HW/SW Codesign Incorporating Edge Delays Using Dynamic Programming

Karthikeyan Bhasyam Kia Bazargan
ECE Department
University of Minnesota
Minneapolis, MN 55455
{karthik, kia} @ece.umn.edu

Abstract

We present an algorithm based on dynamic programming to perform the HW/SW partitioning and scheduling of a given task graph for minimum latency subject to resource constraint. The major contribution of this paper is to consider the edge communication delays in the dynamic programming solution of the problem. The algorithm has a polynomial run time complexity on trees. We also introduce a pruning technique to reduce the runtime of the worst-case scenario of directed acyclic graphs (DAGs). The algorithm has been implemented and the results are reported. A very fast quality heuristic is also proposed and implemented to provide good solutions in negligible run time.

1. INTRODUCTION

The hardware/software (HW/SW) co-design problem involves the allocation of operations in a data flow graph (DFG) to computing resources to optimize a particular optimization objective. This gives rise to several conceptually related problems in codesign depending on the objective being optimized and the modeling of the resources and operations in the data flow graph. The two popular problems are latency minimization for given resource constraints and resource minimization with a given latency constraint. HW/SW codesign is a very important step in the design flow and greatly influences the performance of the final design. Thus efficient and high quality solutions are required to the HW/SW codesign problem.

There has been extensive research in the area of HW/SW codesign. Jantsch et al proposed a dynamic programming solution for latency minimization subject to resource constraints [1]. They map the problem to a knapsack problem and propose a dynamic programming solution to meet resource constraints. However, they do not perform any graph traversals, i.e., they do not consider the interaction between different nodes but consider the speed up obtained by mapping an operation in HW (FPGA, ASIC) independently. This does not account for the resource binding for the fanin and fanout nodes of an operation. A dynamic programming approach is proposed by Knudsen et al [2] where they move resources to HW until the resource constraints are met without performing any graph traversal.

Bender et al [12] proposed a MILP based model, which has high runtime complexity. Several approaches ([4], [5], [10]) depend on greedy heuristics, code profiling etc, which assign operations to resources greedily. Chang et al propose a dynamic programming solution to the codesign problem [3]. They use the classic dynamic programming approach, which has been used extensively for problems not only in codesign but also in buffer insertion, technology mapping etc. The solution proposed is extensive and is used for codesign of communication systems. However they target area minimization given latency constraint and do not have resource constraints. Also their approach is for coarse-grained HW/SW partitioning while this work targets fine-grained HW/SW partitioning. More importantly they do not consider delays associated with edges in the task graph between two processes and hence their approach is different from the problem formulation in this work. In embedded system design the communication delay between operations especially those placed in different partitions is significant. Hence incorporating these communication costs in the dynamic programming framework is essential. In this paper we propose a dynamic programming approach to perform the resource constrained HW/SW partitioning for minimum latency considering these edge delay costs.

2. PROBLEM FORMULATION

A task is defined by its Data Flow Graph, which is represented by a DAG. The DAG consists of a set of operations and communicating edges between operations.

A set of possible implementations for each operation in HW and SW is also given. Also the communication between two operations incur a delay depending on the partitions in which the two communicating nodes are placed. The resource constrained minimum latency problem is to assign the operations to HW or SW and schedule them for minimum latency subject to a finite HW resource constraint.

The task graph represents the high level system to be implemented. It consists of a set of nodes and a set of edges and is represented by *G=(V, E)* where V is the set of nodes and E is the set of edges. Each node represents an operation (ADD, MULTIPLY, MEMREAD, etc.) that has a specific cost of implementation and speed of execution on HW, which could be HW (e.g., FPGA, ASIC) and SW (e.g., general-purpose processor, DSP chip).

The edges in the data flow graph have communication delays depending on the partitions in which the two nodes incident on the edge are present. The delay can take different values depending on whether the two nodes are present in SW/SW or SW/HW or HW/HW. The resource constrained scheduling problem is intractable as shown in [11]. However the dynamic programming solution provides good solution quality for some special case problems in polynomial time as shown in this paper. A highly effective heuristic is also provided for task graphs that have exponential runtimes in the dynamic programming solution.

