processors spend between the node pairs are: node A to node D, 0; node D to node E, 0; node E to node C, 1; node C to node A, 2; and node B to node B, 1. Thus there are 4 processors, TBI is 4, TCE is 12, and T_{idle} is 4. This results in Equation 4 taking the form 4*4 = 12+4, which meets the requirements.

Graph Markings

The graph markings (token-arc pairings) are necessary for the following work. First a set of steady state markings are found. Then the initial graph markings are determined.

For steady state markings, the approach is to find a set of markings which satisfy the TGP diagram. The original graph is also needed, including the tokens necessary to meet its initial conditions. A timing point is then established in the TGP diagram (time t). For convenience, t is taken as that point in the TGP diagram when the source fires. Then the set of nodes with activity at time t^+ are examined. Two conditions arise: a node fires at time t, or a node was already

executing at time t^- and continues at time t^+. A final condition is handled at time $(t+TBI)^-$ where if a node completes before time $t+TBI$ but the ensuing node on a outgoing arc does not fire before time $t+TBI$, then a token must be placed on the outgoing arc. The conditions are handled by the following procedure:

A) All nodes firing at time t must have all tokens available to fire. Therefore, tokens are placed on all incoming edges for these nodes. If a token already existed on an edge from the original graph, an additional token should not be added.
B) All nodes currently executing are considered as going short and completed at time t^-. When a node finishes, it deposits tokens on all of its outgoing edges. Therefore appropriate tokens are placed to indicate the completion of each node in this category.
C) For all nodes in the graph, check all nodes associated with outgoing arcs. If these nodes do not fire before $t+TBI$ in the TGP diagram, then place a token on the arc between the two nodes.

When performed on the DFG in Figure 1, two new tokens are added: between nodes D and E, and between nodes C and E.

Given the markings for the graph in steady state, a set of initial markings are determined to allow the graph to initiate play and achieve steady state. The purpose for finding the initial markings is to remove some tokens from the graph to simplify analysis. These markings are found by advancing the steady state markings as far as they will go without injecting new inputs. This is a marking coinciding with the graph being played in steady state and then having inputs stopped and allowing the graph to come to rest. This all happens at some time less than zero. Then at time 0, new inputs are injected initiating graph play. The transient from allowing the graph to execute from steady state with no inputs and the transient from initiating graph play compliment each other allowing the graph to return to steady state. The resulting initial markings for the graph in Figure 1 match the initial markings provided.

Processor Schedule Model

A model is presented which explicitly incorporates processor scheduling in the data-flow context. The model is intended for use with hard real-time systems where meeting deadlines is the primary concern. The model is developed by utilizing the features of cyclo-static scheduling [1] in the data-flow graph such that the graph will play by the rules of the schedule in the presence of time varying nodes and dynamic schedulers.

The model is first presented for a homogeneous processor system and then later extended to heterogeneous systems. The homogeneous system demonstrates the new model which includes resource scheduling in the data-flow graph. It will also provide the mechanism for development of the necessary tools for heterogeneous systems.

Resource/Data-Flow Graph (RDFG)

A new data-flow graph model, called the resource/data flow graph (RDFG), is introduced which explicitly shows the relationship between resources and nodes, . In this paper, the resources are considered to be processors, although the model is extendable to other resources such as communication channels. The RDFG introduces new arcs in the DFG to describe how processors migrate through the graph nodes in a cyclo-static schedule. Processors

are represented as tokens in the graph, and thus are treated similarly to data in the evaluation of the graph.

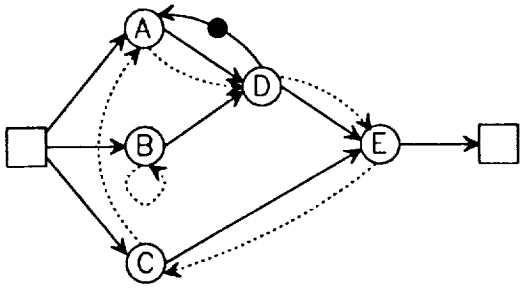

Figure 4. Resource/Data Flow Graph (RDFG).

