Code Generation: On the Scheduling of DAGs Using Worm-Partition

Hatem M. El-Boghdadi
Computer Engineering Dept.
Cairo Univ., Giza, EGYPT
helboghdadi@eng.cu.edu.eg

Mohamed Bohalfaeh
Computer Science Dept.
Cairo Univ., Giza, EGYPT
bohalfaehm@yahoo.co.uk

Abstract

Code generation consists of three main stages, instruction selection, scheduling and register allocation. The scheduling stage is very important because it affects the execution time of resulting code as well as the associated memory space needed to store the program. This paper deals with scheduling directed acyclic graphs (DAGs) using worm-partition. First, we develop a new algorithm to partition DAGs into a collection of worms while ensuring that the worm-partition is legal. Although the algorithm does not guarantee the minimum number of worms, it runs in optimal $O(|V| + |E|)$ time. This is in contrast to the known method [4] for producing the minimum number of worms that runs in $O(|V|^2 + |V||E|)$. We apply the algorithm to benchmark real problems and show its comparable results to the previous method. Then we study some DAG properties that are related to worm partitioning. We derive a necessary condition for the minimum number of worms in a given DAG. In other words, a lower bound for the number of worms. Then we identify two important classes of DAGs, for which this necessary condition is sufficient as well; i.e. we show that the lower bound is a tight one. Finally, we show that our algorithm generates the minimum number of worms for theses classes of DAGs.

1 Introduction

Digital signal processors (DSPs) are special-purpose microprocessors that are designed to perform extensive arithmetic computations. Often in real time environment writing applications in assembly language have been the dominant DSP programming convenient. However, writing these applications in high level languages is becoming more desirable.

To meet the constraint of the specialized applications, DSPs have different features including multiple memory banks and multiple buses to support higher memory bandwidth and uses a separate address generation unit. Also, these chips have dedicated address registers to provide more parallism, limited number of registers and one or more accumulators to support arithmetic and logic operations.

Thus, it is essential to produce code of the highest quality that is achievable in a reasonable amount of time. In general, code generation consists of three phases; instruction selection, scheduling and register allocation. Constructing the schedule takes place after instruction selection and register allocation are done. The ordering of the instructions will cause some data transfer between allocated registers and memory unit(s) hence the number of these data transfer should be minimized for real time processing [2]. Therefore, scheduling is very important not only because it affects the execution time of the resulting code but also it determines the associated memory space needed because the registers and memory have critical capacity [4]. Thus, designing an efficient method to schedule the control flow of the instructions is a must. This paper studies the problem of scheduling directed acyclic graphs (DAGs) using worm-partition; in which each vertex in the DAG under consideration corresponds to some computation and each edge represents a dependence or precedence relationship between computations.

For optimal code the instructions must be scheduled in such way that no memory spills are introduced. One important scheduling algorithm [4] uses worm-partition to partition the DAG into a set of legal worms by finding the longest legal worm in the DAG in each iteration (worm partitioning is defined formally in Section 2). The advantage of using worm-partition is to minimize the number of data transfers, and to enable memory spills free schedule. In this paper we present an new algorithm to partition the DAG into a set of legal worms. We show that the algorithm has optimal time complexity. We also study some of the properties of DAGs that are related to worm partitioning which gives us a better understanding about the partitioning problem.

Aho and Johnson [1] showed that even for one register machine, generating optimal code is NP complete. They showed that the absence of cycles among the worms of in a worm-partition graph is a sufficient condition for the legality of the worm-partition. They presented optimal code generation algorithm for one register machines, and they defined that a worm-partition of a DAG is a set of disjoint worms. They also showed that the optimal code generation problem is NP-complete on two-address machines, even when the number of registers is unlimited.

Liao [5, 6] used clauses with adjacency variables to describe the all legal worm-partitions. He applies binate covering formulation to find optimal scheduling. He presented a new theorem of code generation for the non-commutative one register machine, based on a compact binate covering formulation. He defined two reasons to make the worm-partition illegal; namely *reconvergent paths* and *interleaved sharing*, and he showed that for a worm-partition to be legal, the vertices of each worm should appear consecutively in the schedule. However, his work does not provide a constructive algorithm to worm partitioning DAGs.