3. DYNAMIC PROGRAMMING

Our method considers the directed acyclic dependency graph (DAG) in a bottom-up fashion. Partial solutions of subtrees are kept as non-dominating lists and merged as the graph is traversed towards the primary outputs. For each node, a list of non-redundant delay/resource usage pairs is kept, and used in forming the non-dominating solutions of the successor nodes.

A solution S for node *n* is defined as the pair (T, L) where T is the time at which *n* is scheduled to complete and L is the amount of finite resource used for this solution (e.g., amount of memory or functional units). Each solution also corresponds to the node being placed either in HW/SW. *This solution is obtained by combining the solution sets of all its fanin nodes.* Thus given a solution for a node *j* mapped to HW or SW we can obtain the time step at which it is scheduled and resource assignment of every node in the fanin cone of this node. Thus each solution for a node corresponds a particular HW/SW partitioning of the sub-graph rooted at node *j*. The load then corresponds to the amount of HW used for this particular partitioning solution of the sub-graph. If a node is placed in SW then its cost is zero since HW partition has finite resource constraint while the SW partition is assumed to have infinite resources. Each node has a solution set which corresponds to a set of solutions of the type described above. The solution set can be represented as a load delay curve with the corner points being the non-inferior points. A queue of nodes is created by performing a topological sort on the graph and the nodes are processed in that order to obtain their load-delay curves. The topological sort ensures that primary inputs are processed first and the solution sets for the nodes are by combining the solution set of its fanin nodes. When the sink of the DAG is reached the solution set obtained represents all the non-inferior solutions to this task graph. The solution with minimum latency is selected from this set and represents our final (optimum) solution. If the original given task graph has multiple primary outputs (*PO*) i.e. sinks then a dummy node is added to the graph and edges are created between this dummy node and all the primary outputs. The cost of implementation and delay of this dummy node in both HW and SW is assumed to be zero. Further the edges connecting this dummy node to the original primary outputs are assigned zero delays. The topological sort is then performed on this modified task graph.

3.1 MERGE OPERATION

Merge operation is the process of forming a solution of a node by combining the solution sets of its fanin nodes. The merge operation is described below. For the purpose of illustration we assume the task graph is a balanced tree with no re-convergent fanouts. Consider a node A with two parents B and C as shown in Figure 1. Let S_b and S_c be the solution sets (delay-load curves) for the two fanin nodes. Also let the HW resource constraint be L_{max}

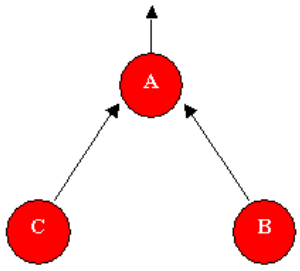

Figure 1. Merge Operation

Initially assign node A to the HW partition. Let S_n^i be the i^{th} solution in the solution set for node *n*. Then for a solution say S_b^1 (T_b, L_b) in S_b find the resulting schedule time T_a at A i.e. the time at which A gets the input from this fanin node (B) by adding the edge delay between A and B. So $T=T_b+$ (*edge delay between A and B*). The value of the edge delay between A and B depends on the partition in which B is placed corresponding to the solution

point S^1_b. This is a valid schedule time at A if there is a solution in S_c (T_c, L_c) such that:

$$T_c + \text{edge delay between } A \text{ and } C \leq T. \quad (1)$$
$$L_c + L_b + \text{cost of implementation of A} \leq L_{max} \quad (2)$$

If there is no such solution in S_c then T_a is not a feasible solution for A. If more than one such solution exists in C then the solution with minimum load is chosen. Then the total load used by this solution L_a is $L_b + L_c$+ cost of implementation of A in HW[1]. The time at which A completes operation is T_a=T + delay of node A in HW. So one of the possible solutions for A is (T_a L_a). This process is performed for all solution points of B and C. Thus we obtain a solution set for node A when it is assigned to HW partition. This process is again repeated by now assigning node A to the SW partition. Note that now node A does not contribute to the final load as it is placed in SW and only affects the delay of the schedule, including possible communication delays. Thus we obtain the complete solution set for A which consists of two load-delay curves corresponding to HW or SW implementation of node A. Whenever a solution is formed for node A it is checked for non-inferiority with the current solutions of A. If it is non-inferior, it is added to the solution set of A, else discarded.