The RDFG is developed from the DFG and a given cyclo-static schedule. A cyclo-static schedule may be represented as a set of circuits, called processor circuits, describing the order in which processors execute nodes. Processor circuits are formed by adding arcs, called processor arcs, to the original DFG so that a token representing a processor, when placed in this circuit, visits in order all nodes assigned to the processor. Each circuit may have multiple processors, but each processor can reside on only one circuit. The resulting circuits are disjoint and encompass all nodes. An example RDFG for the cyclo-static schedule developed earlier and for the graph in Figure 1 is illustrated in Figure 4.

Given a RDFG, finding a set of steady state markings is the next step in the analysis. This is done by the same process as for the DFG. These markings introduce the necessary processor tokens representing processors onto each processor circuit. A fully static schedule is represented as an RDFG where all processor circuits have only one token. Figure 5 is a possible set of steady state markings imposed on Figure 4. The initial graph markings are shown in Figure 6.

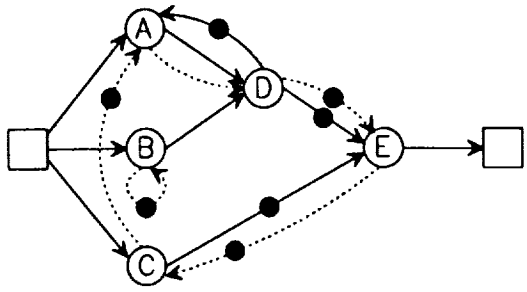

Figure 5. Steady state graph markings.

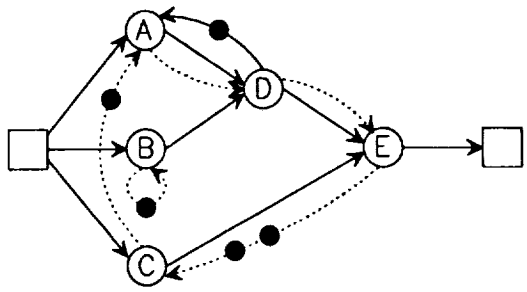

Figure 6. Initial graph markings.

Graph Reduction

The addition of processor arcs to the DFG may create redundant arcs in the RDFG in terms of necessary precedence relationships. It is helpful to delete redundant arcs to reduce the number of paths and circuits which must be considered during analysis and to better allow observance of graph activities.

The determination of which arcs are redundant is based on node precedence. An arc from node *a* to node *b* is redundant (and thus can be removed) if an alternative path exists from *a* to *b*. The alternate path may contain several arcs (and thus have stronger precedence relationships) or may be a single arc. If a single arc, the two arcs are equivalent, and thus either arc may be removed. The following rule may be used to eliminate one of the arcs. If the two arcs are of the same type (data or

control/processor), then randomly select one for elimination. If they are different, then eliminate the control arc since the data arc has more meaning in data-flow, maintaining the original graph in the RDFG, while the processor arc is a means only to scheduling. Figure 7 was reduced in preparation for analysis.

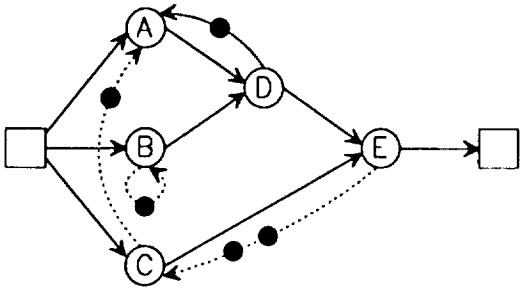

Figure 7. Reduced RDFG.

Unfolded Graph

A third graph is introduced to allow improved observation of individual node and processor behavior which is particularly useful in the heterogeneous system discussion. This graph is an unfolded graph such that all processor-node combinations are presented. The object is to create an equivalent graph having only one processor token per processor circuit. This means that each node in the unfolded graph is executed by only one processor which allows the examination of the relationship of a node to a specific processor. The foundation of the unfolded graph is found in [2] where the purpose was to transform a data-flow graph to allow fully static scheduling. The difference between their work and the current work is the RDFG already has a schedule imposed which may reduce the necessary unfoldings.

The basic method of developing the unfolded graph is to replicate the graph k times, where k is the greatest common multiple of the number of processor tokens on each processor circuit. Arcs with initial tokens are moved to reflect the data dependencies between iterations. The source of the arc remains intact, but the destination is moved from the node in iteration i to the corresponding node in iteration $(i+n) \bmod k$ for n initial tokens on the arc. The modulus function results in iteration $k-1$ cycling back to iteration 0 for repeated inputs. Figure 8 is a k-unfolded RDFG $(k=3)$ for the graph shown in Figure 7.