Hong [4] proposed and evaluated a new algorithm to construct a legal worm-partition while keeping the number of worms generated as small as possible. He introduced an additional source vertex S successively in each stage of his algorithm to prevent including interleaved shared vertices in the worms, and to handle interleaved sharing in the same way as reconvergent paths. He showed that there is no reason to consider the reconvergent paths and interleaved sharing which make a worm-partition illegal as distinct cases. However his algorithm has a complexity of $O(|V|^2 + |V||E|)$.

This paper studies the problem of scheduling DAGs using worm partitions. First, we develop a constructive algorithm to solve the scheduling problem to control flow of the instructions based on worm partitioning of a binary DAG. The algorithm is time optimal where it runs in $O(|V| + |E|)$ steps. Although our algorithm does not guarantee the number of generated worms to be minimal, its performance is comparable to the algorithm presented by Hong [4]. The algorithm has been tested on well-known benchmarks and gives a comparable performance on the average. Then we study some DAG properties that are related to worm partitioning. We derive a necessary condition for the minimum number of worms in a given DAG. In other words, a lower bound for the number of worms. Then we identify two important classes of DAGs namely *Reduction DAGs* and *Broadcasting DAGs* for which the necessary condition is sufficient as well; i.e. we derive a tight lower bounds for the number of worms for these classes of DAGs. Finally, we show that our algorithm guarantees the minimum number of worms for these classes of DAGs.

The next section presents some definitions and preliminaries. In Section 3 we present our algorithm for worm partitioning. The experimental results are given in Section 4. Section 5 derives a necessary condition for the minimum number of worms in DAGs. Sections 6 and 7 prove that the necessary condition is sufficient as well for two important classes of DAGs. For these two classes of DAGs, Section 8 shows that the algorithm generates the minimum number of worms. Section 9 gives some concluding remarks and directions for future work.

2 Background

In this section we introduce some definitions that will be needed in the rest of the paper. A Directed Acyclic Graph (DAG), $G(V, E)$, consists of $|V|$ vertices and $|E|$ edges with no cycles among vertices. Each vertex, $v \in V$, represents a computation. Each edge, $e \in E$, represents dependence or precedence relation between computations. Figure 1 shows a DAG with 6 Vertices and 7 edges. The DAG is a binary one where the *in-degree* and the *out-degree* of each vertex is at most 2. Vertices v_5 and v_6 are *leaves* of the DAG since the *in-degree* for each of them is zero.

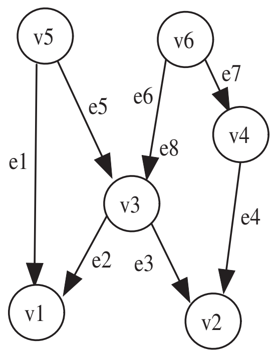
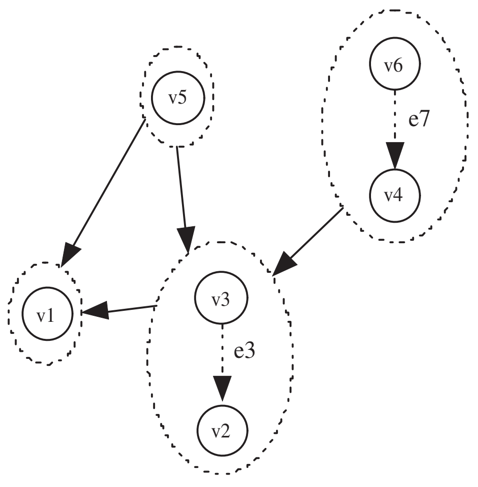


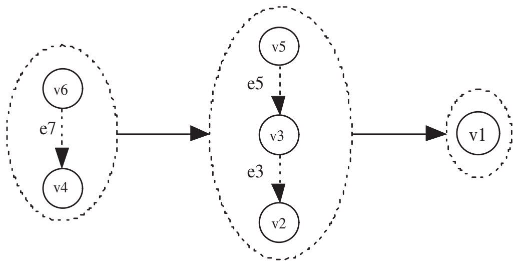
Figure 1. An Example of directed acyclic graph.

