Hardware Software Partitioning with Integrated Hardware Design Space Exploration

Vinoo Srinivasan Shankar Radhakrishnan Ranga Vemuri

Digital Design Environments Laboratory
Department of ECECS, University of Cincinnati
Cincinnati, OH 45221–0030
E-mail : {vsriniva, sradhakr, ranga}@ececs.uc.edu

Abstract

This paper presents an integrated approach to hardware software partitioning and hardware design space exploration. We propose a genetic algorithm which performs hardware software partitioning on a task graph while simultaneously contemplating various design alternatives for tasks mapped to hardware. We primarily deal with data dominated designs typically found in digital signal processing and image processing applications. A detailed description of various genetic operators is presented. We provide results to illustrate the effectiveness of our integrated methodology.

1 Introduction

Hardware/Software codesign generates a mixed hardware and software implementation of the specification that satisfies the given timing and cost constraints [1, 2, 3, 4]. Codesign is the natural solution when a full hardware realization satisfies the timing but not the cost constraint whereas a full software solution is not fast enough. With the advent of high speed microprocessors, and new high performance cost-effective hardware technologies like FPGAs and CPLDs, mixed hardware software solution has become very effective for several real-time and embedded systems applications.

There have been several approaches to solve the problem of hardware software partitioning [1, 5, 6, 7, 8, 9]. Fully automatic partitioners are in existence for quite some time now [5, 1, 6]. Gupta and De Micheli [5] start with an all hardware solution and iteratively move one task at a time to software until no further improvement is possible. Ernst and Henkel [1] on the contrary follow a software oriented approach which starts with an all software solution and uses a simulated annealing partitioning engine. Hou and Wolf [6] proposed a process level partitioning heuristic based on hierarchical clustering. Eles [8] performs a performance guided partitioning based on simulated annealing and tabu search. Catania [9] uses fuzzy logic based techniques to partially automate the development of embedded systems.

Although there has been extensive amount of work in the hardware/software partitioning domain, to the best of our knowledge, none of the work presented thus far take into account the alternatives in the design space that can be explored when implementing a given task in hardware. For example, consider a task which is a multiplication of two 2x2 matrices. Depending on how many multipliers and adders are available for the task, the hardware cost and time information will change. Therefore, there is not just a single hardware time and hardware cost pair associated with each task but there is a large set of options available. As the granularity of the tasks increase the number of such options tend to increase at a much faster rate. It is hence important for the partitioner not only to bind tasks to hardware and software but it must also choose the right architectures for the tasks mapped to hardware such that the overall timing and cost constraints are met. In this paper we propose an approach to investigate hardware design alternatives during the cosynthesis procedure. We feel that that the hardware design exploration or the architecture binding problem has to be tackled at the time of hardware software partitioning. There has been several contributions in the area of hardware design space exploration and pruning recently [10, 11, 12, 13, 14]. We use similar techniques to prune the hardware design space and create a tractable number of design points so that an integrated environment for hardware software partitioning and design space exploration is feasible.

This paper presents a Genetic Algorithm (GA) [15] based evolutionary approach to hardware-software partitioning and architecture binding. Figure 1 presents our overall methodology. The input to the system is a task graph. Nodes in the graph represent computational tasks that can be mapped to hardware or software. The edges represent the inter task data dependencies. The first step in our design process is the task level performance estimation. We generate hardware and software models for each task in the graph. A software profiler and a hardware performance estimator are used to determine the required software and hardware metrics for each task to aid the genetic partitioner in performing hardware/software trade-off analysis. The genetic partitioning engine is tied to a codesign performance estimator module that evaluates the fitness of the chromosomes the GAproduces. A detailed explanation of the steps in our methodology is presented in the sections to follow.

