A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures

Gilbert C. Sih, *Member, IEEE*, and Edward A. Lee, *Member, IEEE*

***Abstract*— This paper presents a compile-time scheduling heuristic called *dynamic level scheduling,* which accounts for interprocessor communication overhead when mapping precedence-constrained, communicating tasks onto heterogeneous processor architectures with limited or possibly irregular interconnection structures. This technique uses dynamically-changing priorities to match tasks with processors at each step, and schedules over both spatial and temporal dimensions to eliminate shared resource contention. This method is fast, flexible, widely targetable, and displays promising performance.**

***Index Terms*—Compile-time scheduling, dynamic level, heterogeneous processors, interconnection constraints, interprocessor communication, parallel processing.**

I. Introduction

As parallel processing architectures evolve, designers have increasingly emphasized the importance of dedicated hardware support for interprocessor communication (IPC). IPC has become the major performance limitation in parallel processing environments, due to the transmission/synchronization delays and conflicts for shared communication resources created by data exchange. Specialized hardware devices, designed to permit high throughput, low latency communications, are essential for efficient parallel hardware utilization [1], [2]. Parallel architectures that mix different types of processing devices have also become more widespread. This trend is especially evident in digital signal processing, where the extreme processing rates demanded by applications such as high-definition television, medical imaging, or seismic and weather prediction often require a mixture of general purpose processors, programmable digital signal processors, and application specific integrated circuits (ASIC's). The efficient exploitation of parallelism in such environments requires effective partitioning and scheduling strategies that account for IPC overheads, and consider both algorithmic and architectural characteristics to achieve an effective mapping of tasks to processors.

We have developed a nonpreemptive, compile-time scheduling heuristic that maps directed, acyclic precedence graphs onto architectures that may contain heterogeneous processing devices interconnected in an irregular fashion. This technique, called *dynamic-level scheduling,* accounts for interprocessor communication overheads, and eliminates shared communication resource contention by incorporating information about the processor interconnection topology. The input to the scheduling algorithm is an acyclic precedence expansion graph (APEG) $G = \{N, A\}$, where $N = \{N_i: i = 1, \cdots, n\}$ is a set of nodes (tasks) which represent program computations, and A is the set of directed arcs $\{A_{ij}\}$ which represent both precedence constraints and data paths. Each arc A_{ij} carries label D_{ij} which specifies the amount of data (in bits, bytes, or words) that N_i passes to N_j on each invocation. These graphs may be derived from data flow graph algorithmic descriptions which fit the Synchronous Data Flow (SDF) model [3]. This description naturally exposes inherent internode parallelism in the algorithm and allows partitioning, scheduling, and insertion of synchronization primitives to be performed at compile-time, avoiding much run-time overhead. Under a *fully-static* scheduling paradigm, the class of applications effectively expressed using this programming model is limited. Constructs such as conditionals and data-dependent iteration must be excluded to provide deterministic program behavior, making the application domain most suitable for signal processing algorithms and some scientific computations. Under a *self-timed* scheduling paradigm, several of these constraints can be relaxed, permitting the class of relevant applications to be broadened. See [4] for a discussion of these issues.

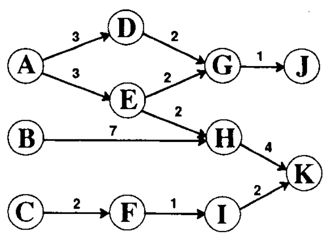
The target architecture, which contains a set of potentially heterogeneous processors $P = \{P_k: k = 1, \cdots, p\}$, is assumed to contain dedicated communication hardware so that computation can be overlapped with communication. We assume that a fairly accurate estimate of the execution time of node N_i on processor P_j, $E(N_i, P_j)$, is available at compile-time for each node-processor pair. This assumption is valid for most signal processing algorithms, due to the deterministic nature of the computations. If node N_i cannot be executed on processor P_j the execution time $E(N_i, P_j)$ is infinite. An example APEG and its associated node execution time table are shown below in Fig. 1. For scheduling purposes, we assume that each node is indivisible; no attempt will be made to use intranode parallelism.

Our scheduling objective is to minimize the *schedule length,* or *makespan,* where all interprocessor communication overheads are included. This goal equivalently maximizes the

Manuscript received May 23, 1991; revised September 30, 1991. This work was supported by SRC, Cygnet, Dolby Laboratories and the State of California Micro program.

G. C. Sih is with QUALCOMM Inc., San Diego, CA 92121.
E. A. Lee is with the Electrical Engineering and Computer Science Department, University of California, Berkeley, CA 94720.

IEEE Log Number 9204625.



Execution Times

	P0	P1
A	3	5
B	5	6
C	2	4
D	4	4
E	5	3
F	2	3
G	4	∞
H	5	6
I	4	∞
J	3	3
K	4	6

Fig. 1. An APEG and its node execution time table.

speedup, defined as the shortest time required for sequential execution of the APEG on a single processor, divided by the time required for parallel execution on multiple processors. This scheduling problem is NP-complete in the strong sense, even if there are an infinite number of processors available [5]. Hence we will rely upon heuristics.

Many related works have concentrated on task allocation, attempting to assign tasks to processors in order to achieve some objective, such as load balancing, minimization of IPC, or some combination of the two [6]–[10]. Although these works are innovative, they either ignore precedence constraints or attempt to minimize an objective other than schedule length, and thus are not applicable in this context.

A relevant approach, called *linear clustering,* has been proposed by Kim and Browne [11]. This technique decomposes the graph into a set of linear clusters, where a linear cluster is defined as a degenerate tree in which every node contains exactly one immediate predecessor and one immediate successor. At each step, the algorithm clusters together the most expensive directed path (in computation and communication) into a single linear cluster. The clustered nodes are removed from the graph and this procedure is repeated until the entire graph has been divided into clusters. After some refinement steps, the clusters are mapped onto the specified architecture using graph theoretic techniques.

The *internalization* approach, proposed by Sarkar [5], clusters nodes together to minimize the schedule length on an unbounded number of processors. The algorithm initially places each task in a separate cluster and considers the arcs in descending order according to the amount of data transferred over each arc. Given arc A_{ij} connecting nodes N_i and N_j, the algorithm merges the clusters containing these nodes ($C(N_i)$ and $C(N_j)$) to "internalize" any communications between nodes in these respective clusters. This merging step is accepted if it does not increase an estimate of the parallel execution time of the graph on an infinite number of processors, where nodes in the same cluster are constrained to be executed on the same processor. After exhausting the list of arcs, a processor assignment phase maps the clusters to the physical processors using a modified list scheduling approach.

The dynamic-level scheduler was developed as part of an interactive design system for digital signal processing (DSP) called Gabriel [12] which allows rapid prototyping of new DSP algorithms using a block-diagram oriented graphical interface. The blocks range in granularity from simple operators such as adders or multipliers, to basic functions such as FFT's or FIR filters, to complex signal processing subsystems such as a speech coder. Using feedback information from the scheduler, an algorithm designer can iteratively refine both task graph and target architecture to maximize performance. This environment imposes several stringent constraints on the permissible scheduling strategies. First, the interactive nature of the design approach requires that the scheduling technique execute rapidly. Since this scheduling problem involves exponential complexity (unless P = NP), optimal solution strategies must be sacrificed in favor of fast heuristic techniques. Second, the scheduling technique must be flexible enough to handle mixed-granularity task graphs. Third, the scheduling scheme must be adaptable to the diverse architectures encountered in digital signal processing. The dynamic level scheduling algorithm is a promising approach toward meeting these goals.

