

Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput

Phu D. Hoang and Jan M. Rabaey, *Member, IEEE*

Abstract—Multiple processor architectures have long been acknowledged as the cost-effective alternative to custom implementations for handling the enormous amount of computation in digital signal processing (DSP). However, the lack of effective techniques to automatically schedule the DSP program onto the multiple processors has prevented their widespread use. Although many strategies have been proposed, none has been able to fully exploit all types of concurrency present in the program, and at the same time, address practical issues of multiprocessor systems such as interprocessor communication delays, and memory and processor availability constraints. A program scheduling algorithm that simultaneously considers pipelining, retiming, parallel execution, and hierarchical node decomposition to maximize throughput while meeting these resource constraints is presented. The results on a set of benchmarks demonstrate the algorithm's ability to achieve near optimal speedups across a wide range of applications of various types of concurrency, with good scalability with respect to processor count.

I. INTRODUCTION

IN recent years, we have seen a tremendous increase in the computing power of digital signal processors. However, we have concurrently experienced an even greater increase in computational requirements of digital signal processing (DSP) applications. Eventually, physical constraints will limit the performance of single processors, and the evolution to multiple processors will be inevitable. The major obstacle to the prevalent use of multiprocessor systems, however, has been the absence of adequate tools to automatically schedule the program onto the multiple processors. A careful analysis reveals four major issues in finding such a schedule: Concurrency, granularity, estimation, and feasibility.

Concurrency: The concurrency can be both *spatial concurrency (parallelism)*, where tasks can be executed by several processors simultaneously, and *temporal concurrency (pipelining)*, where chains of tasks can be divided into stages, with every stage handling results obtained from the previous stage. Pipelining is possible here due to the inherent nature of DSP to repeat the same computation to each frame of the input stream. When there is recursion in the program, however, pipelining is no longer

applicable as it may alter the computation intended. One efficient way to obtain speedup in this case is to perform a retiming transformation [13], which moves logical delays around in the cycle to minimize the execution time.

Granularity: The concurrency to be exploited can exist at all levels of granularity, from functions to loops to individual operations, and the appropriate level must be determined for the scheduler to be efficient, yet effective in exploiting all the necessary concurrency.

Estimation: To obtain meaningful results, accurate estimations of the computation times and memory requirements of the tasks are necessary. A clear model of the underlying processor interconnection is also required to calculate costs due to interprocessor communications.

Feasibility: The scheduling tool must reject illegal schedules such as those that do not satisfy data dependencies, those that use more processors or more memory than available, or those that attempt to pipeline where forbidden to do so.

The scheduler to be presented in this paper is a compile-time, hierarchical multiprocessor scheduler that strives to maximize the throughput of the resultant implementation while considering all the issues stated above. The algorithm works on a flow graph representation of the program, where nodes represent computations and edges represent data precedences. The flow graph is hierarchical in that the body of a function call or an iteration is represented by a subgraph, which is contracted into a single node at the next hierarchy level. The hierarchical iteration node has parameters indicating the range of the iteration as well as whether the iteration can be executed in parallel or serial. A flow graph representation of the program exposes the available concurrency explicitly, and is crucial for any multiprocessor scheduling technique.

II. AN EXAMPLE

Consider an example of a pitch extractor algorithm [21], whose flow graph is shown in Fig. 1(a). For simplicity, nodes are labeled with the same computation costs (in μs) and communication delays are ignored. On a single processor, the time elapsed between sample iterations, or the iteration period, is $160 \mu\text{s}$. The computation has both spatial and temporal concurrency. To maximize the throughput, both types have to be exploited. Fig. 1(b) shows the flow graph pipelined into six stages, with the fifth stage having three tasks executed in parallel. If eight processors

Manuscript received September 3, 1991; revised June 2, 1992. The associate editor coordinating the review of this paper and approving it for publication was Dr. E. F. Deprettere.

The authors are with the Department of Electrical Engineering, University of California, Berkeley, CA 94720.

IEEE Log Number 9208199.

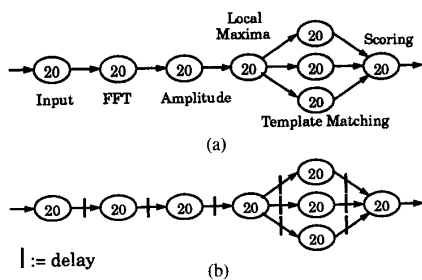


Fig. 1. Pipelining and parallel execution.

are available, this partition is optimal, yielding an iteration period of $20 \mu\text{s}$ and a speedup of 8. For scheduling techniques that only consider pipelining, the parallel template matching task becomes a bottleneck, limiting the iteration period to be $60 \mu\text{s}$, a speedup of only 2.7 out of 8. When considering only parallelism, only the template matching task can be sped up. The critical path limits the iteration period to $120 \mu\text{s}$, a speedup of 1.3 out of 8.

This example does not contain cycles, so retiming is not necessary. However, if the flow graph has a structure like Fig. 2(a), retiming would be needed to yield the optimal throughput. The feedback in Fig. 2(a) prohibits the addition of pipeline stages on the feedforward path, making the cycle the bottleneck computation. However, the cycle can be retimed by moving a delay from the feedback path to the forward path as shown in Fig. 2(b). The combination of retiming, pipelining, and parallelism fully exploits the available concurrency in the graph.

III. COMPILATION ENVIRONMENT

The scheduling technique has been implemented as part of a multiprocessor compiler system called McDAS [5]. McDAS currently targets the Sequent and Switchable multiprocessor architecture supporting real-time applications (SMART) [10] multiprocessor systems. The Sequent system contains 14 Intel 386 processors connected by a single shared bus. The SMART system is composed of a linear array of 10 AT&T DSP32C DSP processors connected by a shared configurable bus. The input language used is an extended version of Silage [3], a signal-flow language developed especially for DSP specification. The compilation process involves translating the Silage input into a hierarchical flow graph, scheduling the flow graph, and generating C or assembly code for each partitioned subgraph. The basic concept of the scheduling technique as applied to the SMART system was introduced in [4]. This paper describes the scheduler independent of the underlying architecture. All aspects of the algorithm are described in greater detail, its computational complexity derived, and more extensive results are given.

IV. PREVIOUS WORK

Many techniques have been proposed to schedule DSP programs onto multiprocessors. These techniques can be classified into two categories: *Nonoverlap execution*

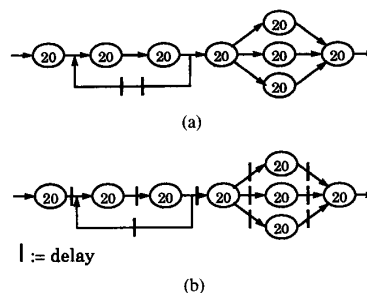


Fig. 2. Pipelining, retiming, and parallel execution.

scheduling, which schedules one program execution and replicate the schedule for each input frame, and *overlap execution scheduling*, which allows for overlapped execution of successive iterations. Typical approaches in the first class are based on list scheduling, which exploits concurrency within one program execution to achieve speedup [2], [9], [11], [15], [16], [18], [20]. All exploits only spatial concurrency, except for [16], which can consider systolic implementations as well. The second class, based on cyclostatic scheduling [12], [14], [19], takes advantage of the stream processing nature of DSP to exploit temporal concurrency between successive iterations. While these can yield optimum solutions, their exponential search strategies preclude a practical implementation of large applications. Chain partitioning [1] pipelines serial tasks on a linear array of processors to maximize throughput. Although this technique does not exploit parallelism, it comes closest to our stated objectives and is a basis for the approach we have adopted.