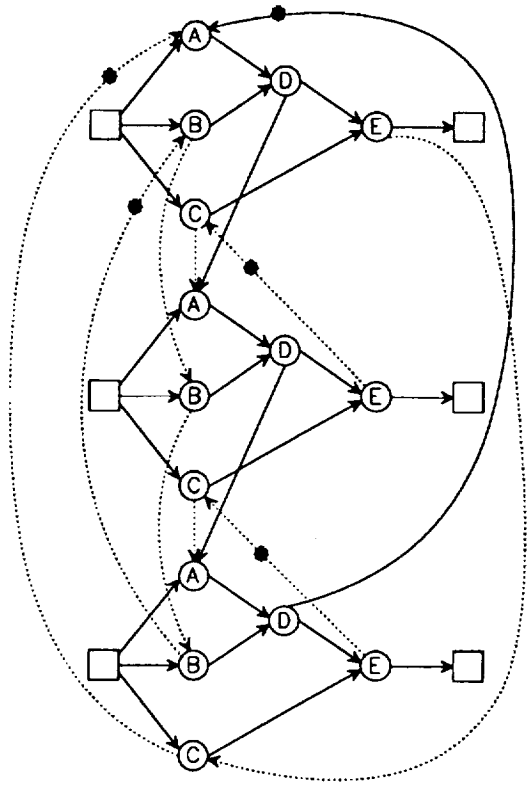

Figure 8. Unfolded RDFG (k = 3).

Hard Deadline Guarantee

It is a well known phenomenon that small reductions in the execution time of a node may adversely affect performance, thus

preventing the system from meeting deadlines. The reason performance may degrade is that the graph may enter resource saturation where there are more enabled nodes than processors and some work must be delayed beyond the worst case start time. Much work has been done to address this problem [11,12].

Static scheduling inherently avoids resource saturation and thus guarantees worst case performance. The new model extends this property to dynamically scheduled systems. The RDFG is a graph that plays on a dynamic system just as the equivalent DFG would play on a static system. However, the RDFG has more restrictive precedent constraints as a result of the processor arcs. The processor arcs guarantee that a processor is available for a node to fire without dictating which processor is used. The static schedule simply makes a direct association between processors and tokens on processor edges which is not necessary in dynamic scheduling.

Heterogeneous Processor Systems

The purpose of this section is to demonstrate the utility of the unfolded RDFG in analyzing heterogeneous systems. The construction of the unfolded RDFG depends on the specification of a processor schedule. The scheduling approach used for this paper is to assume a worst case processor schedule. It is known that this precludes the representation of certain valid heterogeneous schedules, but developing heterogeneous schedules directly for the RDFG is left for future work.

The method used addresses the standard practice often used for heterogeneous systems where a node is scheduled exclusively on one class of processors. In the RDFG, such a schedule appears as multiple processor circuits, where each such circuit is assigned a single class of processor. For the schedule used in the previous section in Figure 7, node B would execute on one class of processor and the other nodes would execute on a second class of processor. This is a special case of the systems under consideration.

This section first demonstrates the benefits of using the unfolded RDFG. Then, the analysis of the graph to determine available throughput and latency improvement for a given schedule is presented. The effect of different processor assignments to the RDFG is shown followed by a discussion of heuristic approaches for improving performance for a given schedule operating in a heterogeneous environment.

Unfolded RDFG

The unfolded RDFG is particularly useful for evaluating heterogeneous systems. The reason for this is the fully static nature of the RDFG which allows each node to be mapped to a single processor. Thus, the effect of a processor on the node processing time can be characterized and included in the analysis.

For example, suppose that two of the four processors in the unfolded RDFG in Figure 8 are capable of executing nodes one time unit faster than the time indicated in each node. Then the node times in the processor circuits can be adjusted to reflect the capabilities of the specific processor assigned to the circuit. Figure 9 illustrates an unfolded RDFG for a given processor assignment.