This paper is organized as follows: Section II discusses IPC issues, Section III introduces the dynamic level scheduling strategy for homogeneous processors, and Section IV presents methods of streamlining this algorithm. Section V compares the performance of the DLS algorithm with two other scheduling strategies that account for IPC. Section VI extends the algorithm to the heterogeneous processor case, and Section VII concludes this discussion.

II. Interprocessor Communication

If interprocessor communication can be obtained for free, all available task parallelism can be exploited without cost. That is, given enough processors, an optimal schedule can always be constructed by invoking all simultaneously executable nodes on different processors. The reality of nonzero IPC cost induces a tradeoff between the amount of parallelism utilized and the amount of communication overhead incurred. Addressing this tradeoff requires consideration of the task grain size, the amount of parallelism in the graph, the number of processors, the processor interconnection topology, the transmission and synchronization delay overheads, and the possibility of communication resource contention.

Scheduling in the presence of IPC contains two main aspects: assigning processors to computation nodes, and allocating communication resources for interprocessor data transfers. These two problems, often referred to as the "mapping problem" and the "traffic scheduling" problem respectively, have traditionally been dealt with separately. A task allocation algorithm first assigns nodes to processors in accordance with some objective function, and then a routing algorithm performs interprocessor traffic scheduling upon the specified node mapping [8], [13]. This separation is impractical, because the ease with which the traffic scheduling can be performed is directly dependent on the properties of the mapping; the best isolated assignment of nodes to processors is invariably suboptimal after simultaneous consideration of both communications and computations.

By addressing both issues concurrently, our scheduling strategy averts overloaded communication resources by adjusting the node-processor mapping accordingly. Just as computations are scheduled upon processors, communications are scheduled upon IPC resources by dedicating the resources used in a data transfer for the duration of the transmission. With guarantee of resource availability, the communication time can be calculated deterministically (or upper bounded) using the locations of source and destination processors, the amount of data to be transferred, and the characteristics of the communication architecture. A routing algorithm, employed by the scheduler, uses knowledge of previous resource usage to reserve a path between source and destination processors for the duration of this time window.

For illustrative purposes, consider the scheduling of the APEG from Fig. 2 onto the target architecture shown in Fig. 3, which consists of four processors interconnected through four full-duplex interprocessor links. For simplicity, assume that the time needed to communicate D units of data between any two processors is merely D time units, regardless of the distance between them. The upper chart in Fig. 4 shows a possible scheduling of nodes onto processors, while the lower chart in Fig. 4 shows the corresponding scheduling of communications onto links. This simultaneous consideration of spatial (routing) and temporal (scheduling communication time windows) aspects of IPC eliminates the possibility of shared resource contention, which ensures deterministic behavior.

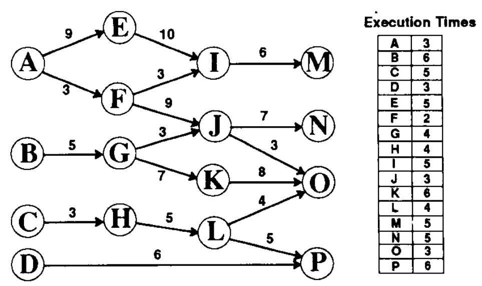
To permit wide retargetability without sacrificing efficiency, we divide the scheduling algorithm into two components. The first component contains the fixed, architecture-independent scheduling routines, while the second component contains the architecture-dependent communication resource scheduling and routing routines. This division permits us to use special-purpose routines which are optimized for a particular architecture within the second component. A specific interface is defined at the boundary, which allows the topology dependent sections to be interchangeable, as illustrated in Fig. 5. Given the interface specifications and a predetermined routing strategy, an average programmer can code the topology dependent portion for a new architecture within a few hours.

III. Dynamic Levels for Homogeneous Processors

We first discuss the case in which all processors are homogeneous. The dynamic level scheduling algorithm uses a variant of the classic list scheduling formulation in which nodes are assigned priorities and placed in a list, sorted in order of decreasing priority. Nodes that have all immediate predecessors executed are referred to as being ready (for execution). A global time clock regulates the scheduling process and processors that are idle at the current time are referred to as being available (for assignment). Whenever a processor is available, the algorithm assigns it the first ready node in the list for execution. After assignment, the node is deleted from the priority list, the processor is removed from the available processor list, and the algorithm moves to the next assignment. If the list of ready nodes is empty, or the set of available processors is exhausted, the global time clock is incremented until some processors finish execution of their assigned tasks and are available once again.

A. HLFET Scheduling

One popular list scheduling approach is the Highest Levels First with Estimated Times (HLFET) algorithm, an extension of Hu's classic work [14]. This technique assigns each node a level, or priority for scheduling. The level of node N_i is defined as the largest sum of execution times along any directed path from N_i to an endnode of the graph, over all endnodes of the graph. Since these levels remain constant for the entire scheduling duration, we refer to these levels as being "static," and denote the static level of node N_i as $SL(N_i)$. The list scheduling algorithm is then invoked using these priorities. Several studies have revealed that the HLFET algorithm demonstrates near-optimal performance when IPC costs are not included [15], [16]. The success of this scheme



A	3
B	6
C	5
D	3
E	5
F	2
G	4
H	4
I	5
J	3
K	6
L	4
M	5
N	5
O	3
P	6

Fig. 2. An example APEG.

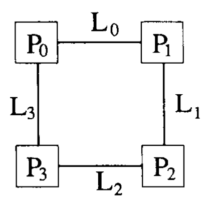


Fig. 3. A four-processor target architecture.

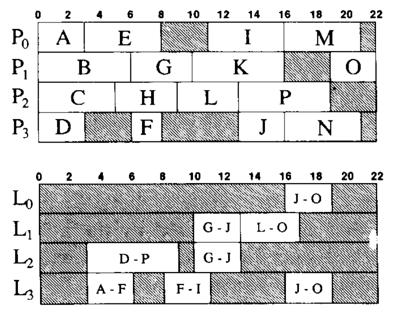


Fig. 4. Scheduling of nodes onto processors and communications onto links.

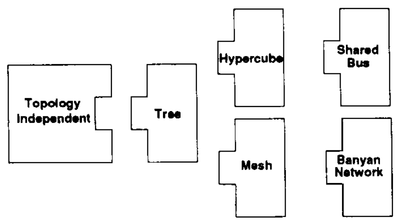


Fig. 5. The processor topology dependent portions are interchangeable.

stems from the accurate representation of a node's priority by its static level. By selecting the ready node with highest static level for placement at each step, the technique successively shortens the longest remaining path to completion.

Yu proposed a technique that extends the HLFET algorithm by including communication delay in the context of a fully-interconnected processor network [17]. His heuristic selects the ready node with the highest static level at each step, and schedules it on the available processor that completes execution of the node at the earliest time. Yu also proposed techniques that use combinatorial matching algorithms to pair ready nodes with available processors. Although these ideas are sound, the use of the traditional list scheduling method with global clock leads to an inherent flaw in operation that is exposed in Section III-C.

B. Dynamic Levels

At each scheduling step, a list scheduling algorithm performs two tasks. It selects the next node to schedule, and chooses the processor to execute the node on. The HLFET algorithm makes these selections independently at each step, causing poor performance in the presence of IPC. Whereas the static levels used in HLFET are fixed, we introduce a new quantity whose value changes throughout the scheduling process. This *dynamic level,* denoted $DL(N_i, P_j, \Sigma(t))$, reflects how well node N_i and processor P_j are matched at state $\Sigma(t)$, where $\Sigma(t)$ encompasses both the state of the processing resources (previously scheduled nodes), and the state of the communication resources (previously scheduled data transfers) at global time t. We first define $DA(N_i, P_j, \Sigma(t))$ to be the earliest time that all *data* required by node N_i is *available* at processor P_j at state $\Sigma(t)$. This quantity, calculated within the topology-dependent section of the scheduler, represents the earliest time at which all data transfers to node N_i from its immediate predecessors can be *guaranteed* to be completed with all communication resources reserved in advance. The dynamic level is then expressed as

$$DL(N_i, P_j, \Sigma(t)) = SL(N_i) - \max\left[t, DA(N_i, P_j, \Sigma(t))\right]. \quad (1)$$

At each step, the ready node and available processor that maximize this expression are chosen for scheduling. Note that the instantaneous dynamic level for the same node-processor pair will change in successive scheduling steps, based on the resource availability.