A number of techniques consider granularity [18], [20]. However, none allows the scheduler to decompose a node into smaller nodes when it is a scheduling bottleneck. This is a serious drawback, since the granularity of the nodes (and not the performance of the scheduling algorithm) is often the constraining factor towards reaching a high quality solution.

The remainder of the paper is organized as follows: Section V discusses our strategy for estimating computation cost, memory usage, and communication delay. Section VI presents the scheduling algorithm and analyzes its complexity. The scheduling results on a set of benchmarks are summarized in Section VI.

V. COMPUTATION, MEMORY, AND COMMUNICATION ESTIMATION

Let us represent the hierarchical graph as $G = (N, E)$, where N is the set of nodes and E is the set of edges in the top level hierarchy of the graph. N can be divided into three sets: The set of primitive nodes N_p , the set of function call nodes N_f , and the set of iteration nodes N_i . Function call nodes and iteration nodes are hierarchical nodes. Let $SG(\cdot): N_f \cup N_i \rightarrow G$ denote the function which returns the underlying subgraph of a hierarchical node, and let $N(\cdot): G \rightarrow N$ be a function which returns the set of nodes of a graph.

A. Estimating Computation Time

To model the computation costs, sample programs were profiled. From this, a cost, or weight w was assigned to each primitive node such as addition and multiplication. Costs were also derived for the overhead w_o of performing loop increments, loop tests, and function calls. With these values, the cost of every node in the hierarchy of any flow graph can be estimated by traversing the flow graph bottom up, accumulating computation times of primitive nodes into subgraphs, and so on up to the root graph.

Definition 1: $\forall n \in N$, the computation time $w(n)$ is given by the following three rules:

1) If $n \in N_p$, $w(n) =$ predefined cost, based on benchmark results.

2) If $n \in N_F$

$$w(n) = w_o(n) + \sum_{\nu \in N(SG(n))} w(\nu)$$

where $w_o(n)$ represents the overhead of the function call, as analyzed in the benchmarks.

3) For $n \in N_I$

$$w(n) = w_o(n, L) + L \cdot \sum_{\nu \in N(SG(n))} w(\nu)$$

where $w_o(n)$ represents the overhead of the loop, as analyzed in the benchmarks, and L is the iteration count.

Table I shows the completion time of a number of DSP applications as estimated using the technique above and as actually measured. The target processor is an AT&T DSP32C, the core processor for the SMART system. Each cycle corresponds to 20 ns. The error shows that the estimated computation time is accurate to within 5% of the actual time. Benchmarks on the Sequent multiprocessor yield the same level of accuracy [6].

B. Estimating Memory Requirements

A communication between two processors incurs memory storage at the destination processor to buffer the data. It is possible to keep track of the buffer memory usage during scheduling so that solutions which violate the memory limit can be discarded. An edge between two nodes assigned to different processors represents an inter-processor communication. Each communication is supported by a FIFO in our compilation strategy [5]. The buffer memory requirement of a node n on a processor p , denoted as $bm(n, p)$, is dependent on the size of the data on the edge and the difference in the source and destination pipeline stages. The parameter $bm(n, p)$ is used by the scheduler to prohibit a node from being assigned to a processor if executing this node would overflow the processor's buffer memory. To do this, each processor p has to also keep track of its remaining buffer memory size during scheduling. This parameter is denoted $bm_{\text{avail}}(p)$. Section VI will describe how $bm(n, p)$ and $bm_{\text{avail}}(p)$ are used by the scheduling algorithm.

There are a number of other memory parameters which can be estimated. These include program memory, mem-

TABLE I
COMPUTATION TIME ESTIMATION

Example	# Operations	Estimated (cyc)	Measured (cyc)	% Error
7th-IIR	55	6497	6414	+1.2%
8pt-DCT	87	2388	2511	-4.89%
Cordic	494	135 098	137 924	-2.04%
2 Norm	1926	109 812	114 099	-3.75%
Histogram	30,687	4 186 747	4 227 738	-0.97%

ory for static and global variables, and stack memory. While not yet implemented, the same technique used in computation estimation is extendable to estimate memory usage as well.

C. Estimating Communication Delay

The communication costs depend on the amount of data being sent and the distance between the source and destination processors. When a node is scheduled on a processor, the data transfers that are needed to bring the input variables to the processor (if data is nonlocal) are also scheduled on the appropriate bus or busses. This is done by building time slots of the proposed data transfers and merging them onto the time slots already scheduled onto the buses. If any proposed time slot conflicts with an existing time slot, it is scheduled on the next available time slot. This allows the scheduler to take bus congestion into account when calculating the arrival time of the data. More details on the time-slot model can be found in [6]. From this model, a number of parameters are derived to characterize the communication delay. The final quantity, the earliest starting time $\xi(n, p)$, is used by the scheduling algorithm to map nodes to processors.

Definition 2: The arrival time $t_{\text{arv}}(n_s, p)$ denotes the time at which data computed in a source node n_s is available at processor p . We assume the node n_s is already scheduled on a source processor and the data transfer(s) are scheduled on the appropriate bus or busses. $t_{\text{arv}}(n_s, p)$ gives the time the last data packet arrives at p , given the state of the bus congestion at that moment in time.

Definition 3: The available time $t_{\text{avail}}(n, p)$ is the time at which all input data to node n is available at processor p . It is calculated over the set $I(n)$ of all input nodes of n as

$$t_{\text{avail}}(n, p) = \max \{t_{\text{arv}}(n_i, p) | n_i \in I(n)\}. \quad (1)$$

Definition 4: The ready time $t_{\text{ready}}(p)$ is the time processor p has finished executing its last assigned node.

For a node n to start on a processor p , all of its input data must be available at processor p , and the processor must have completed any previously assigned computation.

Definition 5: The earliest starting time $\xi(n, p)$ of node n on processor p is defined as

$$\xi(n, p) = \max \{t_{\text{avail}}(n, p), t_{\text{ready}}(p)\}. \quad (2)$$

$\xi(n, p)$ effectively abstracts the underlying architecture to the level of the starting times of nodes on processors.