Performance Characteristics

Once the unfolded RDFG is found for a given schedule and processor assignment, the system performance can be computed. The methods of finding throughput and latency in

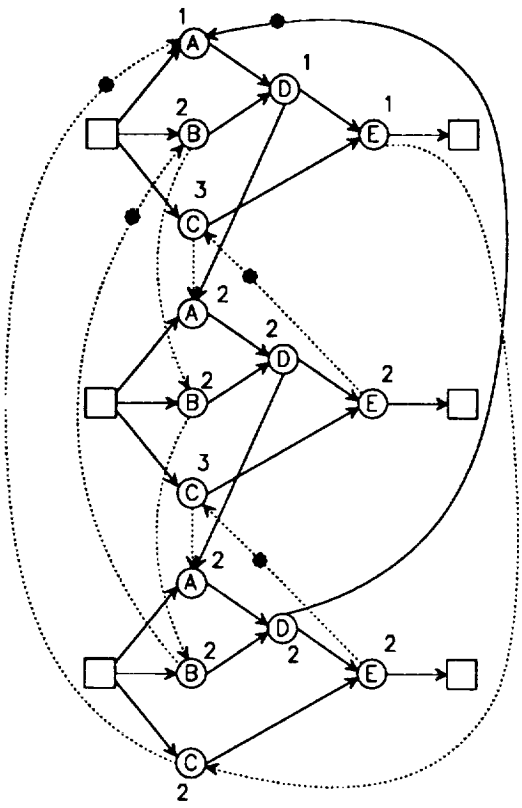

Figure 9. Unfolded RDFG with initial tokens and processor assignment.

the unfolded graph are similar to those used in the original DFG. Performance is still limited by the critical path and critical circuit for the given graph.

Throughput: The throughput for the heterogeneous system is found by determining the critical circuit in the unfolded RDFG. However, this critical circuit defines the throughput for k inputs given a k-unfolded RDFG. Thus the lower bound on throughput (TP_{LB}) is

$$TP_{LB} = \frac{k}{CL_{cr}(unfolded\ RDFG)} \quad (5)$$

though the graph as defined may not be able to play with periodic inputs at this throughput.

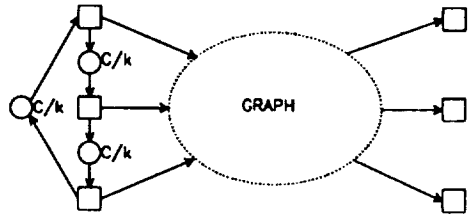

Figure 10. Injection control of an unfolded RDFG.

To reestablish periodic inputs, injection control is imposed on the k sources. The sources are placed in a circuit of length C with k nodes of time C/k between each source firing, as shown in Figure 10. In this manner, each source fires periodically every C time units. Provided $C/k \geq TP_{LB}$, C/k will dictate the actual throughput. This is possible since the addition of the circuit at the set of sources creates no other circuits. Injection control allows a throughput to be defined with periodic inputs by imposing a critical circuit about the sources.

The critical circuit for the graph in Figure 9 has a length of 10. Therefore, each source can fire every 10 time units, and the 3 sources can be made periodic by firing them in sequence separated by $\geq 10/3$ time units.

Latency: Prior to controlling the sources, latencies can be computed from each source to its corresponding sink (L_i for iteration i). The lower bound latency for the given schedule and assignment is

$$L_{LB} = \max(L_i) \quad for\ i=1...k. \quad (6)$$

Imposing periodic inputs may have adverse effects on latency, possibly even increasing latency above L_{WC}. This occurs because some sources may fire before the graph is ready to accept them. Thus, tokens will wait at some point in the critical path,, increasing the total time spent on the critical path.

To compute the periodic latency (L_p), the aperiodic and periodic firing sequences for sources must be compared. The issue is whether a periodic source fires earlier or later than its corresponding aperiodic source. Firing later is not an issue since the source then lags behind the graph, and thus will not add any time to the latency. Firing early means the graph will have to catch up adding the amount the source fired early to the latency (EF where $EF = 0$ if the source fires late). The resulting latency is

$$L_p = \max(L_i+EF_i) \quad for\ i=1...k \quad (7)$$

for the graph. Note that the latency can be improved up to L_{LB} by increasing the throughput which reduces EF_i. Therefore, a trade off between throughput and latency is available for improving performance.

For the graph in Figure 9, the three iterations have latencies of 4, 6, and 6 respectively prior to injection control at the inputs. If the inputs are fired periodically every 10/3 time units, then iteration 0 fires on time, iteration 1 fires 1 1/3 time units after the earliest it could, and iteration 2 fires 1/3 time units early. Therefore, the latencies remain the same under periodic inputs.