The organization of the paper is as follows. In Section 2 we formulate our problem in the perspective of our task graph model and the target architecture model. Section 3 presents the task level performance estimation technique. Details of our genetic

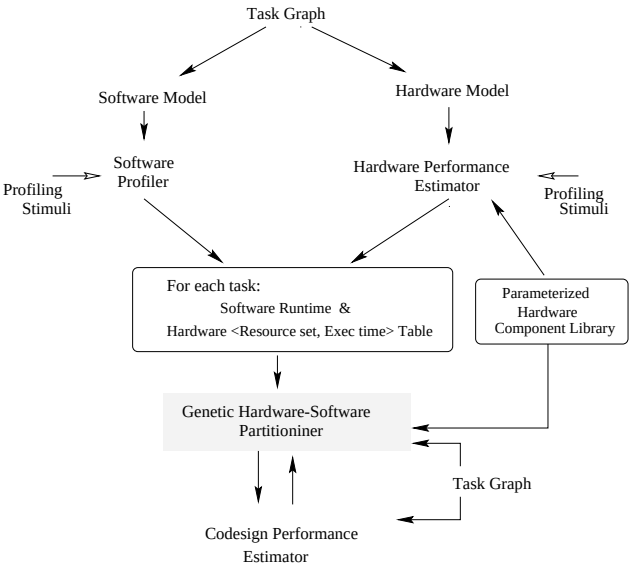


Figure 1: **Partitioning Methodology**

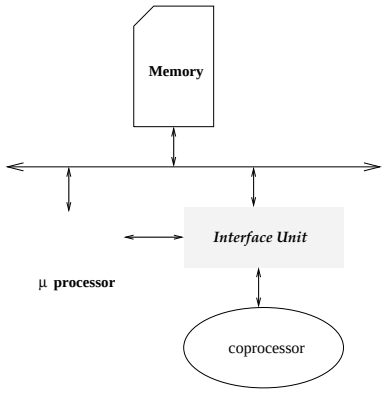


Figure 2: **The Target Architecture**

algorithm is provided in Section 4. Section 5 presents the codesign performance estimation and scheduling algorithm. Results to illustrate the efficiency of our approach is in Section 6. In the final section, we put-forth our conclusions and discuss related issues.

2 Problem Formulation

Our goal is to find a hardware-software implementation of the task graph specification that maps onto a single host processor, single coprocessor type target architecture such that cost and time constraints are not violated.

2.1 Task Graph

The behavior of the design is presented to the partitioning environment through a directed acyclic *task graph*, $G = (V, E)$. Each node $v \in V$ denotes a task in the input design. A task represents a single thread of execution and cannot be preempted, i.e. it is atomic. Each task may be executed in hardware or software. The computational complexities of the task can vary from simple operations (fine grain) to large functional blocks (course grain). Each edge $e \in E$ denotes data dependencies between tasks. Associated to each task are a hardware specification segment and a software code segment. In our case, we use VHDL and C to represent the hardware and software models for each task. Although control constructs cannot be represented at the task graph level, the individual task themselves have no restrictions. The task graph model fits very well in DSP image processing and communication domains where most of the algorithms are computationally intense and data flow oriented with very little or no control flow. Note that the entire task graph may be iteratively executed on different blocks of data, as it is the case with many DSP and image processing algorithms. This lends to issues of loop pipelining which have been addressed elsewhere [16, 17, 18, 19].

2.2 Target Architecture

The Figure 2 shows the target architecture model. This is a single host processor, single coprocessor model. The communication between the software and hardware tasks is through the shared memory. At any given time only one task is allowed to execute on the coprocessor. However, software and hardware tasks may execute concurrently on the microprocessor and the coprocessor, provided the data dependencies are not violated. Each time any task begins execution, it first reads all the required shared variables from the shared memory, performs the computation and writes the required variables back into the shared memory.

3 Task Level Performance and Time Estimation

As shown in Figure 1, the hardware and software performance and cost metrics of each individual task is available prior to the invocation of the partitioner.

3.1 Software Performance Estimation

Software runtimes for each task is evaluated through profiling the software model of the task graph. The software model of the task graph is extracted from the individual C code fragments for all the tasks. We use *Quantify*©, a commercially available profiling tool [20], to gather software runtime statistics for all tasks. A user given set of profile data is used for profiling the software. This set of profile data is considered to emulate real time input patterns. For the tasks with ND (Nondeterministic Delay) operations [2], their runtimes vary with the input stimuli. In such cases, we approximate the software runtime to be the average runtime over a large set of profiling stimuli.