The scheduling algorithm is only concerned with this information to make its decisions, irrespective of the architecture. As a result, $\xi(n, p)$ serves as the interface between the architecture-dependent estimations and the scheduling algorithm. This modularity allows the scheduler to deal with any architecture in a unified manner. Each architecture would only have to provide its own calculation of $\xi(n, p)$.

VI. SCHEDULING STRATEGY

We now present a heuristic scheduling algorithm that simultaneously considers pipelining, parallel execution, and retiming to maximize throughput while meeting processor and memory constraints. The algorithm starts at the top level of hierarchy, and proceeds top-down. Hierarchical nodes are systematically broken down to smaller nodes when more concurrency can be exploited.

Definition 6: The stagetime T is defined as the reciprocal of the throughput of the system.

The stagetime equals the time allocated to each pipeline stage, and thus to each processor in that stage. We minimize this stagetime T to maximize the throughput. During the minimization process, the following condition must always be satisfied.

Maximum Granularity Condition: The computation time of the largest node in the graph (at a given level of granularity) W_{\max} has to be $\leq T$.

The reason for this is clear. A node of computation time $> T$ cannot be scheduled on a processor with only time T to execute.

The algorithm is based on performing a binary search to find the minimal T given the number of processors P . The upper bound of T , denoted as UB, is initialized to W_{total} , the computation time of the entire graph. The lower bound LB is set to W_{total}/P , the ideal stagetime given P processors. The pseudocode for the algorithm is shown below.

Main ():

1. Assign computation times to nodes
2. LB = Lower bounds on T, UB = Upper bounds on T
3. $W_{\max} = \text{MaxWeight}(G)$
4. **repeat while** (LB < UB) {
 1. proc = **Schedule**(G, T)
 2. **if** (proc == P) **then** $G_{\text{opt}} = G$
 3. **if** (proc > P) **then** LB = T
 4. **if** (proc \leq P) **then** {
 - UB = T
 - if** (T == W_{\max}) **then** {
 - G = Expand nodes with weight W_{\max}
 - $W_{\max} = \text{MaxWeight}(G)$
5. T = Max ($W_{\max}, (LB + UB)/2$)

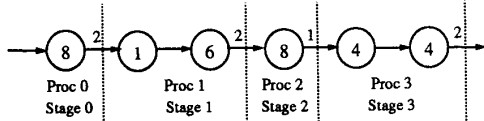
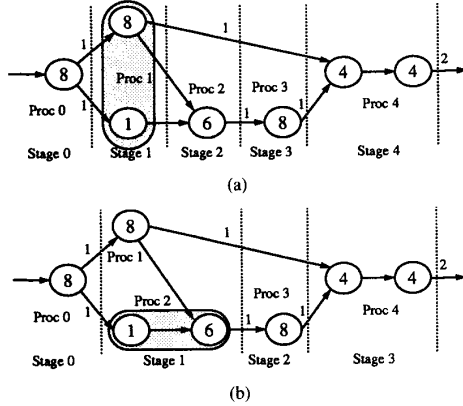
The search repetitively calls a routine *Schedule*(), which determines how many processors are needed to execute the algorithm, given a stagetime T . (This routine also effectively partitions and schedules the flow graph.) The number of processors returned tells whether the proposed T is too small or too large. The LB and UB bounds are updated accordingly, allowing convergence to the minimum feasible stagetime.

When decreasing T , care must be taken to ensure that the maximum granularity condition is not violated. When there are nodes in the graph with weights as large as T , and the search decides that T can be decreased further, it will break up these nodes to find a better solution. We called these nodes *bottleneck* nodes. In this way, large granularity nodes are only decomposed if they block the minimization process, keeping the number of nodes in the graph at a minimum.

The core routine of the search is *Schedule*(). Given the stagetime T , *Schedule*() traverses the graph from input to output, partitioning the graph into stages of pipelines. Nodes are scheduled onto a processor until the total computation costs of the nodes plus the communication cost of output edges exceeds the stagetime T . Once a pipeline is filled, the scheduler proceeds to schedule the remaining nodes on the next pipeline stage. At the end, the graph is partitioned into a number of pipeline stages, and the total number of processors needed is returned. An example of how the scheduler work on a simple sequential flow graph is shown in Fig. 3. Values inside the nodes represent estimates of their computation costs, and values on the edges represent the additional delays for communication (for the sake of simplicity, the time slot model is not used here). When two nodes are assigned to the same processor, communications between them incur no cost. For a sequential flow graph like Fig. 3, the node accumulation is straightforward. To schedule general graphs, it is necessary to resolve how to handle the parallelism available in branching paths. One naive algorithm would be to continue accumulating nodes to fill the stagetime, whether they are parallel or not. This would result in a schedule shown in Fig. 4(a). A more sophisticated algorithm would exploit the parallelism in the graph to yield a schedule shown in Fig. 4(b). This schedule uses the same number of processors but has a smaller latency and communication cost. Exploiting parallelism while pipelining makes the scheduling task much more difficult. Since the number of processors is fixed, not all parallelism can be exploited, and the algorithm must decide which operations deserve extra processors and which do not. The naive approach, on the other hand, does not have to do this as it always puts one processor on a pipeline stage. The exact criteria used for node scheduling is discussed in the next subsection.

A. Node Scheduling

The goal of the node scheduling algorithm is to start the nodes as soon as possible, taking into account communication delays, memory capacity, and processor

Fig. 3. Scheduling a serial flow graph with $T = 10$.Fig. 4. Scheduling a general flow graph with $T = 10$. (a) A naive approach. (b) A better approach.

availability. At any point in the scheduling process, we keep a list of ready nodes \mathfrak{R} and a list of available processors \mathcal{P} . \mathfrak{R} contains all nodes whose input nodes are already scheduled. Initially, it contains all input nodes. \mathcal{P} contains all processors that have been assigned to a pipeline stage, plus an extra processor, called the “new” processor. When a node is scheduled on the new processor, the processor is assigned the pipeline stage appropriate for the node, and another new processor is added to \mathcal{P} . This allows the scheduler to use as many processors as it deems appropriate. The scheduling steps are as follows: $\forall n \in \mathfrak{R}, \forall p \in \mathcal{P}$, we calculate the earliest starting time $\xi(n, p)$. $\xi(n, p)$ is set to K_∞ , a very large constant, if any of the following is true:

- 1) There is insufficient buffer memory in processor p to execute node n , i.e., $bm_{\text{avail}}(p) < bm(n, p)$.
- 2) p was already assigned a pipeline stage which is different from the stage needed to execute n .
- 3) There is insufficient available time left in p to execute n within the stagetime limit.

A processor p for which $\xi(n, p) < K_\infty$ is called a feasible processor for n . Condition 1 assumes the buffer memory is local to the processor. If it is in a centralized memory, the condition would be if $bm_{\text{avail}}(\mathbf{P}) < bm(n, p)$, where $bm_{\text{avail}}(\mathbf{P})$ is the remaining buffer memory of the entire system.