Processor Assignment

An alternative to decreasing throughput to preserve latency is to consider other processor assignments. A one to one assignment of processors to processor arcs is made for a given schedule. Different assignments produce different performance characteristics. Therefore, the performance obtained by various assignments should be considered.

The different assignments change performance in several ways. Assignments potentially affect latency on different iterations, the critical circuit of the unfolded RDFG, and the idle time spent on the critical path on different iterations. The combination of these three factors may result in many different performance points.

The various assignments should be compared to determine the appropriate choice. Not all markings need to be considered since many potential markings can be proven equivalent. The appropriate assignment will then be based on the desired goal of the implementation, either to improve throughput or latency.

Schedule Potential

An attempt is made to provide insight to the ability for a given schedule to have improved performance over other potential schedules for a given graph. A schedule ideally should allow nodes to have lower execution times without providing further graph constraints beyond those present in the original DFG. Different schedules change node times in heterogeneous systems, and evaluating how different schedules improve performance requires knowledge of the peformance goals for a particular heterogeneous system. But the ability of a schedule to avoid imposing further graph constraints can be characterized.

Ideally, processor arcs in the unfolded RDFG should have little influence on throughput improvement. If the destination node of a processor arc is allowed to fire

early by its incoming data arcs, the processor arc may delay the node until a processor is available. In this manner the processor arc imposes further constraints upon the graph. This restricts how much faster the graph runs over and above the data-flow requirements. Thus schedules which result in more idle time on processor arcs are desirable.

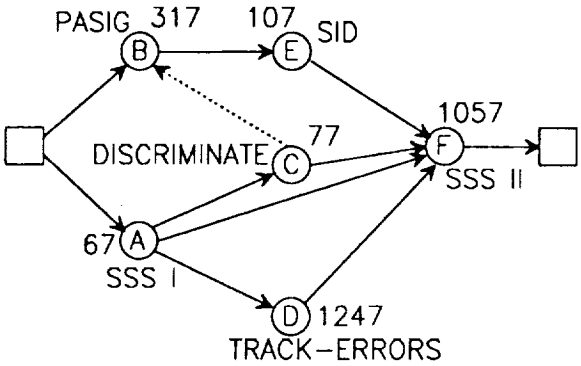

Figure 11. DFG for a space surveillance algorithm.

A Space Application

An example algorithm is presented as a test case for the model. The algorithm is a space surveillance algorithm [9]. Figure 11 illustrates the DFG for the algorithm. At each iteration, the algorithm accepts as inputs the current position coordinates of multiple targets located in the instrument's field of view. The algorithm identifies each target and plots the target's trajectory in three dimensional space. Node labels describe the algorithm operations, and the relative time required for each operation is shown beside the node. Note that a control edge has been added to the graph. The extra control edge reduces the number of required processors from 4 to 3 on a homogeneous processor system.

The performance measures for the DFG are TBI = 1247 and latency = 2371 with 3 processors. Suppose that one of the processors used was capable of executing nodes at 75% of the specified time. The result of scheduling this processor is presented.

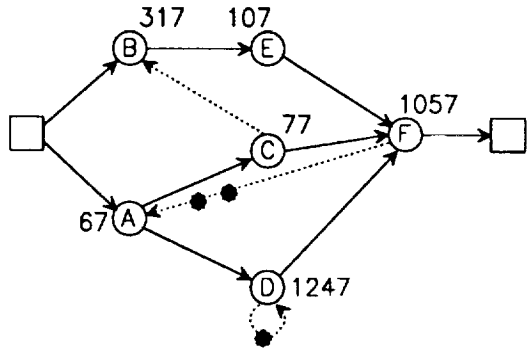

Figure 12. RDFG for a space surveillance algorithm.

A possible cyclo-static schedule for the space surveillance algorithm is for one processor to execute node D and the other processors to cycle through the other nodes in a circuit consisting of nodes A-C-B-E-F in the specified order. The resulting RDFG is shown in Figure 12 and the unfolded RDFG in Figure 13. When the faster processor is applied to the D node, each D node will execute in 936 time units. This results in a new TBI of 1057 and a new latency of 2060. The faster processor cannot lower TBI further since the slow processors handling node F now dictate the throughput.

