Optimal Hardware/Software Partitioning for Concurrent Specification using Dynamic Programming

Aviral Shrivastava, Mohit Kumar, Sanjiv Kapoor, Shashi Kumar, M. Balakrishnan
Department of Computer Science & Engineering, I.I.T. Delhi, New Delhi-110016, INDIA
skapoor,shashi,mbala@cse.iitd.ernet.in

Abstract

An important aspect of hardware-software co-design is partitioning of tasks to be scheduled on the hardware and software resources. Existing approaches separate partitioning and scheduling in two steps. Since partitioning solutions affect scheduling results and vice versa, the existing sequential approaches may lead to sub-optimal results. In this paper, we present an integrated hardware/software scheduling, partitioning and binding strategy. We use dynamic programming techniques to devise an optimal solution for partitioning of a given concurrent task graph, which models the co-design problem, for execution on one software (single CPU) and several hardware resources (multiple FPGA's), with the objective of minimizing the total execution time. Our implementation shows that we can solve problem instances where the task graph has 40 nodes and 600 edges in less than a second.

1 Introduction

New tools which extend design automation to system level have to support the integrated design of both the hardware and the software components of an embedded system. The input specification accepted by such design tools describes the functionality of the system together with some performance constraints and is typically given as a set of interacting processes. Satisfaction of these performance constraints can frequently be achieved only by hardware implementation of some components of the specified system. In this paper we describe a dynamic programming based approach to select parts of functionality for hardware and software implementations with the objective to minimize the total execution time. This is a well-known hardware/software partitioning problem.

The general "hardware/software codesign" problem is known to be a hard problem and no polynomial time algorithm exists for this problem. Many heuristics based on different cost functions and approaches like greedy approach, simulated annealing [2, 3, 4, 5], TABU search [5] etc. have been developed. Developing an algorithm to find an optimal solution is still very important to get a deeper understanding of the problem, as well as for comparing different heuristic solutions. Optimal partitioning techniques that have been proposed are based on ILP formulations or Branch and Bound techniques and suffer from very high computational requirements even for small problem sizes [6].

Our target architecture consists of a single processor and reconfigurable hardware in the form of multiple FPGAs. In this scenario, apart from partitioning, we need to address the scheduling invocation of individual tasks and binding of tasks to FPGAs. In our approach, we integrate the tasks of scheduling, partitioning and binding and solve this optimally by using dynamic programming techniques.

This paper is divided into five sections. Section 2 describes our assumptions about the co-synthesis environment and the execution model. Sections 3 deals with the problem formulation and an algorithm for the dynamic programming solution of the problem. In section 4 we present and discuss our results. We also discuss the data generation strategy used to empirically test our algorithm. We draw some conclusions in the last section.

2 The Co-Synthesis Environment

Our underlying model of architecture is a single processor host connected to a reconfigurable multiple FPGA board system through a bus. The program initially is in software and contains a set of functions which can either be mapped onto the FPGAs or executed in the processor itself. We have a library of FPGA configuration files of all the functions that can possibily be mapped onto FPGA's.

If a function is to be executed on an FPGA and that function is not already configured on that FPGA, then first the configuration file is written onto the FPGA, followed by passing the parameters and then execution of the function [7]. When the function execution ends on the FPGA, an interrupt is raised and the results of the computation are read

from predefined "locations" on the FPGA by the processor. Functions can be executing in parallel on the processor (in software) and the FPGA's (in hardware).

We assume that a function is mapped onto exactly one FPGA and all the parameter passing is controlled by the processor. Further, we assume that once a function starts on a FPGA, it proceeds uninterrupted till it ends. This means that if a function is executing on an FPGA, only when it ends, can the next scheduled function start.