The dynamic level has a straightforward interpretation. The maximization term represents the earliest time that node N_i can start execution on processor P_j, because the node cannot start execution until the current time, and it cannot be invoked until all the data from its predecessors has been received. So the dynamic level $DL(N_i, P_j, \Sigma(t))$ is the difference between the static level of node N_i and the earliest time the node can start execution on processor P_j. This expression, which simultaneously incorporates both execution and communication time aspects, is intuitively appealing. When considering prospective nodes for scheduling, a node with large static level is desirable because it indicates a high priority for execution. When comparing candidate processors for assignment, a later starting time is undesirable because it indicates either a busy processor or a long interval required for interprocessor communication. Processors that incur a large amount of IPC are penalized by a later starting time, which lessens the dynamic level. The algorithm evaluates dynamic levels (which may be negative) over all combinations of ready nodes and available processors to find the best node-processor match for scheduling at the current state. We designate this approach the Highest *Dynamic* Levels First with Estimated Times (HDLFET) algorithm. This method uses the classic list scheduling formulation with dynamic levels.

To gain an indication of the performance improvement obtainable from replacing static levels with dynamic levels, we scheduled randomly generated task graphs containing between 50 and 250 nodes onto a 16-processor mesh. Node execution times and nearest-neighbor communication times were chosen randomly from the same uniform distribution. We investigated several options for node and processor selection. The first method initially selects the available processor with smallest index and then chooses the ready node that maximizes the dynamic level with this processor. The second method initially selects the ready node with highest static level and chooses the available processor maximizing the dynamic level with this node. The third method examines all possible combinations of ready nodes and available processors and chooses the node-processor pair maximizing the dynamic level. In each case, we compared the percentage improvement in speedup obtained over the HLFET approach using independent node and processor selection. Communication costs were included in all cases (including HLFET).

As shown in Fig. 6, an interesting set of curves emerges when the speedup improvement is compared against the graph parallelism, where graph parallelism is measured as

$$\left\lceil \frac{1}{\max\limits_{j} SL(N_j)} \sum_{i=1}^{n} E(N_i) \right\rceil. \quad (2)$$

This is a lower bound on the number of processors required to execute the graph in time bounded by the critical path (the longest path from any initial node to any terminal node) when IPC costs are not included. Each point in Fig. 6 represents the average percentage improvement in speedup over HLFET obtained for all the randomly-generated graphs with the specified amount of graph parallelism.

The initial processor selection technique exhibits little improvement when the amount of graph parallelism is small compared to the number of processors, but starts to gain improvement rapidly as the amount of parallelism increases. This phenomenon can be simply explained by examining the number of nodes ready for execution at each step. When the number of processors far exceeds the graph parallelism, there are few ready nodes at each scheduling step, often just a single node. By initially selecting the processor, the algorithm forces this single ready node to be executed on the processor, and therefore performs the same steps as the HLFET scheme using independent node and processor selection. As the amount of graph parallelism increases, the number of ready nodes at each

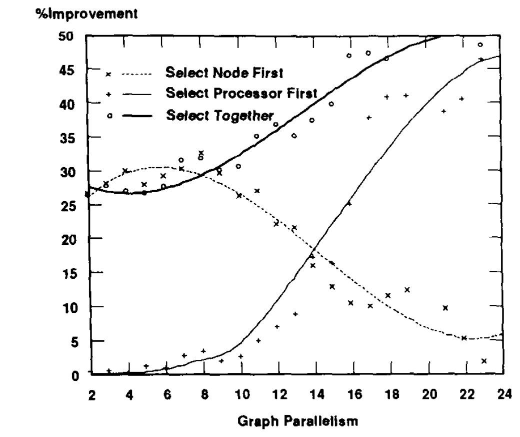


Fig. 6. Percent improvement in speedup of HDLFET over HLFET.

step increases, permitting a better match between nodes and processors.

Conversely, the initial node selection technique exhibits large improvement when the number of processors exceeds the amount of graph parallelism, but tapers off as the parallelism is further increased. This can be accounted for through consideration of the available processors at each step. When the number of processors far exceeds the graph parallelism, there is little contention for processors. Each node is able to select its "preferred" processor that incurs little communication overhead out of the available processor list. As the amount of parallelism increases, parallel paths in the graph must begin to share processing resources, and scheduling steps may have multiple ready nodes desiring a common processor. As processors are successively removed from the available processor list, the node with highest static level is often forced to be executed on one of the remaining available processors for which excessive IPC is incurred. The performance degradation is exacerbated as the amount of parallelism is further increased.

Selecting the highest dynamic level ready-node, available-processor pair out of all combinations retains good improvement throughout, increasing slightly as the amount of graph parallelism increases. This method does not incur the increased-parallelism performance degradation exhibited by the initial node selection method because it has more flexibility in choosing nodes. At any step, a node that does not have the highest static level may form the best match (in dynamic level) with an available processor, and therefore be selected for scheduling. Instead of forcing execution on unsuitable processors, this strategy allows a ready node to wait until a successive time step when its desired processor is again available. This method demonstrates superior performance over all the other techniques, generating speedup improvements of over 50% in comparison with the HLFET algorithm. However, this performance is attained at the price of added computational complexity.

C. Processor Selection Revision

Although the use of dynamic levels significantly improves performance, the HDLFET algorithm still exhibits the list

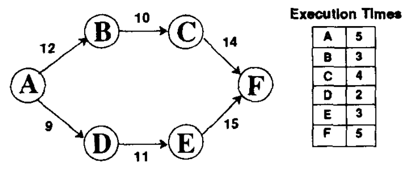


Fig. 7. A fine-grained precedence graph.

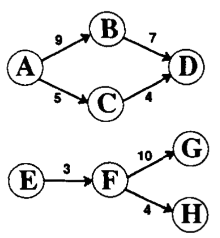
scheduling deficiency of being unable to idle "available" processors. Consider the graph shown in Fig. 7, and for simplicity assume a two-processor system with communication model $C = D$, so that the number of cycles needed for IPC equals the number of data units. The optimal schedule executes every node on a single processor while idling the other processor completely, a solution that is unattainable under the current list scheduling methodology. This example exposes an inherent flaw in the HDLFET algorithm: it restricts processor candidates for a ready node to the set of currently available processors. The global clock is updated to add additional available processors only if the available processors or ready nodes are exhausted. The original philosophy behind this processor selection mechanism, which attempts to exploit as much task parallelism as possible, is no longer valid in the presence of IPC and causes poor scheduling performance in a mixed-grain environment.

To remedy this difficulty, we alter the fundamental operation of the algorithm by removing the global timeclock used to update the current time at each scheduling step. There is no difficulty with causality because scheduling is performed at compile-time, not at run-time. Since processors are no longer classified as being "busy" or "available," *all* processors can be considered candidates for scheduling at each step. This allows the same processor to be chosen in consecutive scheduling steps, and permits some processors to have nodes scheduled far ahead of other processors. This revision necessitates a few changes in the dynamic level. We change the state of the processing and communication resources $\Sigma(t)$ to Σ, and let $TF(P_j, \Sigma)$ represent the time that the last node assigned to the jth processor finishes execution. We then redefine the dynamic level as

$$DL(N_i, P_j, \Sigma) = SL(N_i) - \max\,[DA(N_i, P_j, \Sigma), TF(P_j, \Sigma)]. \tag{3}$$

The revised algorithm, which operates without a global clock and uses the dynamic level shown in (3), is the Dynamic Level Scheduling (DLS) algorithm.