The most important property of this process is that if the solution set of B has *p* solutions and solution set of C has *q* solutions then each load-delay curve for A can have at most *(p+q)* solutions, which results in polynomial time complexity for trees and most practical cases of DAGs.

3.2 EDGE DELAYS

The presence of edge delays requires reformulation of feasibility constraint as shown in (1) with respect to the constraint in [3]. More importantly it requires a different pruning approach to be used in the merge operation. In the *absence of edge delays*, a solution S_1 (T_1, L_1) of a node *j* is said to be inferior to another solution S_2 (T_2, L_2) of the same node if it meets the following constraint, used by some previous works:

$$T1 \geq T2 \;\&\&\; L1 \geq L2 \quad (3)$$

We show that this condition can lead to pruning of optimal solutions and potentially result in significant quality degradation. The main issue that pruning condition (3) misses is the consideration of edge delays: it does not require that S_1 and S_2 be in the same partition. S_1 may be a solution when *j* is placed in HW while S_2 may correspond to a solution when *j* is placed in SW. The following example illustrates why this could lead to quality degradation.

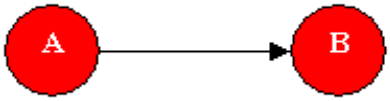

Figure 2. Edge Delay

Assume nodes A and B are operations of different types in Figure 2. Types here refer to functionality of the node say ADD, MULT, SHIFT etc. Let the edge delay for communication between HW and SW be 10. The edge delay for communication between SW and SW or HW and HW is 1. Note that time steps are counted from zero so if a node is a primary input and has a delay of 2 in HW then it will finish its operation at time step 1. Let the HW capacity constraint be 10 units

Table 1. Delay and Cost of Operations

Node	Delay in HW	Delay in SW	Cost in HW
A	3	3	2
B	2	7	2

Table 1 gives the delay and cost of implementing each node in SW/HW. Note that the cost of implementing in any node in SW is zero since only HW has resource constraint

Table 2. Solution Sets for nodes

Solution for A	Solution for B
(2,2) A is in HW	(5,4) B is in HW
	(19,2) B is in SW
(2,0) A is in SW	(14,2) B is in HW
	(10,0) B is in SW

Table 2 gives the solution set formed without any pruning. Since A is a primary input the solution set of A is just the set of possible implementations for A. For each of these solutions of A the corresponding solution for B is calculated assuming it is placed in HW and in SW. This is the exhaustive solution set for both A and B

[1] Note that adding these three terms as presented here assumes no resource sharing is allowed. However, resource sharing can be added to this formulation at increased runtime cost by keeping track of resource usages at each node and combining the schedules of the parent nodes.

Table 3. Solution sets of nodes after pruning

Incorrect Pruning (A)	Incorrect pruning (B)	Correct Pruning (A)	Correct Pruning (B)
(2,0) in SW	(10,0) in SW	(2,2) in HW	(5,4) in HW
		(2,0) in SW	(10,0) in SW

Table 3 gives the solution set of nodes by incorrect and correct pruning. The incorrect pruning uses condition (3) to prune solution (2,2) of A. As a result, when solutions of B are formed, A is always implemented in SW. Hence, the optimal solution for B, which is (5,4), is never generated. To avoid such cases, we use an additional condition when pruning a solution. A solution S_1 is said to be inferior to S_2 if it satisfies (*3*) *and* if both S_1 and S_2 correspond to node *j* being placed in HW (SW).

3.3 RECONVERGENT FANOUT

The data flow graph (DFG) of a task is represented by a directed acyclic graph (DAG). The DAG may have nodes with reconvergent fan-out. The presence of reconvergent fanout imposes a constraint in addition to (1) and (2) during the merge operation. Figure 3 shows node A with reconvergent fanouts.