Definition 7: The difference measure $\delta(n)$ is defined as

$$\hat{\xi}(n) = \min \{ \xi(n, p) \mid p \in \mathcal{P} \} \quad (3)$$

$$\hat{p}(n) = \{ p \in \mathcal{P} \mid \xi(n, p) = \hat{\xi}(n) \} \quad (4)$$

$$\bar{\xi}(n) = \min \{ \xi(n, p) \mid p \in \mathcal{P} - \hat{p}(n) \} \quad (5)$$

$$\delta(n) = \bar{\xi}(n) - \hat{\xi}(n). \quad (6)$$

$\delta(n)$ gives a measure of how good the best assignment $\hat{\xi}$ is, compared to the second best $\bar{\xi}$. A node n with a large $\delta(n)$ says that the best assignment is much better than the second best, whereas a small $\delta(n)$ says there exist comparable choices. Thus, it is more urgent to assign nodes with a large $\delta(n)$. As a result, the candidate node and candidate processor are chosen using the following heuristic: The node n , $n \in \mathfrak{R}$, corresponding to the largest $\delta(n)$ is chosen as the candidate node, and the processor where it achieves its earliest starting time is chosen as the candidate processor. In case of a tie, the first node with the largest $\delta(n)$ is chosen. Formally, the candidate node n^* and processor p^* pair is given as

$$n^* = \{ n \in \mathfrak{R} \mid \delta(n) \text{ is maximum} \} \quad (7)$$

$$p^* = \hat{p}(n^*). \quad (8)$$

Table II gives the earliest starting time $\xi(n_i, p_k)$ of three nodes n_0, n_1, n_2 , on three available processors p_0, p_1, p_2 . The $\hat{\xi}(n)$, $\bar{\xi}(n)$, and $\delta(n)$ values are also shown. The earliest starting node is n_0 on processor p_0 . However, the candidate node chosen is n_2 on p_1 since its alternative choice p_2 is much worse. $\xi(n_2, p_0) = K_\infty$, signifying that p_0 is not a feasible processor for n_2 .

Once a candidate pair is chosen, the scheduled node is removed from \mathfrak{R} , and new ready nodes are added. If the new processor in \mathcal{P} was used, another new processor is added to \mathcal{P} . Processors assigned to pipeline stages which are no longer considered are removed from \mathcal{P} to avoid unnecessary computations. $\xi(n, p)$ and $\delta(n)$ values that are affected by the assignment are updated, and the next node-processor pair is chosen. The scheduling algorithm ends when all nodes are scheduled. The pseudocode for `Schedule()` is described below:

Schedule(\mathbf{G}, \mathbf{T}):

1. Input nodes $\rightarrow \mathfrak{R}, p_0 \rightarrow \mathcal{P}$
2. **Repeat while** $\mathfrak{R} \neq \emptyset$
 1. **For each** $n \in \mathfrak{R}, p \in \mathcal{P}$ **do**
Calculate $\xi(n, p)$
Calculate $\delta(n)$
 2. Schedule candidates n^* and p^*
 3. Update $\mathfrak{R}, \mathcal{P}$

The complexity of `Schedule()` is $O(N(N + E))$, and is derived as follows: Let the number of nodes be N , the number of edges be E , and the number of available processors be P . The calculation of $t_{\text{arv}}(n, p)$ takes $O(1)$ time. Given a fixed processor p_k , for each new node n put in \mathfrak{R} , the calculation of $\xi(n, p_k)$ requires calculating $t_{\text{arv}}(n, p_k)$ for each input edge. For the whole graph, all edges are visited, for $O(E)$ calculations. Adding and removing N nodes to and from \mathfrak{R} takes $O(N)$ time. Hence the total computation runs in $O(N + E)$ time. Since P processors are considered for each node, the complexity is $O(P(N + E))$. Finally, for nodes in \mathfrak{R} which were not chosen in a scheduling step, $\xi(n_i, p^*)$ is updated in constant time to reflect the new $t_{\text{ready}}(p^*)$. Since there are at most N such nodes in \mathfrak{R} , at most N $\xi(n, p)$ updates are made each

TABLE II
EARLIEST STARTING TIME

	$w(n_i)$	$\xi(n_i, p_0)$	$\xi(n_i, p_1)$	$\xi(n_i, p_2)$	$\xi(n_i)$	$\bar{\xi}(n_i)$	$\delta(n_i)$
n_0	4	4	8	12	4	8	4
n_1	5	7	16	10	7	10	3
n_2	3	K_∞	6	12	6	12	6

scheduling step. For N scheduling steps, N^2 calculations are required. The total complexity is $O(P(N + E) + N^2)$. Finally, since the algorithm can actually continue to add processors beyond P , the potential number of processors considered can be N , the number of nodes. The final complexity is $O(N(N + E))$.

B. Path Merging

In its attempt to maximize parallelism, the algorithm may under-utilize a processor. This would be the case if, for example, the node assigned to processor 1 in Fig. 4(b) were to have a cost of 1 instead of 8, as shown in Fig. 5. To solve this problem, we can merge these parallel paths so they are forced to execute on 1 processor. This is achieved by adding a dependency edge to serialize the computation, as shown in Fig. 5.

Parallel paths are characterized by branch nodes, i.e., nodes that have two or more output nodes. All parallel paths can be considered by examining only branch nodes.

Definition 8: The overlap path time t_{op} between two parallel paths measures the time both paths are simultaneously active. This is equivalent to the elapsed time between the branch node and the point where either the two paths join again or one of them terminates.

t_{op} is the minimum of the computation costs of the two paths, and it measures the amount of parallelism that exists between the paths. The smaller the t_{op} , the less the parallelism that can be exploited. The pair of parallel paths with the smallest t_{op} is the best candidate for merging as it underutilizes its processor the most. To facilitate the calculation of t_{op} , the transitive closure matrix of the flow graph is used to quickly locate the merging point of paths. In addition, the nodes are leveled from output using their computation costs as weights, allowing the computation cost of a path to be given by the difference between the output levels $ol(n)$ of the branch and join nodes. An $O(N^2)$ algorithm for calculating t_{op} is given as:

OverlapPathTime(G):

For each branch node $n_i \in N$ **do**
For each pair of nodes (n_j, n_k) which are successor nodes to node n_i **do**
 1. Locate join node n_{jk} of (n_j, n_k) as the node with the highest output level which is a descendant node to both n_j and n_k
 2. $t_{op}(n_j, n_k) = \min [ol(n_j), ol(n_k)] - ol(n_{jk})$

After the smallest parallel path is found, a routine *MergePath()* adds a dependency edge from one end of

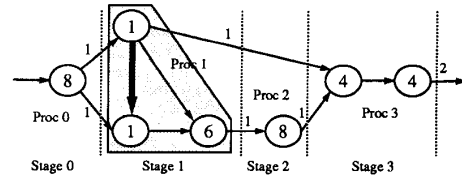


Fig. 5. Path merging to improve processor utilization.