Conclusion

The RDFG is a graph model which allows schedule development with data-flow constructs on parallel computing systems. The RDFG also guarantees that hard real-time deadlines are met. The graph will meet deadlines even when executed on a system with a dynamic scheduling scheme such as a queue.

An extended RDFG, termed the unfoled RDFG, allows addressing heterogeneous

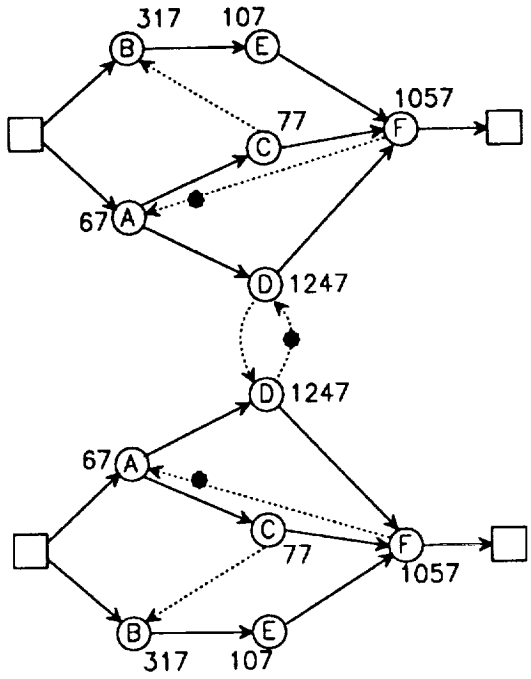

Figure 13. Unfolded RDFG for a space surveillance algorithm.

processor systems. It is a fully static form of the RDFG where each node executes on one processor. Then each node time can be adjusted to describe the execution on a specific processor for a given processor assignment. The graph is then analyzed to find the available performance in terms of throughput and latency.

The unfolded RDFG was demonstrated on an example problem for heterogeneous processors. The space surveillance algorithm is an example of a control system with repeated inputs for which the model is well suited. The unfolded RDFG provided the ability to analyze the performance in terms of throughput and latency.

References

[1] D. A. Schwartz, T. P. Barnwell, III, "Cyclo-static multiprocessor scheduling for the optimal implementation of shift invariant flow graphs," *Proc. ICASSP-85*, Tampa, FL, Mar. 1985.

[2] K. K. Parhi, D. G. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," *IEEE Transactions on Computers*, v. 40, n. 2, pp. 178-195, Feb. 1991.

[3] B. Shirazi, M. Wang, G. Pathak, "Analysis and evaluation of heuristic methods for static task scheduling," *Journal of Parallel and Distributed Computing*, v. 10, pp. 222-232, 1990.

[4] S. M. Heemstra de Groot, S. H. Gerez, O. E. Herrmann, "Range-chart-guided iterative data-flow graph scheduling," *IEEE Transactions on Circuits and Systems-I*, v. 39, n. 5, pp. 351-364, May 1992.

[5] M. Marrakchi, "Optimal parallel scheduling for the 2-steps graph with constant task cost," *Parallel Computing*, North-Holland, v. 18, pp. 169-176, 1992.

[6] S. Ha, E. A. Lee, "Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration," *IEEE Transactions on Computers*, v. C-40, pp. 1225-1238, Nov. 1991.

[7] G. C. Sih, E. A. Lee, "A Compile-Time Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Transactions on Parallel and Distributed Systems*, v. 4, No. 2, pp. 175-, February 1993.

[8] Y. Hu, Z. Xie, X. Lu, "Approaches to Decentralized Control of Job Scheduling for Homogeneous and Heterogeneous Parallel Computing Systems," *Future Generation Computer Systems*, North-Holland, v. 6, pp. 91-96, 1990.

[9] S. Som, R. R. Mielke, J. W.

Stoughton, "Prediction of performance and processor requirements in real-time data flow architectures," *IEEE Transactions on Parallel and Distributed Systems*, v. 4, n. 11, Nov. 1993.

[10] K. M. Kavi, B. P. Buckles, "A formal definition of data flow graph models," *IEEE Transactions on Computers*, v. C-35, pp. 940-948, Nov. 1986.

[11] G. K. Manacher, "Production and stabilization of real-time task schedules," *Journal of the Association for Computing Machinery*, v. 14, n. 3, pp. 439-465, July 1967.

[12] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, v. 17, n. 2, pp. 416-429, March 1969.