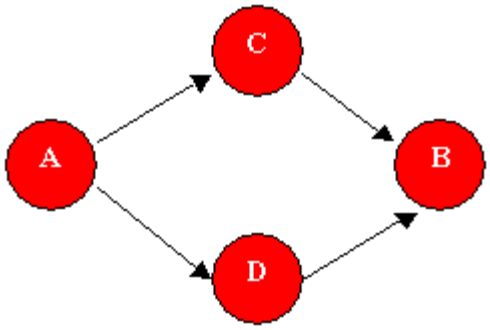

Figure 3. Reconvergent fanout

During *merge* when solutions of parent nodes are combined it should be made sure that the solutions being combined are consistent for any common parent nodes. For example in Figure 3 solutions of nodes D and C are combined to form the solution set for node B. Let a solution of D result in node A being placed in a particular partition, e.g., HW. To ensure consistency then a solution from C, which will be added to this solution, should also result in node A being placed in HW. Otherwise for this particular solution of B, node A will be assigned to conflicting partitions.

This problem is taken care of by the use of status vectors in the data structure for each solution similar to the solution proposed in [4]. The vector is of size *n* where *n* is the number of nodes. When a node is being processed during the merge operation, vectors of the solutions being added are checked for consistency. As already discussed, each solution for a node *j* corresponds to a HW/SW partition solution for the sub graph rooted at *j*. Each bit in the vector of a solution for node j corresponds to a node in the graph. If node *k* is placed in SW for this solution then the k^{th} entry is set to 0 and if it is placed in HW for this solution the entry is set to 1.If the k^{th} node is not present in the sub-graph rooted at node *j* then the entry is set to the default value of –1.

Consider the example shown in Figure 3. Here a solution from D and C are combined if and only if all entries that are not default (-1) in both vectors are the same. This ensures that a node, which is present in solutions of both parents, can be combined if and only if it is in the same partition in both solutions.

The reconvergent fanout also affects the pruning algorithm for nodes on *converging paths* in two ways: it could lead to pruning an optimal solution, or it could lead to no feasible solution being found.

To illustrate the case where no feasible solutions can be found, consider nodes C and D in Figure 3. If only the conditions of Section 3.2 are imposed, then it is possible that all solutions of C correspond to node A being placed in SW and all solutions of D correspond to node A being placed in HW. So when the solutions of C and D are combined to form the solution set for B, there will be no feasible solution for B. This situation arises because reconvergent fanout creates dependence between solution sets of nodes C and D. In the absence of reconvergent fanout, for a given time step the solution corresponding to minimum load can be chosen independently from each of C and D. The same is not true when converging paths exist in the graph.

To show how an optimal solution might be lost, consider two solutions S_1 and S_2 of node C, both corresponding to C being implemented in SW, but one placing A in HW and the other implementing A in SW. If we prune out one of these solutions (which would have been legal according to Section 3.2), then we might lose the optimal solution when we merge solutions of C with the solution set of D. So, when considering two solutions of C for pruning, not only C has to be implemented in the same partition (Section 3.2), but also A has to be in the same partition in these two solutions (converging paths). The same is true for all nodes on a path diverging from A. This has runtime ramifications that will be discussed in Section 6. If a node is on converging paths of more than one node, all such parent nodes (with reconvergent fanouts) put restrictions on pruning of the node. Consider a

node *j* which is on a converging path of node *k*. Consider a solution S_2 which is about to be added to the solution set of *j*. S_2 is inferior to some solution S_1 of *j* if and only if it meets the *additional* pruning constraint given below. Vector1 and Vector2 are the status vectors of S_1 and S_2 respectively.

For all i such that i is a reconvergent fanout parent node

Vector1[i]==Vector2 [i] or Vector1[i]==-1 or Vector2[i]==-1 (4)

The constraint is imposed to avoid pruning an optimal solution and also avoid situations where no feasible solution can be obtained (e.g., all solutions from C correspond to A mapped to HW, and all solutions of D correspond to A placed in SW). Identifying reconvergent fanout nodes and intermediate and final sink nodes can be handled during the topological sort process. The vector for a node is built from the vectors of its parent nodes during the merge process.