To illustrate the effect of the global clock removal more clearly, we contrast the scheduling steps taken by the HDLFET algorithm and the DLS algorithm using the APEG shown in Fig. 8. This example is scheduled onto a two-processor system where the processors are assumed to be interconnected by a full-duplex link. Both scheduling methods will select a node and processor simultaneously using dynamic levels at each scheduling step. As shown in Fig. 9, the scheduling steps taken by the two approaches start to diverge after nodes A and B have been scheduled on P0 and nodes E and F have been scheduled on P1. At this point, the HDLFET algorithm's global clock is at time 5, when P1 is the only processor



Execution Times

A	3
B	3
C	3
D	7
E	2
F	3
G	5
H	6

Static Levels

A	13
B	10
C	10
D	7
E	11
F	9
G	5
H	6

Fig. 8. An example APEG.

available for scheduling, and nodes C, G, and H are ready for execution. After evaluating dynamic levels for these three nodes with P1, the algorithm finds that node C forms the best match with P1. It therefore schedules node C on P1, schedules communication A to C on the link from P0 to P1 in the interval [3, 8], and updates the global clock to time 6, when P0 becomes available. After evaluating dynamic levels for nodes G and H with P0, the algorithm schedules node H on P0 and communication F to H on the link from P1 to P0 in the interval [5, 9]. The global clock is updated to time 11, freeing P1 for the scheduling of either node G or D. Although node D has higher static level than node G, it has a lower dynamic level with P1 at the current state. The input data from node B can not be guaranteed to be available until time 15, because the previous communication from node A to node C has reserved the link from P0 to P1 until time 8. Node G is therefore scheduled on P1. Finally, node D is scheduled on P0, with communication from C to D being scheduled in the interval [11, 15] on the link from P1 to P0. This approach yields a makespan of 22 time units.

In contrast, after nodes A, B, E, and F have been scheduled, the DLS algorithm evaluates dynamic levels for nodes C, G, and H with both processors P0 and P1, and discovers that node C and P0 form the best match. Node C is subsequently scheduled on P0 and node D is immediately released into the list of ready nodes even though it cannot be executed until time 9. Next, after evaluating dynamic levels for each of nodes D, G, and H with P0 and P1, the DLS approach selects and schedules node H on P1. The final two steps schedule node D on P0 and node G on P1, which results in an optimal schedule with makespan 16. By prohibiting the global clock from limiting processor selection, the DLS algorithm produces a more efficient schedule. Since additional processors are incorporated only as they are needed, the DLS approach constructs schedules that exhibit a natural "clustering" of nodes that communicate heavily, without sacrificing efficient use of the communication resources.

The performance curves in Fig. 10 show the percentage improvement in speedup obtained over HLFET for both the HDLFET and DLS algorithms. Although the two methods exhibit comparable performance at modest levels of graph parallelism, the DLS approach exhibits sharply increasing performance as the amount of parallelism increases, displaying average speedup improvements exceeding 75% over the HLFET algorithm. This illustrates its enhanced ability to assign each node its "preferred" processor. Notice that the DLS algorithm using initial node selection does not exhibit the performance degradation displayed by the HDLFET approach as the graph parallelism increases. Instead, it exhibits nearly equal performance as simultaneous node and processor selection. A likely explanation is that the greater freedom in processor selection allows the same scheduling steps to occur in these two cases, but in a different order. We will compare the performance of the DLS algorithm with two other scheduling techniques which account for IPC in Section V.

IV. Algorithm Streamlining

The dynamic level scheduling algorithm can be streamlined to provide faster execution without significant degradation in performance.

A. Initial Node Selection

In addition to the performance gain attained through removal of the global clock, the ability to gain analogous performance using initial node selection yields a big savings in execution time because examination of all node-processor combinations is no longer necessary. After investigating several methods for initially selecting a single ready node, we narrowed the choices down to the following two possibilities:

$$\max_i \{SL_i + C_{adj}[\max_k (D_{ki})]\} \tag{4}$$

$$\max_i \left\{ SL_i + \sum_k C_{adj}(D_{ki}) \right\} \tag{5}$$

D_{ki} represents the number of data units passed from node k to node i, while $C_{adj}(D)$ denotes the time needed to communicate D data units between adjacent processors. The first method selects the ready node that has the largest sum of the static level and the communication time to transfer the largest number of data units passed into the node from any immediate predecessor. The second method selects the ready node that maximizes the sum of the static level and the sum of the adjacent processor communication times for *each* data transfer from the ready node's predecessors. Equation (4) performs slightly better than (5). The maximum communication time is more important than the sum of the communication times, presumably because it may be possible to avoid only one IPC cost if the predecessors are located on different processors. Moreover, the DLS algorithm is particularly effective at overlapping communication with execution of other nodes that can be immediately invoked, thereby obtaining the other communications virtually without cost.

B. Limiting Processor Selection

To further decrease the time required for scheduling, we can reduce the number of processors for which dynamic levels are evaluated with the given node. For many multicomputer networks, a scheme based on a center of mass principle proves to be effective. This method identifies the processor locations for each predecessor of the candidate node to be scheduled, and obtains the number of data units to be sent in each transfer. Using the predecessor processor positions in an appropriate coordinate system, and the number of data units

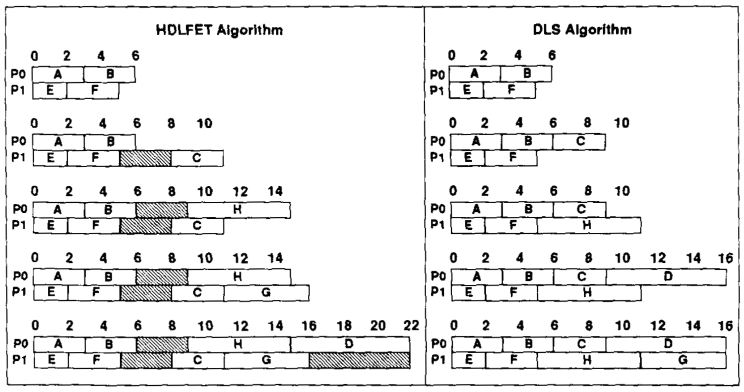


Fig. 9. Scheduling progressions taken by the HDLFET and DLS algorithms.

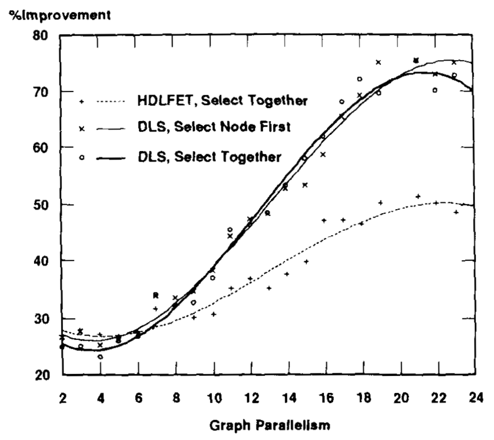


Fig. 10. Performance improvement in speedup over HLFET.