A *worm*, $w = \{v_1, v_2, .., v_k\}$ in a directed acyclic graph $G(V, E)$ is a directed path of G such that the vertices $v_i \in w$ ($1 \le i \le k$ and $1 \le k \le |V|$) are scheduled to execute consecutively. A *worm-partition*, $\mathcal{W} = \{w_1, w_2, w_3, .., w_m\}$ of $G(V, E)$, is a partitioning of the vertices V of the graph into disjoint sets w_i, $1 \le i \le m$, such that each w_i is a worm. The worm-partition is said to be *legal* if a valid schedule can be derived from G such that the vertices of each worm appear consecutively in the schedule [5]. Figure 2 shows examples of legal worm-partition while Figure 3 shows an illegal worm-partition (because of the existence of a cycle among the worms in the worm-partition graph).

As shown in Figures 2 and 3, we denote the worm by dot



(a)



(b)

Figure 2. Examples of legal worm-partition.

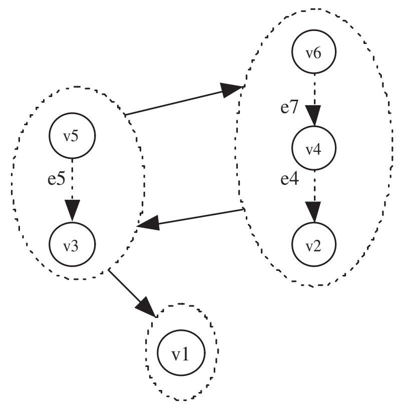


Figure 3. Example of illegal worm-partition.

ellipse shape, the relation between the vertices inside the worm by a dot line, and the relation between the worms by a straight line.

If there are two or more distinct paths from vertex A of a DAG to vertex B in that DAG, then these paths are said to be *reconvergent* paths. Also vertex B is a *reconvergent* vertex [4]. As in Figure 1 there are two paths from v_6 to v_2. The first path passes through vertex v_3 and the second path passes through vertex v_4. Hence those two paths are reconvergent paths. Also vertex v_2 is a reconvergent vertex. If the vertex has two parents and is not a reconvergent vertex, then it is called *shared* vertex. If the vertex is either a shared or reconvergent then it is called a *bug* vertex.

For two vertices A and B in a DAG, if there is a path from A to B and there is a direct edge from A to B then the edge (A, B) is said to be *reconvergent* edge. As shown in Figure 1 there is path from v_5 to v_1 passes through vertex v_3 and there is direct edge from v_5 to v_1. This edge is a reconvergent edge.

3 The New Worm-Partition Algorithm

In this section, we present a new algorithm for worm-partition. Although our algorithm does not guarantee to partition the DAG into the minimum number of worms, the algorithm is time optimal. We also show that the algorithm runs in $O(|V| + |E|)$ time. This is in contrast to the algorithm presented in [4] that runs in $O(|V|^2 + |V||E|)$. Section 4 gives some experimental results using benchmark real problems and a comparison to previous methods.

Generally speaking, the algorithm consists of three main phases. The first phase applies depth first search (DFS) to the DAG $G(V, E)$ to collect data about each vertex; i.e. its parents (p_1, p_2) and its children (c_1, c_2). The second phase uses the data collected in the first phase to discover the leaves of the DAG. The third phase partitions the DAG into a legal worm-partition. The DFS algorithm gives the priority for right child to be selected in a worm during the search. For that reason we denote the right child as c_1 and the left child as c_2. Section 3.1 gives a high level description of the algorithm. Also, it derives its time complexity.