the path to the beginning of the other. t_{op} is then updated taking the new dependency into account. The *Schedule()* routine is repeated along with *MergePath()* as long as it needs more processors than available and as long as there are still parallel paths to merge. Upon exit of the loop, we either have a feasible schedule using P or less processors, or an infeasible schedule ($proc > P$) which uses the minimal number of processors for the given stagetime T . For this latter case, a longer stagetime is needed. The overall routine, called *Partition()*, replaces the *Schedule()* routine in *Main()*:

Partition(G, T):

1. $proc = \text{Schedule}(G, T)$
2. **Repeat while** ($proc > P$ && $\text{MergePath}(G) == \text{TRUE}$)
 1. Update Transitive Closure Matrix, Output Level
 2. $proc = \text{Schedule}(G, T)$

The calculation in the inner loop involves updating t_{op} , finding the minimum t_{op} , adding a dependency edge, and scheduling the graph. Updating t_{op} involves updating the output levels and the transitive closure graph. To update the output levels, only the source node of the newly added dependency edge and its ancestors need to have their output levels recalculated. This takes $O(N)$ time. In the transitive closure graph, an edge has to be added between the source node and its ancestors to each descendant of the destination node. This takes $O(N^2)$ time. Locating the minimum t_{op} takes $O(N^2)$, and scheduling takes $O(N(N + E))$ time. This inner loop repeats at most N times since there are at most $N - 1$ merges possible. Hence, the algorithm runs in $O(N^2(N + E))$ time. For most examples that we have encountered, $E \approx N$, giving an $O(N^3)$ approximation. Fig. 6 shows a sample flow graph and the improvement in processor utilization using path merging. Each node in the graph has a cost of 1.

C. Retiming

The scheduling algorithm presented so far works well for flow graphs which do not contain cycles. For those that do, some enhancements to the algorithm are needed. In this section, a modification to the algorithm is presented to allow the retiming of flow graph cycles to improve scheduling.

Consider Fig. 7(a). In order to execute node A , we need the data d from two samples back in time, which we will denote as $d@2$. If we apply our scheduling strategy to the flow graph without considering the feedback path, we may

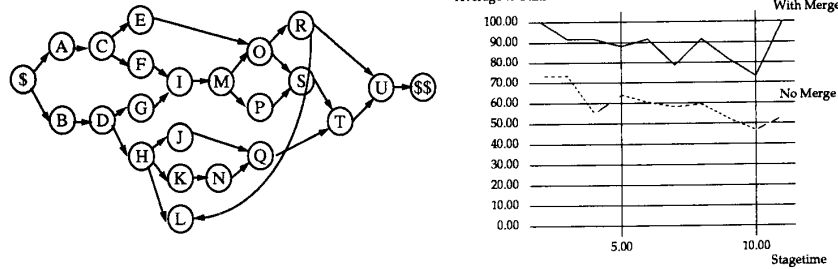


Fig. 6. Improvement in processor utilization due to path merging.

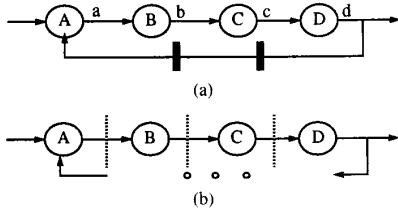


Fig. 7. Retiming flow graphs.

pipeline the computation as shown in Fig. 7(b). Since node D is now three samples behind node A , $d@2$ will not be available when node A needs it. Thus it is necessary to limit the number of pipeline stages on the forward path to ensure that proper data is available when it is needed. Referring to Fig. 7(a), assume node A is applied to sample 0 at time 0. Since the sample period is the stagetime T , node A will be applied to sample two at time $2T$. To execute this node, the output data of node D on sample 0 must be available. This implies that the execution of the nodes A, B, C, D on sample 0 (indeed, on all samples) must complete within time $2T$. In general, if Δ_c is the number of delays in the cycle, then the cycle must execute within time $\Delta_c T$. The following definitions formalize this bound of the execution of cycles.

Definition 9: Let G be the flow graph at some granularity level, and let C be any cycle in G . Let $\Delta_c \equiv$ the number of delays in cycle C , $W_c \equiv$ the total computation time of cycle C , and $E_c \equiv$ the total time a valid schedule would need to execute cycle C .

Theorem 1: For any cycle C ,

$$W_c \leq E_c \leq \Delta_c T. \quad (9)$$

The proof for $E_c \leq \Delta_c T$ is an obvious generalization of the argument above. $W_c \leq E_c$ because a schedule may not necessarily execute cycle C contiguously.

With this result, we get the following condition, called the cycle scheduling bound condition.

Lemma 1.1: Let n_0, n_1, \dots, n_N be the set of nodes in cycle C , and let n_0 be the first scheduled node among them. Let $t_{\text{start}}(n_0)$ be the starting time of node n_0 , and $t_{\text{completion}}(n_k)$ be the completion time of any node n_k in the cycle. A valid schedule must satisfy

$$t_{\text{completion}}(n_k) - t_{\text{start}}(n_0) \leq \Delta_c T. \quad (10)$$

This follows directly from the fact that $t_{\text{completion}}(n_k) - t_{\text{start}}(n_0) \leq E_c$, for every node n_k in the cycle. Lemma 1.1 will be used by the scheduling algorithm to ensure the cycle scheduling bound is satisfied when nodes of a cycle are scheduled.

Lemma 1.2: From Theorem 1, we also have for every cycle c , $T \geq W_c / \Delta_c$. In particular, $T \geq T_{\text{CB}}$, where T_{CB} is the stagetime cycle bound given by

$$T_{\text{CB}} = \max_c W_c / \Delta_c. \quad (11)$$

W_c is a function of the granularity level of the graph. As the granularity gets finer, the computation times of cycles decrease as hierarchical nodes in the cycles are replaced by only those nodes of the subgraph which belong to the cycle. In this case, T_{CB} also decreases. Some cycles are entirely imbedded in hierarchical nodes, and are only uncovered when these nodes are expanded. For that case, T_{CB} may actually increase. Once all cycles are uncovered, however, T_{CB} decreases monotonically to a bound known as the iteration period bound T_{IPB} [19], the stagetime cycle bound for the finest granularity graph. This bound is the minimum achievable latency between sample iterations, and is a theoretical lower bound for our solution.

When the scheduler makes a partition on the forward path, it is in effect putting a logical delay there. To maintain correct functionality of the flow graph, it is necessary to remove a delay in the feedback path. This can be interpreted as moving delays in the cycle around to maximize throughput, a concept known as retiming [13]. By choosing $T \geq T_{\text{CB}}$, one might conjecture that sufficient time is allocated to each pipeline stage to guarantee that at most Δ_c partitions are made in scheduling cycle c , thereby automatically adhering to the cycle scheduling bound. This, unfortunately, is only guaranteed if the nodes in the cycle are scheduled contiguously, the stagetime is exploited completely, and communication delays are not considered. In practice however, data transfers are often present, and using the stagetime completely is difficult due to the granularity of the nodes. An example of a cycle scheduling violation due to large granularity of the nodes is shown in Fig. 8. From Fig. 8(a), the stagetime cycle bound T_{CB} is found to be $30/3 = 10$. Using a stagetime $T = T_{\text{CB}} = 10$, we find that we need four partitions, which violates the cycle scheduling bound condition (Fig.