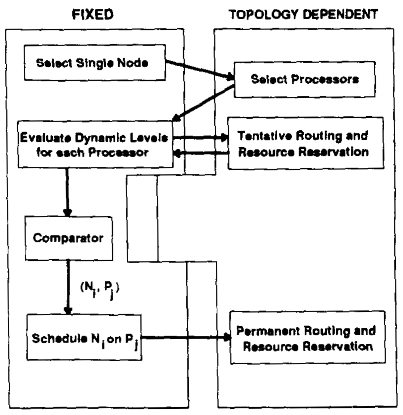

FIXED
TOPOLOGY DEPENDENT
Select Single Node
Select Processors
Evaluate Dynamic Levels for each Processor
Tentative Routing and Resource Reservation
Comparator
(N_i, P_j)
Schedule N_i on P_j
Permanent Routing and Resource Reservation

Fig. 11. The algorithm specification.

as a weighting function, the technique calculates the center of mass of the data and rounds it to the nearest processor location. It then limits candidate processors to those within a fixed radius of this center processor, with a few processors located outside this range included to promote spreading of the load. Intuitively, the center of mass calculation compels each predecessor processor to pull the candidate node toward it, with an attractive force that is directly proportional to the amount of data communicated. Although this technique is not intended as a panacea for all architectures, it is reasonable to assume that similar techniques that limit the number of processors can be developed for each topology with only minor performance penalty.

The diagram in Fig. 11 illustrates the operations performed by the streamlined algorithm at each scheduling step, where each operation is classified as residing in the fixed or topology-dependent component. The fixed component begins by selecting a single node according to (4). It passes the chosen node to the topology dependent section, which returns a list of candidate processors for scheduling this node. The topology independent section evaluates dynamic levels for each processor using the tentative routing and resource reservation routines contained in the topology dependent section. After obtaining the processor that maximizes the dynamic level, the algorithm schedules the node, and invokes the permanent routing and resource reservation routines. The simplicity of this scheme permits execution speeds suitable for a prototyping environment.

V. Performance Results

In this section, we compare the performance of the streamlined DLS algorithm with the *internalization* [5] clustering technique and a modified version of the *linear clustering* [11] algorithm. A modification to the latter algorithm was necessary because the method used to assign clusters to processors was not described clearly enough for implementation. Our modified version, which we refer to as the critical-path clustering algorithm, uses the linear clustering method to determine a set of clusters, but substitutes a processor assignment algorithm from the declustering approach [18].

To get an indication of relative performance, we randomly generated 100 graphs containing between 50 and 150 nodes, where the node execution times were uniformly distributed over [4, 20], and the number of data samples passed between nodes was uniformly distributed over [1, 5]. We scheduled these graphs onto simulated shared bus architectures con-

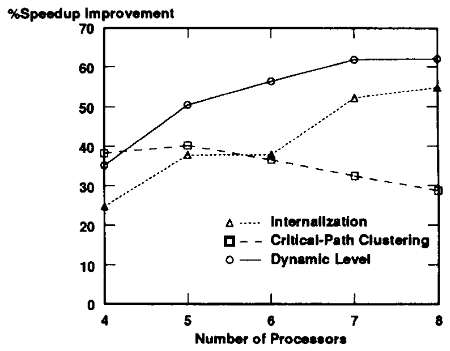


Fig. 12. Percentage improvement in speedup over HLFET when $\overline{C(N_i, N_j)} = 0.5 \times \overline{E(N_i)}$.

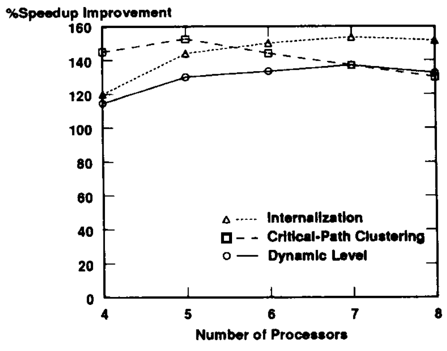


Fig. 13. Percentage improvement in speedup over HLFET when $\overline{C(N_i, N_j)} = \overline{E(N_i)}$.

taining between four and eight processors. Each simulated architecture contained dedicated communication hardware for IPC, allowing each processor to perform computation while communicating with another processor through the shared memory. We examined three separate cases, each of which uses a different value for the IPC cost per data sample. In case 1, each data sample requires 2 time units for communication through the shared bus. The mean communication time between nodes, $\overline{C(N_i, N_j)}$, is therefore 6, which makes the average communication time exactly one-half the mean node execution time $(\overline{E(N_i)} = 12)$. In case 2, each data sample requires 4 time units for communication $(\overline{C(N_i, N_j)} = 12)$, so the mean communication time equals the mean node execution time. In case 3, each data sample requires 8 time units for communication so the mean communication time is twice the average node execution time $(\overline{C(N_i, N_j)} = 24 = 2 \times \overline{E(N_i)})$.

The next three figures show the percentage improvement in speedup over HLFET displayed by each of the three scheduling techniques in cases 1, 2, and 3 respectively. Despite the greedy nature of the DLS algorithm, it outperforms the other two techniques in case 1, as shown in Fig. 12. This performance advantage occurs because the DLS technique is more adept at overlapping communication with computation than the other techniques. As the IPC cost increases, this advantage becomes less important. Any erroneous scheduling decisions taken by the DLS technique become more costly, so the clustering techniques surpass it in performance in cases 2 and 3, as shown in Fig. 13 and Fig. 14.

Although the greedy behavior of the DLS algorithm causes some loss of performance, it imparts a big advantage in scheduling speed. As shown in Fig. 15, the average time needed to schedule each graph on a Sun SPARCstation 1 is significantly smaller than that for the other two techniques. For example, in the 8-processor case, the average scheduling time per graph was 8 s for the DLS algorithm, 175 s for the critical-path clustering strategy, and 228 s for the internalization approach.

To supplement these results, we scheduled four digital signal processing algorithms on a simulated shared bus architecture with four Motorola DSP 56001 processors, each containing dedicated communication hardware to allow overlap of communication with computation. The programs included two

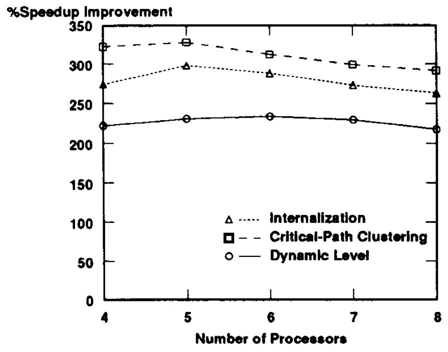


Fig. 14. Percentage improvement in speedup over HLFET when $\overline{C(N_i, N_j)} = 2 \times \overline{E(N_i)}$.

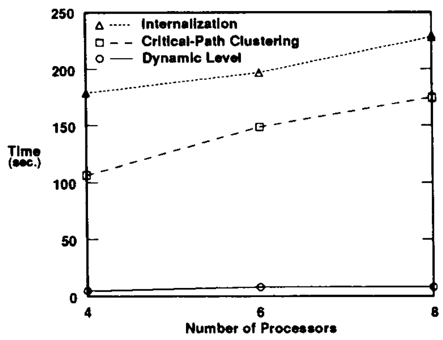


Fig. 15. Average time required to schedule each graph (in seconds).

sound synthesis algorithms, a telephone channel simulator, and a quadrature amplitude modulation (QAM) transmitter. We show the mixed-grain nature of these programs using the node execution time statistics in Table I. Each arc connecting two nodes carries exactly one data sample, which requires 18 processor cycles for IPC.