3.1 Description Of The Algorithm

In this section we gives a high level description of the algorithm. Figure 4 gives the pseudo code of the algorithm. As mentioned before, the algorithm consists of three main steps as follows:

- *Phase 1: Data Collection*: this phase applies the DFS to the DAG $G(V, E)$ to collect data about each vertex. The collected data consists of the two immediate predecessors vertices (right and left parent), p_1, p_2, and the two immediate successors vertices (right and left child), c_1, c_2. The values of p_1, p_2 (resp. c_1, c_2) could be a ϕ representing that the vertex has no right or left parent (resp. child).

- *Phase 2: Leaves Discovering*: this phase discovers the leaf vertices; i.e. vertices with no incoming edges. This is done by checking if both parents of the vertex, p_1, p_2 have a value of ϕ. Construct a set L that contains the leaves of the DAG.
- *Phase 3: Worm Partitioning*: the phase repeats till the list L is empty. Each time it constructs a worm in the worm-partition graph. It starts with a vertex in L and uses the DFS to construct a worm. Let v_d denote the last vertex added to the worm (initially v_d is a leaf selected from L). The worm is constructed according to the following cases:
  (a) If v_d, has no immediate successor, then the worm will be ended.
  (b) If v_d has only one immediate successor c_1 then
    (i) If c_1 has only one incoming edge then add c_1 to the worm.
    (ii) If c_1 is a bug vertex (have another parent) then change the parent ID for c_1 to ϕ instead of v_d. The worm will be ended not including c_1.
  (c) If v_d has two immediate successors c_1, c_2 then
    (i) If c_1 and c_2 have another parent (bug vertices), then change parent ID for both c_1, c_2 to ϕ instead of v_d. The worm will be ended not including neither $c1$ nor c_2.
    (ii) If either c_1 or c_2 is a bug vertex and the other is not, then the non bug vertex will be added to the worm, and change the parent ID for bug vertex to ϕ instead of v_d.
    (iii) If c_1, c_2 are not bug vertices then c_1 will be added to the worm, and c_2 will be added to L.

■

Since the only way that causes cycles is due to reconvergent vertices [5], the algorithm does not include any reconvergent vertex when constructing a worm. This makes the generated worm-partition graph a legal worm-partition.

Algorithm Complexity: Now we show that the algorithm runs in optimal time of $O(|V| + |E|)$ steps.

Lemma 1 *The algorithm visits each vertex in the binary DAG at most twice.*

Proof: The algorithm deals with binary DAGs which contains three types of vertices:

- **Type 1:** Vertex with no incoming edges: a leaf. The algorithm selects each leaf exactly once; i.e. the vertex is visited only once.
- **Type 2:** Vertex with one incoming edge. If the vertex is selected in a worm the first time visited, then it is visited once. If the vertex is not selected the first time visited, it becames a leaf vertex; i.e. the vertex is visited at most twice.
- **Type 3:** Vertex with two incoming edges: a reconvergent vertex. The algorithm does not include the vertex in the worm the first time the vertex is visited. However, it includes it in a worm the second time visited; i.e. the vertex is visited exactly twice. ■

Phase 1 applies the DFS and runs in $O(|V| + |E|)$ steps. Phase 2 checks the parents of each vertex and runs in $O(|V|)$ steps. Phase 3 constructs the worm-partition graph and by Lemma 1, it runs in $O(|V| + |E|)$ steps. Thus, we have the following result.

Theorem 2 *The algorithm correctly partitions a binary DAG into a legal worm-partition in $O(|V| + |E|)$ steps.* ■

4 Experimental Results

In this section we present some experimental results of our algorithm. The algorithm was implemented and applied to randomly generated DAGs (Table 1) as well as to benchmark real problems (Table 2) from the digital signal processing domain (i.e. DSPstone) [7] and from high-level synthesis [3].