3.2 Hardware Performance and Area Estimation

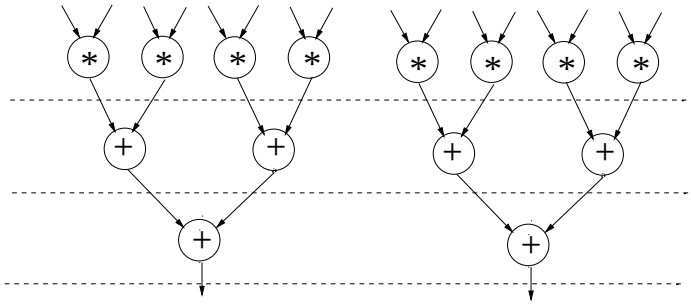
We follow a high level synthesis [21, 22] oriented approach to hardware performance estimation. Typically, the design flow for most high level synthesis environments, starts by exploring the hardware design space available to implement the specification. This step is usually called *module set* generation. A module set is a set of components (with repetitions allowed) from the component library which is sufficient to implement the given specification. The largest module set is the one required by the ASAP (As Soon As Possible) schedule of the specification. The smallest module bag is the one required by the least parallel schedule. Even for reasonably large designs there usually are thousands of possible module sets that are feasible.

```
Ex_task {
   temp00 = inp00*c00 + inp01*c01 +
inp02*c02 * inp03*c03;
   temp01 = inp10*c10 + inp11*c11 +
inp12*c12 * inp13*c13;
       }
```

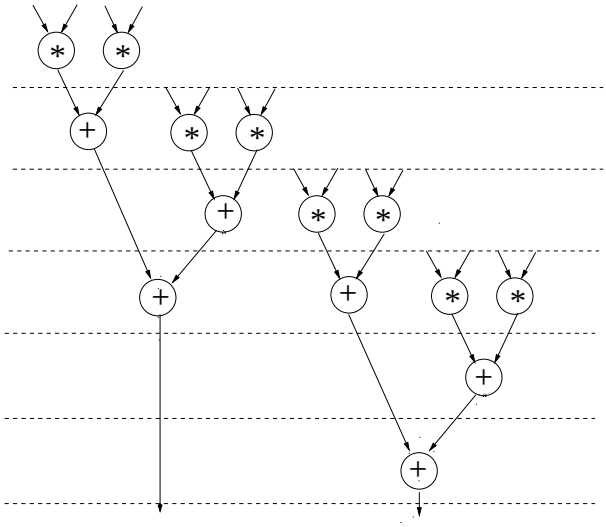
Figure 3: **Code segment for a 4x4 DCT computation**

The feasible module sets form the hardware design space.

Each point in the design space can potentially be considered an area-time trade off point. When there is a large design space available, it is extremely difficult to analyze each point in the space. Thus, efficient heuristics are needed to select a tractable number of candidates that truly capture the flavor of the entire design space. There have been several efforts in the past to perform fast and efficient design space exploration and pruning. A popular approach is to employ lower bound estimation techniques [23, 10] to determine the requirements on the functional unit resources. Later, there has been an effort to provide an exact solution to a 3D design space [11] using integer linear programming and functional unit lower bounding. Traditionally, there have been approaches that employ several performance estimation heuristics [12, 14] to predict design quality. There is also work on hierarchical design space exploration [24] that maintains a set of candidate solutions while pruning the design space. Also, there has been an effort to solve high level synthesis and design space exploration using problem space genetic algorithms [13]. Currently, we run a high level synthesis tool to explore various design points and gather this information [14].



(a) Scheduling under resource constraint : 4 multipliers and 4 adders
Number of Control Steps = 3



(b) Scheduling under resource constraint : 2 multipliers and 2 adders
Number of control steps = 6

Figure 4: **Two different Schedules for Ex_task**

Example:

Consider the task example shown in Figure 3. This task is part of a simple 4x4 DCT (Discrete Cosine Transform) computation. The ASAP schedule for this task requires 8 multipliers and 4 adders. The least parallel schedule requires just one multiplier and one adder. Figure 4 shows two possible hardware implementations of the Ex_task. Figure 4-(a) is the case when the module set $< 8*, 4+ >$ is chosen. The schedule in (a) takes 3 control steps. Figure 4-(b) uses the module set $< 2*, 2+ >$ and takes 6 control steps. The area of a module set is the sum of the areas of the components in the set. We assume that all components are unit cycle operations and thus the cycle time is equal to the maximum delay component selected in the module bag. If there are no control flow nodes and nondeterministic delay operations in the task, then the hardware run time of the task is the product of the number of control steps and cycle time. In the presence of control flow or nondeterministic delay operations, we perform a post-schedule profiling of the schedule generated to estimate the number of cycles the task takes to complete simulation. Similar to the software profiling case, we average the runtimes achieved over a large range of profiling stimuli.