The schedule lengths in processor cycles are shown below in Table II for each scheduling technique. The results in this mixed-grain environment differ slightly from those obtained using the random graphs. Although the median communication time exceeded the median node execution time by at least a

TABLE I
NODE EXECUTION TIME STATISTICS FOR 4 DSP ALGORITHMS

DSP Algorithm(#nodes)	Minimum	Mean	Median	Maximum
Sound Synthesis I (26)	1	11.63	7	73
Sound Synthesis II (27)	1	5.74	4	21
Telephone Channel Simulator (67)	1	12.54	7	146
QAM Transmitter (411)	1	15.77	4	74

TABLE II
SCHEDULE LENGTHS (IN PPROCESSOR CYCLES) FOR 4 DSP ALGORITHMS

DSP Algorithm(#nodes)	Dynamic Levels	C-P Clustering	Internalization
Sound Synthesis I (26)	112	103	113
Sound Synthesis II (27)	150	188	180
Telephone Channel Simulator (67)	346	318	383
QAM Transmitter (411)	4598	3763	4549

TABLE III
TIME REQUIRED TO CONSTRUCT A SCHEDULE (IN SECONDS) FOR EACH SCHEDULING TECHNIQUE

DSP Algorithm(#nodes)	Dynamic Level	C-P Clustering	Internalization
Sound Synthesis I (26)	0.289	2.24	6.42
Sound Synthesis II (27)	0.262	2.51	4.17
Telephone Channel Simulator (67)	2.33	12.64	47.97
QAM Transmitter (411)	21.22	603.41	3272.69

factor of 2 in each of the four programs, the DLS algorithm outperformed the internalization technique in three out of the four cases and returned a makespan within 1.1% in the remaining case. The DLS strategy outperformed the critical-path clustering algorithm in one case and produced makespans within 8.1% in two of the remaining three cases. These results are impressive when we consider the average times required to schedule each graph, shown in Table III. Whereas critical-path clustering and internalization required 10 and 55 min respectively to schedule the QAM transmitter, the DLS algorithm constructed a schedule in less than 22 s with a makespan within 1.1% of that returned by the internalization technique.

We use the DLS technique as one of a collection of scheduling techniques, each of which performs well in certain situations. The DLS algorithm is most useful when scheduling time is a critical factor, as happens in interactive prototyping environments. If performance is the most important criterion, we use the declustering algorithm, which outperforms both the critical-path clustering and internalization techniques [18]. Due to its scheduling speed, the DLS algorithm can be effectively used within iterative scheduling strategies.

VI. HETEROGENEOUS PROCESSOR EXTENSION

The presence of heterogeneous processors further complicates the efficacious matching of nodes with processors because it may be advantageous to separate nodes that lie on an entirely sequential path onto different processors to exploit specialized hardware features. In addition to the tradeoff between exploitation of parallelism and IPC cost, this environment introduces a new tradeoff between varying processor computation speeds and IPC cost.

The dynamic level expression requires several modifications to accommodate a heterogeneous processing environment. Since the static level $SL(N_i)$ loses its meaning when execution times vary between processors, we define the adjusted median execution time of node N_i, denoted $E^*(N_i)$, to be the median of the execution times of node N_i over all the processors, with the stipulation that the largest finite execution time is substituted if the median itself is infinite. $SL^*(N_i)$ correspondingly denotes the static level of node N_i, calculated using these adjusted median execution times. To account for the varying processing speeds, we define

$$\Delta(N_i, P_j) = E^*(N_i) - E(N_i, P_j). \tag{6}$$

A large positive $\Delta(N_i, P_j)$ indicates that processor P_j executes node N_i more rapidly than most processors, while a large negative $\Delta(N_i, P_j)$ indicates the reverse. By incorporating these terms, our first extended dynamic level $DL_1(N_i, P_j, \Sigma)$ quickly follows:

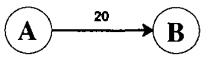
$$DL_1(N_i, P_j, \Sigma) = SL^*(N_i) - \max[DA(N_i, P_j, \Sigma), TF(P_j, \Sigma)] + \Delta(N_i, P_j). \tag{7}$$

The static level component indicates the importance of the node in the precedence hierarchy, giving higher priority to nodes located further from the terminus (in adjusted median execution time). The maximization term reflects the availability of the processing and communication resources, penalizing node-processor pairs that incur large communication costs. The delta term accounts for the processor speed differences, adding priority to processors that execute the node quickly, and subtracting priority from processors that execute it slowly. Note that when all processors are identical, $\Delta(N_i, P_j) = 0$ and $SL^*(N_i) = SL(N_i)$, causing this formula to converge to the homogeneous processor dynamic-level expression.

The chart in table IV shows the mean percentage speedup improvement obtained from using extended dynamic level $DL_1(N_i, P_j, \Sigma)$ for node and processor selection over the

TABLE IV
SPEEDUP IMPROVEMENTS IN USING $DL_1(N_i, P_j, \Sigma)$ FOR NODE AND PROCESSOR SELECTION

Percentage Improvement in Speedup Over HLFET					
Algorithm	Mean	Std. Deviation	Minimum	Median	Maximum
DL1, 1 Node	125.33%	54.02%	59.24%	111.35%	340.09%
DL1, 3 Nodes	127.71%	48.08%	66.96%	113.04%	293.16%
DL1, All Nodes	132.58%	50.81%	70.07%	118.64%	355.45%



Execution Times

	P0	P1
A	5	8
B	∞	7

Fig. 16. A descendant consideration example.

HLFET method using static levels. We obtained these values from scheduling 100 randomly generated graphs onto a 16 processor mesh, where the node execution times and amount of data transferred between nodes were uniformly distributed over [5, 15]. Processor speeds were varied by a factor of 16. To simulate the effects of special hardware features, several randomly chosen nodes were constrained to be executed on a small subset of processors. We investigated three cases: when a single ready node maximizing (4) was considered for scheduling at each step, when the three ready nodes with highest values of (4) were considered at each step, and when all ready nodes were considered at each step. The performance increases slightly as the number of nodes considered at each scheduling step increases. As before, these experiments using randomly generated graphs should be considered indicative, rather than evidential, of performance with practical applications.

A. Descendant Consideration

This natural dynamic level extension overlooks several subtle effects introduced when different types of processors are present. Although $DL_1(N_i, P_j, \Sigma)$ indicates how well node N_i is matched with processor P_j, it fails to consider how well the descendants of N_i are matched with P_j. Consider the simple example shown in Fig. 16, in which node A passes 20 data units to its descendant node B. Since P_0 executes node A faster than P_1,

$$DL_1(A, P_0, \Sigma) > DL_1(A, P_1, \Sigma), \tag{8}$$

causing node A to be scheduled on P_0. However, because node B cannot be executed on P_0, the scheduling of node A on P_0 forces an interprocessor communication of 20 data units, which we assume for simplicity to take 20 time units as shown in the top diagram of Fig. 17. If descendant B had been considered during the scheduling of node A, both nodes would have been scheduled on P_1, as shown in the bottom diagram in Fig. 17. This example illustrates that it is not always advantageous to schedule a node on the processor that executes it most rapidly, because of this descendant effect.

For each node N_i, we consider the descendant to which N_i passes the most data, designating this node as $D(N_i)$, and the amount of data passed between them as $d(N_i, D(N_i))$. Recalling that $E[D(N_i), P_j]$ indicates how quickly node $D(N_i)$ executes on processor P_j, we define another term

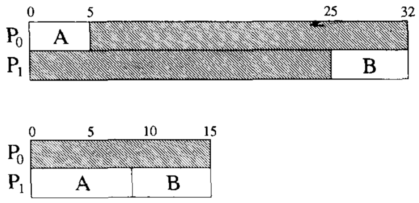


Fig. 17. Schedules for possible placements of nodes A and B.