Table 1 shows the results when applying the algorithm to randomly generated DAGs. Sizes of the DAGs were varied from 50 nodes to 500 nodes. For each DAG size, one hundred DAG were generated randomly. The first column gives the number of nodes. The second column shows the average number of worms obtained when applying the algorithm. The third column gives the average ratio of number of worms to the number of nodes. In other words, how much the graph is reduced. The fourth and the fifth column shows the best and worst ratios respectively.

Table 2 shows the results and a comparison to the algorithm presented by Hong [4]. The third column shows the number of worms obtained in the worm-partition graph when applying our algorithm (denoted as Ours) and when applying the algorithm in [4] (denoted as Hong [4]). The table shows a matching in the results. The fourth column gives the ratio of the number of worms to the number of vertices. Also a comparison is given between our algorithm and previous methods.

According to the results obtained, our algorithm has the same performance when applied to the benchmark real problems as the algorithm presented in [4] that looks for the longest worm first. This is in addition to the optimal running time of our algorithm of $O(|V| + |E|)$ instead of $O(|V|^2 + |V||E|)$ of Hong [4].

```
Main Procedure
Input: $G(V, E)$ where $V$ is a set of vertices and $E$ is a set of edges.
Output: $\mathcal{W} = \{w_i, i \leq |V|\}$.
Begin
1-   Apply DFS to $G(V, E)$ to collect data for each vertex
2-   Call $Discover_Leaves(G(V, E), Leaves)$
3-   $i = 0$
4-   While ($Leaves \neq \phi$)
5-     $w_i = \phi$
6-     Let $u$ be first element in the $Leaves$ set
7-     $Leaves = Leaves - u$
8-   $DFS_Worm(u)$
9-     $w_i = w_i + u$
10-      If $(u(c_1)\&u(c_2)) = true$ then
11-       if $c_1\&c_2$ have another parent then
12-         $c_1(u)\&c_2(u) = \phi$
13-         $\mathcal{W} = \mathcal{W} \cup w_i$
14-       elseif $c_1$ has not another parent and $c_2$ has another
          parent then
15-         $c_2(u) = \phi$
16-         $DFS_Worm(c_1)$
17-       elseif $c_1$ has another parent and $c_2$ has not another
          parent then
18-         $c_1(u) = \phi$
19-         $DFS_Worm(c_2)$
20-       elseif $c_1$ and $c_2$ have not another parent then
21-         $Leaves = Leaves + c_2$
22-         $DFS_Worm(c_1)$
23-       Endif
24-      elseif $u$ has only one child then
25-       if $c_1$ has another parent then
26-         $c_1(u) = \phi$
27-         $\mathcal{W} = \mathcal{W} \cup w_i$
28-       elseif $c_1$ has no another parent then $DFS_Worm(c_1)$
29-       End if
30-      elseif $u$ has no child then $\mathcal{W} = \mathcal{W} \cup w_i$
31-      Endif
32-      $i = i + 1$
33-   End
34-   End While
35-   $Discover_Leaves(G(V, E), Leaves)$
36-      $Leaves = \phi$
37-      For each $v \in G(V, E)$
38-       If $(v(p_1) and v(p_2)) = \phi$
39-         Then $Leaves = Leaves + v$
40-   End
End
```

Figure 4. Pseudocode for the partitioning algorithm.

Table 1. Results when applying the algorithm to randomly generated DAGs.

$\|V\|$	Ave. $\|W\|$	Ave. Ratio $\|W\|/\|V\|$	Best Ratio	Worst Ratio
50	15.07	0.3014	0.18	0.4
100	30.86	0.3086	0.24	0.37
200	61.49	0.30745	0.26	0.345
300	92.78	0.3093	0.276	0.34
500	156.11	0.3122	0.28	0.348

Table 2. Results when applying the algorithm to benchmark real problems.