Notations and Constraints

• Computing resources are denoted by $r_i,\ i \in (0..m)$. r_0 denotes the software resource, the processor. $r_i,\ i \in (1..m)$ denote the m hardware resources (FPGA's) . • We have n function instances denoted by $f_i,\ i \in (1..n)$. Each function instance is chosen from the following set of function types: $\{F_i,\ i \in (1..N)\}$.
• A function $ntoN(i)$ exists which maps function instances to function types.
• Hardware time, Software time, and Configuration time of each function is known and denoted by $hw\_time[i],\ conf\_time[i],\ sw\_time[i],\ \forall i \in (1..N)$, respectively.
• Precedence Constraints are depicted by the $\prec$ (**prec**) operator. If $f_i \prec f_j$ then f_j can start executing only after f_i has ended.
• There is no preemption, i.e. if a function is executing on a resource, next function can start only after it has ended.

The input to the partitioner is a concurrent sequence flow graph and the output is a schedule of functions for each resource, the time when a function has to start on an appropriate resource, and the time taken to execute using this schedule. The objective is to minimize the total time of execution of the functions defined by the sequence flow graph.

3 Dynamic Programming Algorithm

In this section we set up a Dynamic Programming Recurrence which characterizes the optimal solution to the HW/SW partitioning problem. We assume that all times are integral. As a convenience in setting up the recurrence, we use a function f_0 whosse execution time and configuration time are zero on each resource. A function f_0 on any resource implies that the resource is free. We will use discretized time instants for establishing our recurrence.

3.1 Parameters of the recurrence

The DP recurrence will have two parameters, one representing the Current Configurartion of the FPGA's and the other representing the queue of function waiting to be scheduled for execution. The second parameter is necessary since it is not possible to schedule all the functions which become candidates for execution on a FPGA after their precedence constraints are satisfied.

No.	Case Description	Time to be added
1	No function on processor	0
2	No function on FPGA	0
3	A function on processor	$sw\_time(ntoN(i))$
4	A function of same type on FPGA	$hw\_time(ntoN(i))$
5	A function of different type on the FPGA	$hw\_time(ntoN(i))$ $+\ conf\_time(ntoN(i))$

Table 1. Case enumeration for processor

We characterize the configuration as follows:

The current configuration is a $m+1$ tuple of functions.

$$C_i\ =\ <f^c_{i0},\ f^c_{i1},\ f^c_{i2},\ \cdots, f^c_{im}>$$

where $f^c_{ij}\ ,\ i \in (1..n),\ j \in (0..m)$ is the function that is being executed on j^{th} resource at the end of i^{th} event.

The current queue is represented as a list of functions.

$$Q_i\ =\ <f^q_{i0},\ f^q_{i1},\ f^q_{i2},\ \cdots, f^q_{ik}>$$

where $f^q_{ij}\ ,\ i \in (1..n),\ j \in (1..k)$ is the function waiting for execution at the ith event.

The size of the list, k is not known. An upper bound for k may be $n-2$. This occurs when all the $n-2$ nodes are the children of one node and parent of another node. Thus $k \leq n-2$.

The state of the system is uniquely determined by the pair $< C_i,\ Q_i >$.

$$C_i\ =\ <<f^c_{i0},\ t_{i0}>,\ <f^c_{i1},\ t_{i1}>, \ldots, <f^c_{im},\ t_{im}>>$$

$$Q_i\ =\ <f^q_{i0},\ f^q_{i1},\ \ldots,\ t_{ik}>\ ,\ k \leq n-2$$

In the current configuration along with f^c_{ij} is also kept $t_{f_{ij}}$ which is the time left for the complete exceution of f^c_{ij} at the end of the ith event.

3.2 Formulation of the Recurrence

The recurrence reflects the following recursive procedure for optimum partitioning. At the occurrence of an event we perform the following tasks.

- Find the function that is going to finish first.

$$opt(<< f^c_{i\,0}, t_{i\,0} >,\ < f^c_{i\,1}, t_{i\,1} >>,\ < f^q_{i\,1},\ ..,\ f^q_{i\,k_i} >) =$$