$F[N_i, D(N_i), P_j]$ to indicate how quickly $D(N_i)$ can be completed on any other processor if N_i is executed on P_j:

$$F[N_i, D(N_i), P_j] = C_{adj}[d(N_i, D(N_i))] + \min_{k \neq j} E[D(N_i), P_k]. \tag{9}$$

This is a lower bound on the time needed to finish execution of $D(N_i)$ on any processor other than P_j after N_i finishes execution on P_j. We then define a descendant consideration term as

$$DC(N_i, P_j, \Sigma) = E^*[D(N_i)] - \min\{E[D(N_i), P_j], F[N_i, D(N_i), P_j]\} \tag{10}$$

and update the dynamic level expression to

$$DL_2(N_i, P_j, \Sigma) = DL_1(N_i, P_j, \Sigma) + DC(N_i, P_j, \Sigma). \tag{11}$$

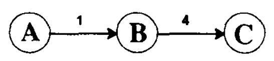
The descendant consideration term is the difference between the adjusted median execution time of $D(N_i)$ and a lower bound on the time required to finish execution of $D(N_i)$ after N_i finishes execution on P_j. This indicates how well the "most expensive" descendant of node N_i matches up with processor P_j, if N_i is scheduled on P_j. If P_j executes $D(N_i)$ very quickly compared with other processors, $DC(N_i, P_j, \Sigma)$ becomes $E^*[D(N_i)] - E[D(N_i), P_j]$, causing an increase in $DL_2(N_i, P_j, \Sigma)$. Conversely, if P_j executes $D(N_i)$ very slowly compared with most processors, $DC(N_i, P_j, \Sigma)$ becomes negative, causing a decrease in dynamic level. If $E[D(N_i), P_j]$ is equal to the median execution time of $D(N_i)$, the descendant term is appropriately zero. The term is also zero when all processors are homogeneous, reflecting the fact that all processors can execute the descendant equally well.

The percentage improvement in speedup over HLFET in using $DL_2(N_i, P_j, \Sigma)$ for node and processor selection is shown in Table V for the same three cases. When using DL2, the mean and median performance improve in every case (1 node, 3 nodes, All nodes) over the corresponding measurements for DL1, reflecting the importance of descendant consideration.

Note that if N_i is scheduled on processor P_j, its "most expensive" descendant $D(N_i)$ is not necessarily scheduled

TABLE V
SPEEDUP IMPROVEMENTS IN USING $DL_2(N_i, P_j, \Sigma)$ FOR NODE AND PROCESSOR SELECTION

Percentage Improvement in Speedup Over HLFET					
Algorithm	Mean	Std. Deviation	Minimum	Median	Maximum
DL2, 1 Node	129.29%	51.90%	65.52%	112.04%	338.10%
DL2, 3 Nodes	135.45%	54.97%	62.91%	121.05%	327.98%
DL2, All Nodes	138.97%	53.88%	79.44%	123.32%	333.96%



Execution Times

	P0	P1	P2
A	3	6	6
B	5	5	5
C	8	4	4

Fig. 18. Another descendant consideration example.

on the same processor. One reason for this is that $D(N_i)$ may itself have a descendant that executes slowly on P_j. Consider the graph shown in Fig. 18. Through evaluation of $DL_2(A, P_j, \Sigma)$ for each processor, the procedure determines that node A prefers P_0. The descendant consideration term is zero, because node B executes equally quickly on every processor. However, the immediate scheduling of node B onto P_0 leads to an inefficiency because node B's descendant, node C, executes slowly on P_0. Although node C could be shifted onto P_1 or P_2, this incurs the communication of four data units between B and C. It is more effective to postpone the scheduling of B until its dynamic level $DL_2(B, P_j, \Sigma)$ is calculated, because after incorporating the descendant consideration term for C, the procedure will correctly shift node B onto either P_1 or P_2.

B. Resource Scarcity

In addition to the descendant consideration effect, a heterogeneous processing environment also introduces a cost associated with a node *not* being executed on a certain processor. This phenomenon occurs because effective processing resources may be scarcer for some nodes than others at certain states. To illustrate this cost, consider a situation in which an APEG is being scheduled onto a five processor system and the partial schedule at the current state Σ' is shown in Fig. 19. There are two nodes left to be scheduled, M and N, whose execution times on each of the processors are shown in Table VI below. After evaluating dynamic levels for M and N, the algorithm finds that both nodes would like to be scheduled on P_0 at the current state. If node M is initially scheduled on P_0 in the time interval [8, 12], the earliest time that node N can finish execution is at time 16 on P_0. However, if node N is initially scheduled on P_0, node M can be scheduled on P_2, leading to a schedule with makespan 12. There is a negligible cost if node M is not assigned to P_0 at state Σ', because P_2 can execute the node almost as rapidly. Conversely, there is a large cost if node N is not assigned to P_0 at this state because P_0 is the only processor which executes node N so quickly.

This example illustrates that while $\Delta(M, P_0)$ and $\Delta(N, P_0)$ account for processor speed variations, they fail to consider how important it is for nodes M and N to obtain processor P_0 at the current state. Here, node N should have greater claim to P_0, because its cost in not obtaining it is much higher.

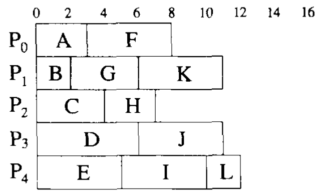


Fig. 19. The schedule at state Σ'.

TABLE VI
EXECUTION TIMES FOR NODES M AND N

Node	P0	P1	P2	P3	P4
M	4	8	5	8	8
N	4	8	∞	8	8

This cost in depriving a node of a certain processor stems from the relative scarcity of the processing resources for each node at the current state. Its existence indicates that a more careful apportionment of nodes to processors is necessary in an inhomogeneous environment.

This situation arises at scheduling steps which have 2 or more ready nodes simultaneously desiring the same processor. To characterize this resource scarcity cost, we first define

$$DL(N_i, P_{j^*}, \Sigma) = \max_j DL(N_i, P_j, \Sigma), \tag{12}$$

thereby designating P_{j^*} as a preferred processor of node N_i. That is, j^* is the index of a processor that maximizes the dynamic level with node N_i at state Σ. We then define the cost in not scheduling node N_i on its preferred processor at the current state as

$$C(N_i, P, \Sigma) = DL(N_i, P_{j^*}, \Sigma) - \max_{k \neq j^*} DL(N_i, P_k, \Sigma). \tag{13}$$

$C(N_i, P, \Sigma)$ is the difference between the highest dynamic level and the second highest dynamic level for node N_i over all processors at the current state. In other words, if a node other than N_i is scheduled on processor P_{j^*} at state Σ, the cost is the amount of dynamic level that is lost in having to settle for the "next best" processor. Notice that this cost is only nonzero when there is a single preferred processor P_{j^*} such that for all $k \neq j^*$

$$DL(N_i, P_{j^*}, \Sigma) > DL(N_i, P_k, \Sigma). \tag{14}$$

If this condition is not satisfied, the cost is zero because even if another node is scheduled onto one of N_i's preferred processors, N_i will still have at least one processor with which it can attain the same dynamic level (assuming communication resources are still available). $C(N_i, P, \Sigma)$ is a function of

TABLE VII
SPEEDUP IMPROVEMENTS IN USING GENERALIZED DYNAMIC LEVELS

Percentage Improvement In Speedup Over HLFET					
Algorithm	Mean	Std. Deviation	Minimum	Median	Maximum
GDL1, 3 Nodes	133.23%	49.35%	74.83%	119.12%	342.18%
GDL1, All Nodes	145.82%	55.50%	60.26%	136.42%	384.21%
GDL2, 3 Nodes	138.18%	57.27%	72.26%	116.81%	364.65%
GDL2, All Nodes	145.92%	58.82%	72.41%	129.55%	389.36%

the entire set of processors P, because calculating this term requires evaluation of dynamic levels for node N_i over all processors at the current state. We now define a "generalized" dynamic level as follows:

$$GDL(N_i, P, \Sigma) = DL(N_i, P_{j^*}, \Sigma) + C(N_i, P, \Sigma). \qquad (15)$$

We now consider several ready node candidates simultaneously, and the node that maximizes $GDL(N_i, P, \Sigma)$ is scheduled on its preferred processor. The scheduling results obtained using generalized dynamic levels are shown in Table VII. GDL1 refers to the generalized dynamic level calculated using DL_1 as its base level. Similarly, GDL2 uses DL_2 as its base level. As indicated by these results, the generalized dynamic levels provide a significant performance improvement, especially when they are evaluated over all ready nodes. However, the algorithm that considers all ready nodes at each step has complexity $O[n^3 p * f(p)]$, where n is the number of nodes, p is the number of processors, and the function used to route a path between two given processors on the targeted architecture is $O[f(p)]$. Since this algorithm is too time-consuming for use in an interactive environment, we recommend the method that chooses three candidate nodes at each step and uses GDL2 for node and processor selection. The complexity of this scheme is $O[n^3 + n^2 p * f(p)]$, which can be trimmed further if the generalized dynamic levels are only evaluated over a chosen subset of processors.