Benchmark	$\|V\|$	$\|W\|$		Ratio $\|W\|/\|V\|$	
		Ours	Hong [4]	Ours	Hong [4]
AR-Filter	28	12	12	0.4286	0.4286
FDCT	42	20	20	0.4762	0.4762
DIFFEQ	11	5	5	0.4545	0.4545
SEHWA	31	16	16	0.5161	0.5161
F2	22	7	7	0.3182	0.3182
DOG	11	5	5	0.4545	0.4545

5 Lower Bound On The Number Of Worms

In this section we study some of the properties of the binary DAG. This would give us a better understanding about the partitioning problem. We first derive a necessary condition for the minimum number of worms in a binary DAG (a lower bound). Next we identify two important classes of DAGs for which the above necessary condition is sufficient (tight bound) as well.

Let $\mathcal{D}_N$ be a binary DAG of N nodes. Let the level of node x in the DAG be $\ell evel(x)$. Assume that the leaves of the DAG is at a level 0; i.e. $\ell evel(x) = 0$ if x is a leaf. Let $m(\mathcal{D}_N)$ be the number of levels in $\mathcal{D}_N$. Clearly, the maximum number of levels in $\mathcal{D}_N$ is N levels.

Let the width of $\mathcal{D}_N$ at level i be $width(\mathcal{D}_N)_i$. The width, w, of $\mathcal{D}_N$ is the maximum width of a level in $\mathcal{D}_N$) where $w = \max\{width(\mathcal{D}_N)_i\}$, $0 \leq i \leq m(\mathcal{D}_N)$. Now we derive a necessary condition for the minimum number of worms of $\mathcal{D}_N$.

Lemma 3 *For any $w > 1$, a directed acyclic graph (DAG) can be partitioned into w worms only if the DAG has a width of at most w.*

Proof: Let $\mathcal{D}_N$ denote a binary DAG of N nodes. We prove that if $\mathcal{D}_N$ has a width of $w + 1$ nodes then the DAG cannot be partitioned into w worms. The existence of one level of nodes in $\mathcal{D}_N$ with $w + 1$ nodes implies that each node of the $w + 1$ nodes should be in a different worm. Thus the worm-partition would have at least $w + 1$ worms. ■

The necessary condition of Lemma 3 applies to any DAG. However, in general, it is possible for a width-w DAG to require more than w worms. Now we define two classes of DAGs for which the above necessary condition is sufficient as well.

6 Reduction DAGs

In this section we consider a particular class of DAGs and prove that this class can be partitioned into w worms if the width of the DAG is w. Consequently, the necessary condition of Lemma 3 becomes sufficient as well.

Definition 1 A *reduction* binary DAG is a one in which each node has *in-degree* ≤ 2 and *out-degree* ≤ 1. ■

Figure 5 (a) shows an example of a reduction DAG. The benchmark problems DIFFEQ and SEHWA (shown in table 2) are two examples of reduction DAGs. These DAGs are from the domain of DSP and High-Level synthesis.

Consider any width-w reduction DAG, $\mathcal{D}_N$, of N nodes. Clearly, every subset of $\mathcal{D}_N$ also is a reduction DAG. To prove that $\mathcal{D}_N$ can be partitioned into w worms (when the width of $\mathcal{D}_N$ is greater than 1), we only need show the existence of a worm w_1, $w_1 \subseteq \mathcal{D}_N$, that reduces the width of $\mathcal{D}_N$ by 1. We now show that the worm w_1 can be constructed.

Lemma 4 *For any connected reduction DAG, $\mathcal{D}_N$, and for any two levels i and j where $i \leq j$, $width(\mathcal{D}_N)_i \geq width(\mathcal{D}_N)_j$. Moreover, $\mathcal{D}_N$ has only one output node.*

Proof: Since the *out-degree* of each node in the reduction DAG is at most 1, then each node in level i will be connected to at most one node in level $i+1$. If each node in level i is connected to a distinct node in level $i+1$, then $width(\mathcal{D}_N)_j = width(\mathcal{D}_N)_i$. If there exists two nodes in level i connected to the same node at level $i+1$ ($in-degree$ ≤ 2), then these two nodes have been reduced to one node in level $i+1$ and $width(\mathcal{D}_N)_j < width(\mathcal{D}_N)_i$.