4. THE ALGORITHM

The pseudo code of our algorithm is presented below

```
dynamic (G){
    if G has multiple PO's add dummy PO and covert to a
            single PO DAG
    q=reverse topological sort(G)
    while (q not empty) {
        node=pop(q)
        merge(node) //form the load-delay curve        }
    node=PO
    S_best=minimum latency solution for node
    For i in 1 to number of nodes
        if S_best.vector[i]==0
            Node i is placed in SW
        else
            Node i is placed in HW
    The HW resource used is L_best and the latency is T_best
}
```

Figure 4. Dynamic programming algorithm

The algorithm implements the dynamic programming solution discussed in the earlier section. *merge* is a routine that implements the basic addition of load-delay curves of a parent. The pruning algorithm is embedded within the merge routine and every solution is checked for non-inferiority through this pruning algorithm before being added to the node's solution set. From the solution set of PO we choose the solution with minimum latency as our final solution. The HW/SW partitioning solution is available from the vector corresponding to the best solution for the PO. If no resource sharing is allowed (e.g., in partitioning memory arrays between HW and SW), the solution obtained is the optimal for resource constrained HW/SW partitioning problem for minimum latency. When resource sharing is considered, the problem is NP-hard.

5. HEURISTIC ALGORITHM

It is also possible to combine the basic dynamic programming algorithm with greedy heuristics to obtain a very efficient and very fast HW/SW partitioning heuristic algorithm. Running time analysis shows that the presence of reconvergent fan out greatly impacts the running time complexity. This is because the minimum load from each parent's solution for a required time step cannot be found independently. Hence several possible combinations among the sub solutions need to be explored to identify the sub solutions which result in minimum load for a given time step. However the fanin of most nodes is usually 2~3 and so we need to explore several possible combination for at most 2 sub solutions and this does not add much in run time complexity in most cases. However a fast approximate algorithm, which gives good results, is helpful to designers to evaluate the effectiveness of a proposed partitioning solution. Further when the number of fanin nodes is very large (which is not true for most practical cases) the heuristic algorithm helps us to find a good solution in negligible time.

The heuristic combines dynamic programming approach to build the solution set bottom up. However when a reconvergent fan out node is encountered, the sub solutions are chosen independently without any consideration of reconvergent fan out. If the chosen sub solutions result in source of reconvergent fan out being assigned to conflicting partitions, it is then assigned to the partition dictated by the sub solution with minimum slack (difference in ASAP and ALAP start times).

For example in Figure 3 let a solution S^1 of D and a solution S^2 of C be combined to form a solution for the node B. Let S^1 correspond to node A being placed in HW and S^2 correspond to the node A being placed in SW. Then A is greedily assigned to HW or SW depending on which of the two sub solutions S^1, S^2 is most timing critical. The heuristic guarantees negligible run time by removing the interdependency between sub solutions, which arises due to reconvergent fan out. Once the solution set for sink node is obtained the solution corresponding to minimum latency is taken and the HW/SW partitioning solution for the entire graph is obtained. From the HW/SW partitioning solution the time step at which each node is scheduled is recalculated and obtained.

6. TIME COMPLEXITY

The dynamic programming algorithm has polynomial time complexity for trees and some DAGs even though theoretical running time is exponential for worst case inputs (where the number of reconverging paths is comparable to V, the number of nodes). However we did not encounter long runtimes for any of the practical cases of DAGs we tried. The time complexity of the algorithm is analyzed as follows. For now let us focus on trees only. Let the number of possible implementations of each type of operation be K. V is the number of nodes in the tree. Let the depth of the tree that is the maximum number of nodes in a path from PI to PO be D. Consider a node A with parents B and C. Let the size of the solution set of B and C be p and q respectively. Then for each possible implementation of A we have at most $p+q$ solutions if we do pruning using only (1). So the total number of solutions for A will be $K*(p+q)$, many of which will be pruned out when we use (3). Since the depth of the tree is D then the number of solutions at the PO will be $O(K^D*N)$ where N is a polynomial number of solutions in terms of V, number of nodes in the graph. So the run time complexity in worst case is $O(K^D*N)$. Even though theoretically D can be linear in terms of V, it is usually in the order of $\log V$ for most practical circuits. Finally when pruning is considered combining p and q solutions results in far fewer than the worst-case $p+q$ solutions. Since we considered only HW/SW for each node, K is 2 and the running time is $O(2^{\log V} * N)$ which is $O(V N)$ where N is polynomial in the number of nodes. So the run time complexity of the algorithm is polynomial for trees.