$$min \begin{cases} opt(<< f_0,\ 0 >, < f^c_{i+1\,1} = f^c_{i\,1},\ t_{i+1\,1} = t_{i\,1} - t_{i\,0} >>,\ < f^q_{i+1\,1} = f^q_{i\,1} >) & (1) \\ opt(<< f^c_{i+1\,0} = f^c_{i\,0},\ t_{i\,0} - t_{i\,1} >,\ < f_0,\ 0 >>, < f^q_{i+1\,1} = f^q_{i\,1} >) & (2) \\ opt(<< f^c_{i+1\,1} = f^q_{i\,1},\ sw\_time[f^q_{i\,1}] >,\ < f^c_{i\,1},\ t_{i\,1} - t_{i\,0} >>,\ <>) & (3) \\ opt(<< f^c_{i\,0},\ t_{i\,0} - t_{i\,1} >,\ < f^q_{i\,1},\ hw\_time[f^q_{i\,1}] >>,\ <>) & (4) \\ opt(<< f^c_{i+1\,0},\ t_{i\,0} - t_{i\,1} >,\ < f^q_{i\,1},\ hw\_time[f^q_{i\,1}] + conf\_time[f^q_{i\,1}] >>,\ <>) & (5) \end{cases}$$

Table 2. Dynamic Programming Solution Recurrence

- Remove the finishing function from the configuration.
- Add the children of the finished function to the queue if all their other predecessors are executed.
- Then we recursively try all the permutations of mapping the functions on different resources. This includes the case when one or more resources are free.
- While backtracking, we choose the permutation that takes the minimum time to execute.

We consider all the cases, so any function can be mapped onto any resource. This includes the case when the resource is kept free.

In each of the different cases that can arise, the execution time to be added is different. The different cases and the time to be added in each case is summarised in Table 1. For illustration purposes, we write the recurrence (Table 2) for $m = 1$ and $k_i = 1$, for the case when no child is added to the queue. We have a general form suitable for multiple resources and multiple functions waiting in the queue [8].

3.3 Iterative Solution of the Recursion

We encode the configuration and the queue by integers so that we can use these integers to access the table. A table is maintained, wherein the $< C_i, Q_i >$ contains $opt < C_i, Q_i >$. The value for a node is calculated only once and later whenever it is required, it is looked upon from the table only.

Encoding of C_i

$$C_i = << f^c_{i0}, t_{i0} >, < f^c_{i1}, t_{i1} >, ..., < f^c_{im}, t_{im} >>$$

We need $\log_{10} n$ digits to store a function. Suppose the max time of a function is t then we require $\log_{10} t$ digits to specify the time value. For one resource we need $\log_{10} nt$ digits. Since there are $m + 1$ functions in the configuration at a certain time, we need $(m + 1) \times \log_{10} nt$ digits for encoding all the functions.

Encoding of Q_i

The length of queue can be at most $n - 2$, the worst case being when all the $n - 2$ functions are children of the first node and are the parent of the last node. We keep a bit to indicate the absence or presence of a function. There are n functions and so we need $\log_2 n$ bits to encode the functions.

no. of function types	no. of function instances	no. of FPGA's	no. of edges	InA time (ms)	InB time (ms)
2	4	2	7	0	0
2	10	2	30	8	7
3	15	3	60	20	15
3	20	3	150	36	24
3	30	3	300	41	38
5	40	4	600	60	52

Test Graphs in InA column are RSCFG's
Test Graphs in InB column are Fork-Join graphs

Table 3. Time taken to partition Sequence Flow Graphs

4. Implementation and Results

4.1. Table Structure

The Dynamic Program looks at all the cases and chooses the one with minimum time. Due to the optimality constraint, the search space becomes extremely large. Traditionally a table is constructed in which times for each configuration-queue pair is kept. Now since the table formed in our case is so huge, we cannot afford to maintain such a table. So we choose to allocate space for this ta-

ble dynamically. We use a structure based on binary search trees . The structure is a tree at two levels. At the first level is the configuration tree, and at the second level is the queue tree. When we are to search for a configuration-queue pair, we first find the configuration node by traversing the configuration tree. Once we find the configuration node, we jump to the queue-tree attached to that configuration node. Now we search for the queue in the queue-tree. Since these are tree structures, the time complexity is proportional to the logarithm of the height of the tree.

4.2. Data Generation and Results

To test the implementation and to compare it with other contemporary implementations we require test examples, i.e. sequence flow graphs. We studied the flow graphs in many real-life examples and found that most of them are of the fork-and-join type. To complete our study, we tested our program on both the random graphs and the fork-join graphs.