From the definition of the reduction DAG, two nodes in the same level could be connected in a lower level. If the reduction DAG is connected (i.e. all nodes are connected), then the DAG has only one output node. Otherwise, the DAG can be dealt with as separate DAGs. ■

The following result is a simple to derive from Lemma 4.

Corollary 5 *For any connected reduction DAG, $\mathcal{D}_N$, the width of $\mathcal{D}_N$ is the number of nodes in level zero; i.e. the number of leaves.* ■

Theorem 6 *Any connected reduction DAG $\mathcal{D}_N$, can be partitioned into w worms if and only if the width of $\mathcal{D}_N$ is at most w.*

Proof: To prove the theorem, we only need show the existence of worm $w_1 \subseteq \mathcal{D}_N$ where $\mathcal{D}_N - w_1$ is of width $w-1$.

We proceed by induction on the width of $\mathcal{D}_N = w$. By Corollary 5, the width of $\mathcal{D}_N$ is the number of leaves of the DAG. Clearly, when $w = 1$, all the nodes could be included in one worm. Assume the theorem to hold for a DAG with $w = H$ leaves and consider a DAG with $H + 1$ leaves.

Construct the worm w_1 of width 1 as follows. Step 1 selects any leaf, ℓ in w_1. This reduces the width of level 0 nodes by 1. For Step 2, if the child c of ℓ has only one input then select c in w_1. This guarantees that width of level 1 nodes is reduced by 1 and Step 2 is repeated. If the child c of ℓ has two inputs then dont include c in w_1 and w_1 is ended. This is because the $level$ of c has a width $\leq H$ and by Lemma 4 all lower levels has a width $\leq H$. This guarantees that $\mathcal{D}_N - w_1$ is of width H. By the induction hypothesis the width H DAG can be partitioned into H worms. Thus the total number of worms of for a DAG of width $H+1$ is $H+1$ worms proving the theorem. ■

7 Broadcasting DAGs

In this section we consider another class of DAGs that can be partitioned into w worms if the width of the DAG is w. Consequently, the necessary condition of Lemma 3 becomes sufficient as well.

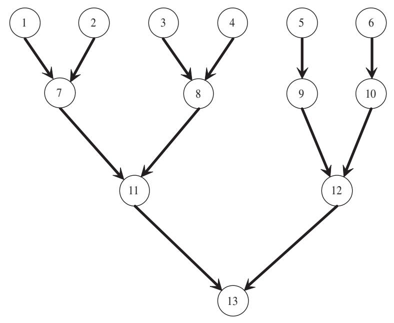
Definition 2 A *broadcasting* binary DAG is a one in which each node has *in-degree* ≤ 1 and *out-degree* ≤ 2. ■

Figure 5(b) shows an example of such DAG. Consider any width-w broadcasting DAG, $\mathcal{D}_N$, of N nodes. Clearly, every subset of $\mathcal{D}_N$ also is a broadcasting DAG. To prove that $\mathcal{D}_N$ can be partitioned into w worms, we only need show the existence of worm $w_1 \subseteq \mathcal{D}_N$ that reduces the width of the remaining DAG by 1. The proofs in this section are analogous to the proofs of Section 6. Thus in this section we only give the final results without the actual proofs.

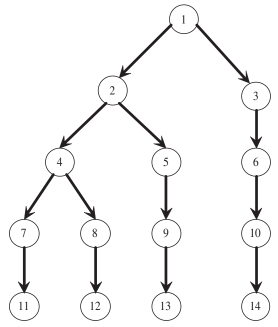
Lemma 7 *For any connected broadcasting DAG, $\mathcal{D}_N$, and for any two levels i and j where $i \leq j$, $width(\mathcal{D}_N)_i \leq width(\mathcal{D}_N)_j$. Moreover, $\mathcal{D}_N$ has only one input node.* ■

Corollary 8 *For any connected broadcasting DAG, $\mathcal{D}_N$, the width of $\mathcal{D}_N$ is number of output nodes.* ■

Theorem 9 *Any connected broadcasting DAG, $\mathcal{D}_N$, can be partitioned into w worms if and only if the width of $\mathcal{D}_N$ is at most w.* ■



(a)



(b)

Figure 5. (a) Example of a reduction DAG. (b) Example of a broadcasting DAG.