When considering general DAGs, the dynamic programming algorithm cannot prune intermediate solutions as freely, and might have to keep exponential number of solutions to make sure optimality is not lost. In the example of Figure 3, node C cannot prune out a solution implementing A in SW in favor of another solution implementing A in HW. Hence, it has to keep two separate lists of intermediate solutions, one for A in HW, and another for A in SW. If the number of nodes with reconvergent fanouts is not a constant, then the runtime becomes exponential. However, this did not happen in any of the practical cases that we tried. Furthermore, the heuristic algorithm has polynomial runtime, as it greedily prunes out solutions, which might lead to inferior solutions.

7. EXPERIMENTAL RESULTS

The algorithm proposed in this work was implemented in C++ and executed on an INTEL PENTIUM 4 machine running on Windows XP with 1 GB of memory and processor speed of 2 GHz. We used a set of benchmark circuits from related publications to test the running time of the algorithm. The results are presented in Table 4. Column 1 gives the name of the bench mark circuit, column 2 gives the number of nodes and edges in the circuit, column 3 gives the amount of HW resources used for this schedule column 4 gives the run time and column 5 gives the latency of the schedule. All benchmarks were run with the following specifications and parameters. Two types of operations were assumed namely memory access operations and arithmetic operations. All primary inputs and primary outputs of a DFG represent memory access operations while all the other nodes in the DFG were assumed to be arithmetic operations. No resource sharing optimization within a partition was done. The edge communication delay was assumed to be zero if the two nodes were placed in the same partition and assumed to be 5 when placed in different partitions. For arithmetic operations the cost of implementation in HW (SW) was 2 (0) while the delay of the operation in HW (SW) was 2 (4). For memory access operations the corresponding values were 1(0) and 1(2). The HW resource constraint was assumed to be 30 units. Diff is taken from [6], bender is taken from [12], xilinx is a filter benchmark from [7], yen is taken from [8], pedram is Figure 13.a from [3] and phoneme is taken from [9].

Table 4. Experimental results for the proposed algorithm

Name	Nodes/Edges	HW used	Run time (s)	Latency
diff	21/24	22	0.062	25
bender	12/14	19	0.062	18
xilinx	28/38	26	1.218	29
yen	6/5	10	0.015	5
pedram	12/15	21	0.015	14
phoneme	23/22	27	0.046	18

In Figure 5 the minimum latency of the DFG for various HW capacity constraint is shown. It can be seen that as HW capacity constraint is tightened (decreased) the latency to execute the DFG increases. This is because more operations (nodes) are assigned to SW resulting in larger execution delay for those operations. The run time showed minimal or no change for different HW capacity constraints.

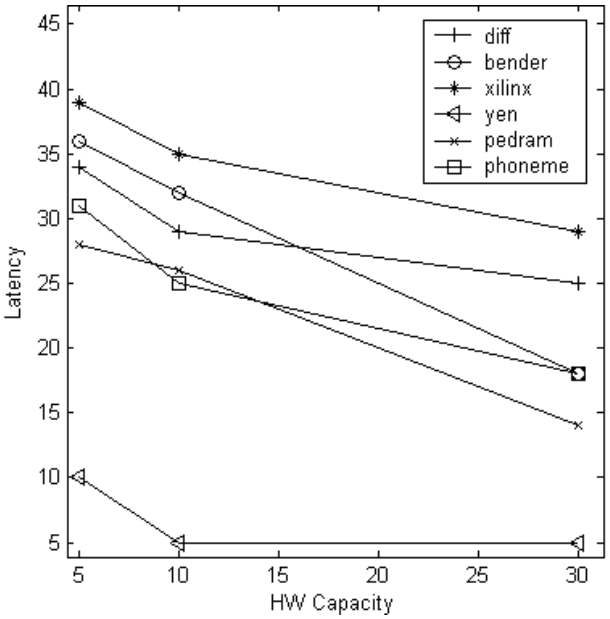

Figure 5. Latency Vs Hardware Capacity

In Table 5 we illustrate the importance of considering edge delays during merge operation (described in Section 3.1) and using modified pruning technique as described in Section 3.2. The experiment was conducted with the same parameters as discussed above. The quality of our proposed algorithm *edge_prune* is compared with a traditional design flow called *general*, which does not account for edge communication delays during merge operation and pruning.