VII. SUMMARY AND CONCLUSIONS

We have presented a new compile-time scheduling strategy called dynamic-level scheduling that accounts for interprocessor communication overheads when mapping precedence graphs onto multiple processor architectures. This technique eliminates shared resource contention by performing scheduling and routing simultaneously to enable the scheduling of all communications as well as all computations. In heterogeneous processing environments, it accounts for varying processor speeds and delivers a more careful apportionment of processing resources. The algorithm is split into two sections to permit a rapid retargeting to any desired multiple-processor architecture by loading in the correct topology-dependent section. The streamlined algorithm is fast enough to be used in interactive environments and is valuable as one of a set of scheduling techniques. This approach attains good scheduling performance by effectively trading off load balancing with interprocessor communication, and efficiently overlapping communication with computation. The DLS technique is also suitable for use in more complicated scheduling techniques as a means of evaluating makespan after a preliminary mapping has been determined.

We are currently focusing on optimization techniques that were too time-consuming for the prototyping environment, but can be applied in the final design phase. An initial prepass-scheduler can be used to promote a more global scheduling perspective by prohibiting the exploitation of certain parallelism instances. This can be accomplished through the use of various graph transformation techniques such as the insertion of additional precedence constraints. Since the DLS technique is fast, iterative approaches designed to reduce the scheduling bottleneck may prove beneficial. The interaction between scheduling and routing also merits further examination. Since previous communication resouce reservations may block a node from being scheduled on a certain processor, the rerouting of data transfer paths may facilitate a better node-processor mapping.

ACKNOWLEDGMENT

We thank the anonymous reviewers for helpful suggestions.

REFERENCES

[1] S. F. Nugent, "The iPSC/s direct-connect communications technology," in *Proc. Third Conf. Hypercube Concurrent Comput. and Appl.*, vol. 1, Jan. 1988.

[2] S. Borkar *et al.*, "iWarp: An integrated solution to high-speed parallel computing," Carnegie Mellon Tech. Rep. CMU-CS-89-104, pp. 1–10, Jan. 1989.

[3] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, no. 2, Jan. 1987.

[4] E. A. Lee and S. Ha, "Scheduling strategies for multiprocessor real-time DSP," in *Proc. Globecom*, Nov. 1989.

[5] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Cambridge, MA: M.I.T. Press, 1989.

[6] W. W. Chu, L. J. Holloway, M. T. Lan, and K. Efe, "Task allocation in distributed data processing," *IEEE Comput. Mag.*, pp. 57–69, Nov. 1980.

[7] W. W. Chu and L. M. T. Lan, "Task allocation and precedence relations for distributed real-time systems," *IEEE Trans. Comput.*, vol. C-36, no. 6, pp. 667–679, June 1987.

[8] S. W. Bollinger and S. F. Midkiff, "Processor and link assignment in multicomputers using simulated annealing," in *1988 ICPP Proc.*, vol. 1, Aug. 1988, pp. 1–7.

[9] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *IEEE Comput. Mag.*, pp. 50–56, June 1982.

[10] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. Software Eng.*, vol. SE-3, no. 1, pp. 85–93, Jan. 1977.

[11] S. J. Kim and J. C. Browne, "A general approach to mapping of parallel computations upon multiprocessor architectures," in *1988 ICPP Proc.*, vol. 3, Aug. 1988, pp. 1–8.

[12] J. C. Bier, E. E. Goei, W. H. Ho, P. D. Lapsley, M. P. O'Reilly, G. C. Sih, and E. A. Lee, "Gabriel: A design environment for DSP," *IEEE Micro Mag.*, Oct. 1990.

[13] R. P. Bianchini Jr. and J. P. Shen, "Interprocessor traffic scheduling algorithm for multiple-processor networks," *IEEE Trans. Comput.*, vol. C-36, no. 4, pp. 396–409, Apr. 1987.

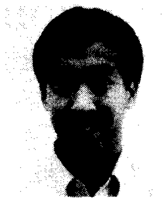
[14] T. C. Hu, "Parallel sequencing and assembly line problems," *Oper. Res.*, vol. 9, no. 6, pp. 841–848, Nov. 1961.

[15] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Commun. ACM*, vol. 17, no. 12, pp. 685–690, Dec. 1974.

[16] W. H. Kohler, "A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems," *IEEE Trans. Comput.*, pp. 1235–1238, Dec. 1975.

[17] W. H. Yu, "LU decomposition on a multiprocessing system with communication delay," Ph.D. dissertation, U.C. Berkeley, 1984.

[18] G. C. Sih and E. A. Lee, "A multiprocessor scheduling strategy," UCB/ERL Memo, M90/119, pp. 1–34, Dec. 1990.



Gilbert C. Sih (S'83–M'91) was born in Madison, WI, on Jan. 31, 1962. In 1985, he received the B.S. degree in applied mathematics, engineering, and physics from the University of Wisconsin at Madison. In 1991, he received the Ph.D. degree in EECS from the University of California at Berkeley, where he worked on the Gabriel and Ptolemy software design systems for programmable digital signal processing.

His research interests include digital signal processing, parallel processing, and multiprocessor scheduling. He is presently a senior engineer at QUALCOMM Inc., San Diego, CA, where he works on speech and adaptive signal processing.

Edward A. Lee (S'80–M'86) received the B.S. degree from Yale University in 1979, the masters (S.M.) degree from M.I.T. in 1981, and the Ph.D. degree from U.C. Berkeley in 1986.

He is an Associate Professor in the Electrical Engineering and Computer Science Department at U.C. Berkeley. His research activities include parallel computation, architecture and software techniques for programmable DSP's, design environments for development of real-time software, and digital communication. He is co-author of *Digital Communication*, with D. G. Messerschmitt (Kluwer Academic, 1988), and *Digital Signal Processing Experiments* with Alan Kamas (Englewood Cliffs, NJ: Prentice Hall, 1989), as well as numerous technical papers. From 1979 to 1982 he was a member of technical staff at Bell Telephone Laboratories, Holmdel, NJ, in the Advanced Data Communications Laboratory, where he did extensive work with early programmable DSP's, and exploratory work in voiceband data modem techiques and simultaneous voice and data transmission.

Dr. Lee was a recipient of a 1987 NSF Presidential Young Investigator award, an IBM faculty development award, the 1986 Sakrison prize at U.C. Berkeley for the best thesis in Electrical Engineering, and a paper award from the IEEE Signal Processing Society. He is chairman of the VLSI Technical Committee of the Signal Processing Society, co- program chair of the 1992 Application Specific Array Processor Conference, and a member of the DSP Committee of the Circuits and Systems Society and the Communication Theory Committee of the Communications Society.