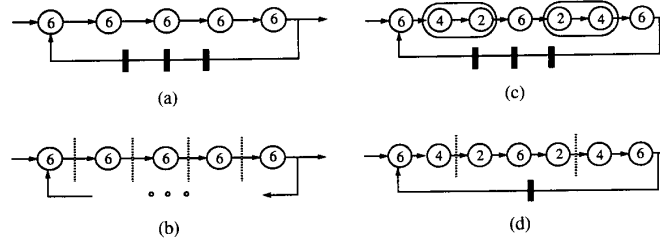


Fig. 8. Partitioning cycles.

8(b)). The weight of the nodes are too big to fit in the remaining available time, leaving holes in the stagetime, which we called slacks. If the nodes are broken down into smaller nodes, as in Fig. 8(c), a feasible partition with no slacks can be obtained (Fig. 8(d)). Communication costs are not considered in this example.

Using the analysis above, the scheduling algorithm presented so far can easily be modified to handle cycles. From Lemma 1.2, the stagetime lower bound LB is modified to $\max(W_{\text{total}}/P, T_{\text{IPB}})$, to include the theoretical lower bound due to cycles. Each time a node n_i in a cycle is scheduled in `Schedule()`, Lemma 1.1 is used to ensure that the cycle scheduling bound for the node is satisfied. If it is violated, the large nodes in the cycle are decomposed in an attempt to minimize slacks in the stagetime. This is repeated until there is no more violation or no more nodes in the cycle can be decomposed. In the latter case, LB is immediately updated to the current T , as no feasible solution can be found at any granularity level for this stagetime T . This extension to `Partition()` is called `CyclePartition()`, and is given as:

CyclePartition(G,T):

1. (proc, ViolateFlag) = `Partition(G,T)`
2. **Repeat while** (ViolateFlag == TRUE && `IsGraphFlat(G) == FALSE`)
 1. G = Expand nodes in critical cycles
 2. Find all cycles in G
 3. Update W_{max} and T_{CB}
 4. (proc, ViolateFlag) = `Partition(G,T)`

In this routine, `Partition()` is called repetitively on finer and finer granularity flow graphs to satisfy the cycle scheduling bound. When the node decomposed in the cycle is an iteration, each decomposition increases the number of processors allocated to the iteration by 1, and terminates when the number of processors reaches P . Thus, `Partition()` repeats at most P steps when decomposing iterations. When decomposing functions, it may repeat more, and there is no maximum bound. Benchmarks show that the typical number of function node decompositions is less than five. The main computations in each pass is the search for all cycles in the flow graph, and the `Partition()` routine. For cycle detection, we use Johnson's algorithm for finding all the elementary cycles of a directed graph [8], which is the fastest algorithm known. It has a time of complexity of $O((N + E)(C + 1))$, where C is the number of cycles in the graph. From

the last section, `Partition()` was analyzed to run in $O(N^2(N + E))$ time. Hence, the running time of `CyclePartition()` is $O(P(N + E)[N^2 + (C + 1)])$.

The updated main algorithm is shown below. Note that T_{CB} is updated each time a node is decomposed, and the proposed stagetime T is modified to include the T_{CB} lower bound.

Main():

1. Assign computation times to nodes
2. LB = Lower bounds on T, UB = Upper bounds on T
3. $W_{\text{max}} = \text{MaxWeight}(G)$
4. $T_{\text{CB}} = \text{CycleBound}(G)$
5. **repeat while** (LB < UB)
 1. `CyclePartition(G,T)`
 2. **if** (proc == P) **then** $G_{\text{opt}} = G$
 3. **if** (proc > P) **then** LB = T
 4. **if** (proc ≤ P) **then** {
 - UB = T
 - if** (T == W_{max}) **then** {
 - G = Expand nodes with weights W_{max}
 - Update W_{max} and T_{CB}
 - if** (T == T_{CB}) **then** {
 - G = Expand nodes in critical cycles
 - Update W_{max} and T_{CB}
 5. T = Max ($W_{\text{max}}, T_{\text{CB}}, (LB + UB)/2$)

D. Node Decomposition

Node decomposition is the means by which the scheduling algorithm traverse from a coarse grain flow graph to a finer grain flow graph. In previous subsections, we saw that nodes are broken up on two occasions: When they are larger than the available stagetime T , and when they are part of a cycle that violates the cycle scheduling bound. The first case is denoted as bottleneck node decomposition and the second case as critical cycle decomposition. There are two types of hierarchical nodes that are candidates for decomposition: Function nodes and iteration nodes. Function nodes are decomposed by replacing the nodes with their subgraphs. Iteration nodes are replaced by parallel or serial subnodes (depending on the data dependencies between iterations), each computing a

subset of the iteration range. The method of breaking up the iterations differs between the decomposition of bottleneck nodes and the decomposition of critical cycles. In both cases, the size of the subnodes is determined dynamically during the scheduling procedure.

When decomposing bottleneck iteration nodes, the subnodes adapt to the available time remaining in the stage. The number of iterations assigned to the subnodes are not fixed until the first subnode is scheduled. At that time, the first subnode is assigned as many iterations as can fit in the remaining stage, and subsequent subnodes are assigned as many iterations as can fit in a new, empty stage with stagetime T . This partitioning strategy allows the iterations to float across processors from one pass of the scheduler to the next to fit the changing stagetime T . When decomposing iteration nodes in critical cycles, the goal is to obtain a partition which satisfies the cycle scheduling bound condition. For nodes with serial dependency, decomposition does not decrease the cycle computation time, but may improve the stagetime slacks. Therefore, the same technique as discussed above is used. For nodes with parallel dependency, decomposition actually decreases the computation time of the cycle, making it easier to meet the cycle scheduling bound.

Consider a cycle containing a parallel iteration node as shown in Fig. 9. Assuming a tight stagetime $T = T_{CB} = 35$ and no communication delays, scheduling Fig. 9(a) with $t_{\text{start}}(n_A) = 0$ would yield $t_{\text{completion}}(n_E) = 2 * 35 + 10 = 80$. Since $t_{\text{start}}(n_A) + 2T = 70$, the cycle scheduling bound is violated. In Fig. 9(b), after decomposition, the stagetime cycle bound is reduced from 35 to 25. T is no longer tight, and scheduling yields $t_{\text{completion}}(n_E) = 1 * 35 + 20 = 55$, $t_{\text{start}}(n_A) + 2T = 70$, meeting the cycle scheduling bound condition. Of course, the price we pay is the two additional processors needed to execute the iterations in parallel. Note that the naive scheduling algorithm described in Fig. 4(a) would not be able to improve the stagetime at all, as the parallelism of the iteration is not exploited.