In general design flow the HW/SW partitioning is done assuming no edge delays (i.e., all communication costs to be zero). Then a HW/SW partitioning solution is obtained with minimum latency assuming no edge delays. For this HW/SW partitioning solution the schedule latency is now recalculated by assuming edge delays using ASAP scheduling. This gives the latency reported in Table 5 column 2. The latency of the solutions obtained through our proposed algorithm is reported in column 3.

Table 5. Comparison of solution Qualities

Name	HW used (*general*)	Latency (*general*)	HW used (*edge_prune*)	Latency (*edge_prune)*
diff	22	25	22	25
bender	19	18	19	18
xilinx	18	39	26	29
yen	9	9	10	5
pedram	17	24	21	14
phoneme	22	29	27	18

As seen from Table 5 we can see that our proposed algorithm provides significantly better solutions for most benchmarks by considering edge delays during the merge and pruning operations.

In Table 6 the results for the heuristic for the various benchmarks are reported. The experiments were run on the same machine with the parameters such as HW capacity, edge delays being the same as for Table 4. It can be seen that the run time is almost zero for all benchmark circuits and it gives good results compared to the dynamic programming method for all benchmarks.

Table 6. Experimental Results for the heuristic

Name	Nodes/Edges	HW used	Run time (s)	Latency
diff	21/24	30	0.0	25
bender	12/14	14	0.0	28
xilinx	28/38	30	0.015	29
yen	6/5	10	0.0	5
pedram	12/15	20	0.0	20
phoneme	23/22	27	0.0	18

8 CONCLUSION

A dynamic programming solution that solves the HW/SW partitioning with finite resource constraint is proposed and implemented. The algorithm has polynomial running time for most practical circuits and this has been experimentally verified. The algorithm is capable of handling edge communication delays effectively by incorporating them into the dynamic programming framework. Pruning techniques are developed to account for edge delays and reconvergent fan out without losing optimal solutions. A very fast and good quality heuristic was also proposed to provide good approximate solutions for negligible run times.

9 REFERENCES

[1] A. Jantsch, P. Ellervee, J. Oberg, A. Hemani, and H. Tenhunen, "Hardware/Software Partitioning and Minimizing Memory Interface Traffic" in *Proceedings European Design Automation Conference*, 1994

[2] P. Knudsen and J. Madsen, "PACE: A Dynamic Programming Algorithms for Hardware/Software Partitioning," in *Proceedings IEEE International Workshop on Hardware / Software Co design*, 1996.

[3] J. M. Chang, M. Pedram "Codex-dp: codesign of Communicating Systems Using Dynamic Programming", *IEEE Transactions on Computer*

Aided Design of Integrated Circuits and Systems, Vol. 19, July 2000.

[4] A. Balboni , W.Fornaciari, and D.Sciuto, "Partitioning and exploration strategies in the TOSCA codesign flow" in *Proceedings IEEE International Workshop Hardware/Software Codesign*, 1996.

[5] J. G. D' Ambrosio and X. Hu "Configuration level hardware/software partitioning for real time embedded systems" in *Proceedings International Workshop Hardware/Software Codesign* 1994.

[6] N. D. Dutt, "High-Level Synthesis Design Repositories", http://www.ics.uci.edu/dutt

[7] http://www.xilinx.com/xcell/xl23/xl23_16.pdf

[8] T.Y. Yen, W. Wolf, "Communication Synthesis for Distributed Embedded systems," in *Proceedings IEEE International Conference on Computer – Aided Design*, 1995.

[9] H. Schmit, D.E Thomas, "Synthesis of application-specific memory design", *IEEE Transactions on VLSI systems*, 1997.

[10] M. F. Parkinson and S. Parameswaran, "Profiling in the ASP codesign environment", in *Proceedings IEEE/ACM International Workshop Hardware/Software Codesign*, 1995.

[11] G. D. Micheli,, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[12] A. Bender, "MILP Based Task Mapping for Heterogeneous Multiprocessor systems," in *Proceedings European Design Automation Conference*, 1996.