8 Applying The Algorithm To Reduction & Broadcasting DAGs

In this section we show that the algorithm presented in Section 3 generates the minimum number of worms when applied to either a reduction or a broadcasting DAG.

Theorem 10 *The algorithm partitions any connected reduction or broadcasting DAG, $\mathcal{D}_N$, into w worms if and only if the width of $\mathcal{D}_N$ is at most w.*

Proof: To prove that the algorithm partitions a width-w DAG into w worms, we need only show that the algorithm reduces the DAG width by 1 when it constructs a worm.

First, consider $\mathcal{D}_N$ to be a reduction DAG where by Definition 1, the $out\text{-}degree$ of each node is at most 1. The algorithm starts a worm with a leaf, v_d, and includes all the subsequent vertices till a bug vertex, v_r, is reached. This reduces the width of all levels ($\ell evel(v_d)$ to $\ell evel(v_r) - 1$) by 1. Since $width(\mathcal{D}_N)_i < width(\mathcal{D}_N)_{\ell evel(v_r)-1}$, $i \geq \ell evel(v_r) - 1$, then the width of $\mathcal{D}_N$ is reduced by 1 proving the theorem for the reduction DAG.

Consider $\mathcal{D}_N$ to be a broadcasting DAG where by Definition 2, the $in\text{-}degree$ of each node is at most 1. The algorithm starts a worm with a leaf, v_d, and include all the subsequent vertices till an output vertex is reached. This reduces the width of all DAG levels by 1 proving the theorem. ■

9 Concluding Remarks

In this paper we have proposed a new worm partitioning algorithm for DAGs while maintaining the legality of scheduling. Although the algorithm does not guarantee to produce the minimum number of worms, its performance was shown to be comparable to previous methods when applied to benchmark real problems. The proposed algorithm was shown to be time optimal where it runs in $O(|V| + |E|)$ time. This is in contrast to the known method [4] for producing the minimum number of worms that runs in $O(|V|^2 + |V||E|)$.

Also the lower bound problem for the number of worms in DAGs has been studied. We derived a lower bound for the number of worms for any connected DAG. We also showed that for some classes of DAGs in the field of DSP and High-Level synthesis, this lower bound is tight. We also showed that our worm-partitioning algorithm generates the minimum number of worms for these classes of DAGs.

This work can be extended in several directions. One direction is to improve the algorithm to guarantee the minimum number of worms while maintaining the optimality of the algorithm. Other directions include studying other classes of DAGs for which the lower bound is tight as well.

References

[1] A. Aho, S.C. Johnson, and J. Ullman, "Code Generation for Expressions with Common Subexpressions," *Journal of the AMC*, 24(1), pp. 146-160, 1997.

[2] G. Araujo, "Code Generation Algorithms for Digital Signal Processors," *PhD thesis Princeton Department of EE*, June 1997.

[3] G. De Micheli, "Synthesis and Optimization of Digital Circuits," *McGraw-Hill,* 1994.

[4] J. Hong, "Memory Optimization Techniques for Embedded Systems," *PhD thesis, The Department of Electrical and Computer Engineering, LSU, USA,* August 2002.

[5] S. Liao, "Code Generation and Optimization for Embedded Digital Signal Processors," *PhD thesis, MIT Department of EECS*, January 1996.

[6] S. Liao, K. Keutzer, S. Tjiang, and S. Devadas, "A new viewpoint on code generation for directed acyclic graphs," *ACM Transactions on Design Automation of Electronic System*, 3(1):51-75, January 1998.

[7] V. Zivojnovic, J. Velarde, and C. Schlager, "DSPstone: A DSP-oriented benchmarking methodology," *Proc. 5th International Conference on Signal Processing Applications and Technology*, October 1994.