The goal in the parallel iteration node case is to parallelize the node just enough to satisfy the cycle scheduling bound condition, as excessive parallelizing would only use more processors than necessary. To accomplish this, the parallel iteration node is incrementally divided into equal weight subnodes until a partition with no violation or the maximum decomposition is reached. Since the cycle scheduling bound is tightest on the subnode with the maximum computation, a division into equal weight subnodes maximizes the chance of meeting the bound.

In our approach, loops can only be divided at the boundary of each iteration. No attempt will be made, for instance, to partition 3.5 iterations of a loop on one processor and the remainder on another.

VII. RESULTS

The first example is a histogram algorithm to be mapped onto a Sequent multiprocessor. The input is an array of 128 samples, to be partitioned into 32 subclasses. The

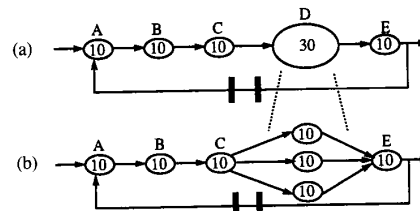


Fig. 9. Decomposing parallel iterations in cycles.

computation proceeds as follows: First, the minimum and maximum values of the array are determined. Next, the width of each subclass is calculated from the total range. The third task fills in all the subclasses. This is a parallel task as each subclass can scan the array and count how many numbers fall into its subclass. The fourth task determines the baseline class, the subclass whose range covers 0. Finally, the last task computes the number of samples in the five subclasses around the baseline class. Fig. 10 shows the flow graph partitioned onto 6 processors of the Sequent. The computation was pipelined into two stages, with processor 0 in stage 0 and processors 1–5 in stage 1. Analyzing the completion times of the processors as estimated by the scheduler and as actually measured on the Sequent (Fig. 11), we see that the load was evenly distributed, and the estimated time agrees very well with the actual running time. The same example was then scheduled and executed on different numbers of processors to assess how the speedup varies with the processor count. Fig. 12 shows the speedup as calculated by the scheduler, and the actual speedup on the Sequent, both as functions of the processors. For this example, we observe that the scheduler is able to consistently achieve a faster throughput with each additional processor. For comparison, the ideal speedup is shown. This can only result if there is perfect load balancing and there is no cost for interprocessor communication.

The second example is a Cordic algorithm mapped onto the SMART multiprocessor. It converts Cartesian to polar coordinates iteratively in 20 steps. It takes as input an (X, Y) coordinate, as well as an array of correction angles. After an initialization, there is a sequential loop of 19 iterations. The scheduler is able to achieve good speedup by pipelining the loop and assigning successive loop iterations to successive processors. In contrast to the histogram example, the Cordic program is communication intensive. A linear array architecture thus can execute this program more efficiently than a single shared bus architecture. Fig. 13 plots the speedup for the Cordic example. The “stair case” effect of the speedup curves results from the fact that the scheduler does not partition loops at the middle of an iteration. As a result, when there are 12 to 18 processors available, the scheduler still has to assign at least 1 processor 2 iterations to execute. The throughput remains constant until enough processors are available to allow each iteration to be executed by its own processor. This occurs at 20 processors, after which a great leap in throughput is attained. At this same time, the commu-

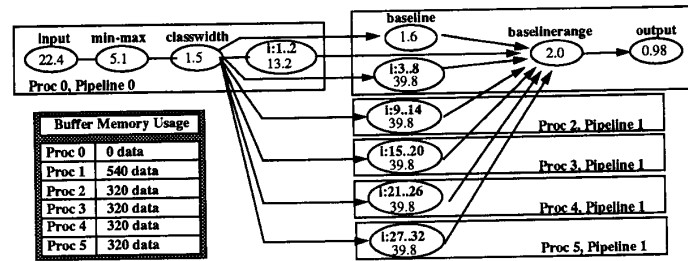


Fig. 10. Histogram example scheduled on 6 processors.

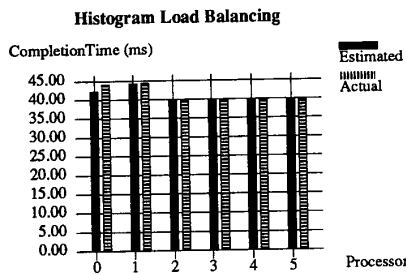


Fig. 11. Load balancing on 6 processors.

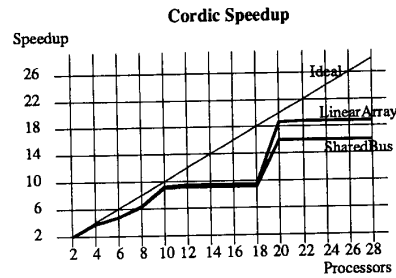


Fig. 13. Speedup versus processors.

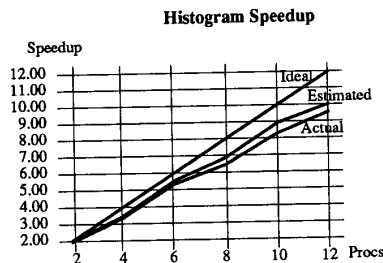


Fig. 12. Speedup versus processors.

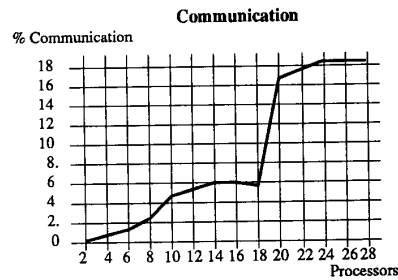


Fig. 14. Percent communication versus processors.

nication load on the bus also increases dramatically as all processors are communicating (Fig. 14). The linear array architecture can support this neighbor-to-neighbor communication well, yielding a better speedup than a shared bus architecture.

The scheduling algorithm has also been tested on a number of other DSP examples with different types of concurrency and communication patterns. Table III shows the performance of the algorithm on these examples. All assume that the number of available processors is eight, and a single shared bus interconnection. $F(G)$ and $H(G)$ give the number of nodes in the flattened and hierarchical graph, respectively. $R(G)$ gives the number of nodes considered by the scheduler during its top-down search. The # Pipelines shows the amount of pipelining and parallel execution. The CPU measurements are on a Sun Sparc II, and include the Silage to flow graph compilation.

The dynamic time warp (DTW) algorithm [17] for speech recognition processes 1000 templates to compute the score of each template with respect to an unknown signal. The matrix multiplication example multiplies two 64×64 matrices. The DFT algorithm computes a 256-pt

discrete Fourier transform in two nested loops. The pitch extractor [21] derives a candidate pitch from an input signal and compare it against 20 templates. The 2-norm squares and accumulates each element of a vector (size 128). All of these examples are computationally intensive and require little interprocessor communications. Hence, good speedup was achieved. An exception is the DCT example, which is communication intensive (51% communication overhead on the shared bus).