Random Concurrent Sequence Flow Graphs: To generate *Random Concurrent Sequence Flow Graph* (*RCSFG*) of n nodes we first fill the top-right-half adjacency matrix size $n-1$ by $1's$. Now we generate random numbers between 0 and $\frac{n(n-1)}{2}$. If the generated random number denotes a element in top-right-half adjacency matrix, which is 1 and the changing of this element from 1 to 0 does not violate the following row or column constraint , then we change the element from 1 to 0.

Row Constraint : There is at least one "1" in each row.

Column Constraint : There is at least one "1" in each column.

We take as input the number of edges in the graph from the user, and change $1's$ to zeros until only desired number of edges remain.

Fork-Join Graphs: To generate *Fork-Join Graphs* we first break the graph into *single nodes* and *fork-join units* that denote the fork and joins. This we do by first choosing a random number k between 1 and n. Now treat these as k slots. First fill all the slots with one node . Now randomly distribute the remaining node into the slots. The slot that contains just 1 is a single node, and a slot that contains more than 1 nodes is a *fork-join unit*. A *fok-join unit* is a single entry single exit graph. We can generate this just like random concurrent sequence flow graphs. This strategy even allows straight line graphs to be generated.

We tested for n ranging from 4 to 40, m ranging from 2 to 4, and N ranging from 2 to 5. The number of edges range from 7 to 600. The graphs of both the types were generated randomly. The results are tabulated in Table 3.

The results clearly confirm our hypothesis that actually the search space won't be very large inspite of the large solution space. The use of dynamically allocated memory has resulted in an efficient implementation with low space requirements. The results show that optimal solution to large real life problems are possible using dynamic programming based approach.

5 Conclusions

In this paper, we have described a dynamic programming based formulation to find optimal solution to the integrated scheduling, partitioning and binding problem in the context of hardware/software codesign. We have demonstrated that it is possible to find optimal solution to large real life problems using our approach. This is in contrast to experiences with other approaches like *Integer Linear Programming* and *Branch & Bound*.

However, our formulation is limited to handle concurrent sequence flow graphs which do not have loops or conditional branch nodes. It will be interesting to attempt extending our formulation to incorporate these extensions. Our testing has also been limited since we have only used random sequence flow graphs and fork-join graphs. The approach needs to be evaluated for large real life case studies.

References

[1] D.D. Gajski, F. Vahid, "Specification and Design of Embedded Hardware-Software Systems", *IEEE Design & Test of Computers*, Spring 1995, pp. 53–67.

[2] J.K. Adams, D.E. Thomas, "Multiple-Process Behavioral Synthesis for Mixed Hardware-Software Systems", in Proc. International Symposium on System Synthesis, 1995, pp. 10–15.

[3] Z. Peng, K. Kuchcinski, "An algorithm for partitioning of Application Specific Systems", in *Proc. European Design Automation Conference, EDAC*, 1993, pp. 316–321.

[4] R. Ernst, J. Henkel, T. Benner, "Hardware-Software Co-Synthesis for Micro controllers", *IEEE Design & Test of Computers*, September 1993, pp. 64–75.

[5] P. Eles, Z. Peng, K. Kuchcinski, A. Doboli, "System Level Hardware/Software Partitioning based on Simulated Annealing and Tabu Search", Design Automation for Embedded Systems, Vol 2 No.1 , Jan 1997, pp. 5-33.

[6] S. Harikumar and Shashi Kumar, "Multi-Objective Search Based Algorithms for Circuit Partitioning Problems", Proc. of the 10th IEEE Int. confernce on VLSI Design, Jan 1997, Hyderabad, India.

[7] Sitanshu Jain et al., "Speeding Up Program Execution Using econfigurable Hardware and a hardware function Library, Proc. of the 11th IEEE Int. confernce on VLSI Design, Jan 1998, Chennai, India.

[8] Aviral Shrivastava, Mohit Kumar, "Hardware Software Partitioning and Synthesis targeted towards FPGA based Implementation", *B.Tech Thesis*, CSE Dept, IIT Delhi, May'99.