Table IV shows three additional examples: The adaptive differential pulse code modulator (ADPCM), the decision feedback equalizer (DFE), and the echo canceller [7]. All contained single delay recursions in their computation, which prohibited pipelining. Nevertheless, they contain parallel iterations in the cycle, which allow for parallel execution. In the ADPCM and DFE examples, the stagetime obtained was close to the cycle bound, although the speedup was nowhere near 8. Faster speedup for these examples can only be obtained by transforming or reorganizing the algorithm themselves. In the echo canceller case however, eight processors were not enough to exploit all the parallelism in the cycle. As a result, each

TABLE III
SCHEDULING RESULTS

Example	$F(G)$	$H(G)$	$R(G)$	Concurrency	# Pipelines	Speedup	CPU (sec)
DTW	1.7e8	98	15	Pipe/Par	2	7.08	30.2
Mat. Mult	2.0e6	24	11	Pipe/Par	2	6.64	5.8
256pt DFT	7.6e5	35	9	Pipe/Par	3	6.94	8.2
Pitch Extractor	1.2e5	270	22	Pipe/Par	3	7.53	21.7
2-Norm	1926	23	12	Pipe/Par	4	7.61	11.2
8pt DCT	87	62	75	Pipe/Par	5	3.26	89.3

TABLE IV
SCHEDULING RESULTS

Example	Concurrency	Stagetime	Cycle Bound	Speedup
ADPCM	Retime/Par	1381.5	1362.8	3.72
DFE	Retime/Par	1500.8	1238.5	4.22
Echo Canc	Retime/Par	12157.0	6649.0	2.34

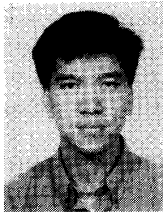
processor allocated to the parallel iteration still has to perform a number of iterations sequentially. In this case, additional processors will allow for a further reduction of the stagetime.

VIII. CONCLUSION

A flow graph scheduling algorithm that simultaneously considers pipelining, retiming, parallelism, and hierarchical node decomposition is presented. The ability to simultaneously consider the many types of concurrency allows the scheduler to find efficient multiprocessor solutions for a wide range of DSP applications. It has been implemented as part of a software environment for scheduling DSP programs onto fixed and configurable multiprocessor systems. The results on a set of benchmarks demonstrate that the algorithm still achieves near ideal speedups across programs with different types of concurrency.

REFERENCES

- [1] S. Bokhari, "Assignment problems in parallel and distributed computing," in *Parallel Processing and Fifth Generation Computing*, Kluwer, 1988.
- [2] M. L. Campbell, "Static allocation for a data flow multiprocessor," in *Proc. 1985 Int. Conf. Parallel Processing*, 1985, pp. 511-517.
- [3] P. Hilfinger, "SILAGE, a high level language and silicon compiler for digital signal processing," in *Proc. IEEE CICC Conf.*, Portland, May 1985.
- [4] P. Hoang and J. Rabaey, "Program partitioning for a reconfigurable multiprocessor system," presented at the IEEE Workshop on VLSI Signal Processing IV, Nov. 1990.
- [5] P. Hoang and J. Rabaey, "A compiler for multiprocessor DSP implementation," presented at the ICASSP, Mar. 1992.
- [6] P. Hoang, "Compiling real-time digital signal processing applications onto multiprocessor systems," Ph.D. dissertation, Univ. California, Berkeley, 1992.
- [7] M. L. Honig and D. G. Messerschmitt, *Adaptive Filters: Structures, Algorithms, and Applications*. Kluwer, 1984.
- [8] D. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM J. Computing*, vol. 4, no. 1, Mar. 1975.
- [9] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Trans. Comput.*, vol. C-33, no. 11, Nov. 1984.
- [10] W. Koh, A. Yeung, P. Hoang, and J. Rabaey, "A configurable multiprocessor system for DSP behavioral simulation," presented at the ISCAS Symp., May 1989.
- [11] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, Jan. 1987.
- [12] S. H. Lee, C. J. M. Hodges, and T. P. Barnwell, III, "An SSIMD compiler for the implementation of linear shift-invariant flow graphs," presented at the ICASSP, Mar. 1985.
- [13] C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing synchronous circuitry by retiming," in *Proc. Third Caltech Conf. VLSI*, 1983, pp. 23-26.
- [14] K. K. Parhi, "Rate-optimal fully-static multiprocessor scheduling of data-flow signal processing programs," presented at the ISCAS Symp., May 1989.
- [15] C. Polychronopoulos and U. Banerjee, "Speedup bounds and processor allocation for parallel programs on multiprocessors," in *Proc. 1986 Int. Conf. Parallel Processing*, 1986.
- [16] H. Printz, "Automatic mapping of large signal processing systems to a parallel machine," Ph.D. dissertation, Carnegie-Mellon Univ., May 1991.
- [17] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-26, pp. 43-49, 1978.
- [18] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors* (Research Monographs in Parallel and Distributed Computing). Cambridge, MA: M.I.T. Press, 1989.
- [19] D. A. Schwartz, "Synchronous multiprocessor realizations of shift-invariant flow graphs," Ph.D. dissertation, Georgia Institute of Technology, June 1985.
- [20] G. Sih and E. A. Lee, "A multiprocessor scheduling strategy," ERL Int. Rep., Univ. California, Berkeley, 1990.
- [21] R. J. Sluyter, N. J. Kotmans, and A. V. Leeuwen, "A novel method for pitch extraction from speech and a hardware model applicable to vocoder systems," in *ICASSP*, 1980, pp. 45-48.



Phu D. Hoang was born in Saigon, Vietnam, on January 21, 1965. He received the B.S. degree in electrical engineering and mathematics in 1985 from the University of Maryland, College Park, and the M.S. and Ph.D. degrees in electrical engineering and computer science in 1987 and 1992, respectively, from the University of California, Berkeley.

His research interests include multiprocessor scheduling and compilation, high-level synthesis, and design-exploration frameworks. He is currently with Redwood Design Automation, engaged in research and development of high-level design automation tools.



Jan M. Rabaey (S'80-M'83) was born in Veurne, Belgium, on August 15, 1955. He received the E.E. and Ph.D. degrees in applied sciences in 1978 and 1983, respectively, from the Katholieke Universiteit, Leuven, Belgium, where he worked on computer-aided design tools for switched capacitor circuits.

From 1983 till 1985, he was associated with the University of California, Berkeley, as a Visiting Research Engineer, where he developed an automated synthesis system for multiprocessor DSP architectures. From 1985 till 1987, he directed the Architectural and Algorithmic Strategies Group of the Design Methodologies for VLSI System Division at IMEC, Belgium. In 1987, he joined the faculty of the University of California, Berkeley, as an Assistant Professor. His current interests are the computer-aided analysis and automated design of digital signal processing circuits and architectural synthesis.