Table 1 is the result of hardware performance estimation for Ex_task of Figure 3. The table gives 10 hardware design options for Ex_task. It can be noticed that area is represented in terms of CLBs (Configurable Logic Block) in a FPGA based coprocessor. A parameterized component library with each component characterized for delay and area values is available to the performance estimator. In Table 1, the fastest implementation has an execution time of 420 ns and occupies 888 CLBs while the slowest implementation has an execution time of 1540 ns but occupies only 316 CLBs. The execution times are based on a clock period of 140 ns.

The hardware/software partitioner we discuss in the next section performs two levels of binding. First, it binds tasks to hardware and software and secondly, it makes an architecture selection for the tasks in hardware. In our space exploration model discussed above we do not restrict the possibility of existence of multiple implementations of a single library component. For example, the component library may have two different implementations of an adder - ripple carry, and carry look ahead. In such a case we consider each implementation as a unique component.

3.3 Communication Performance and Time Estimation

As discussed earlier in Section 2.2, we have a shared memory communication model between the cpu and the coprocessor. The cost and performance values re-

Module Set	#mult	#add	#mux	#reg	#c-steps	exec-time (ns)	Area (CLBs)
1	8	4	18	16	3	420	888
2	8	2	22	16	4	560	880
3	8	1	24	16	7	980	876
4	6	2	26	12	4	560	704
5	4	3	36	8	4	560	564
6	4	2	38	8	4	560	560
7	4	1	40	8	7	980	556
8	4	2	46	4	6	840	400
9	4	2	48	4	7	980	396
10	4	1	52	2	11	1540	316

Table 1: Hardware Design Space Table for Ex_task

ported by the software profiler and the hardware performance estimator do not include the communication time to read and write data from and to the shared memory and the communication cost involved in communication and synchronization hardware. The communication cost and time scale linearly with the number of data variables to be read and written back to the shared memory. Let T_r^s and T_w^s be unit data read and write times between the software (cpu) and the shared memory. Similarly, let T_r^h, and T_w^h be the unit data read and write times between the coprocessor and the shared memory.

All data that is shared between hardware and software tasks is maintained in the shared memory. *Read set* of a task is the set of data variables that the task reads from the shared memory and *write set* is the set of data variables that the task writes to the shared memory. Once tasks are bound to hardware and software, it is easy to determine the read and write sets for all tasks. For the tasks mapped to hardware, notice that read and write sets are independent of the implementation. For a task t, we further define $r(t)$ to be the size of the read set of t and $w(t)$ to be the size of its write set. The hardware-software communication time for a task t, is given by:

$$T_{comm}(t) = \begin{cases} T_r^s * r(t) + T_w^s * w(t) & \text{if } t \text{ is mapped to SW} \\ T_r^h * r(t) + T_w^h * w(t) & \text{otherwise} \end{cases} \quad (1)$$

The communication and synchronization hardware for a task t that is mapped to hardware is given by :

$$C_{comm}(t) = c_{comm} * (r(t) + w(t)) \quad (2)$$

where, c_{comm} is the communication hardware cost incurred for unit data variable read or write between the FPGA and the shared memory.

4 Hardware/Software Partitioning Algorithm

4.1 Genetic Algorithms

We model and solve the partitioning problem through a Genetic Algorithm (GA). To form an effective search strategy, GAs combine "survival of the fittest" approach with a randomized, yet structured information exchange mechanism among potential solutions. Search using GAs strikes a reasonable balance between exploiting the available information and exploring the unsearched regions of the design space. The genetic search procedure was developed by John Holland in 1975 [15], and since then has been used successfully for solving several combinatorial problems in VLSI design automation [25, 26, 27]. Following is a summary of how a genetic algorithm works.

A genetic algorithm consists of an iterative procedure during which a series of *generations* of *populations*, one per iteration, are created. Each member of of population, also called chromosome, represents a solution of the problem being solved. The solution representation is based on a suitable encoding of the solution space. The population during iteration i of the genetic algorithm is denoted by the set, $p_i = \{x_1^i, \cdots, x_n^i\}$, where, x_m^i denotes the mth member of the population in ith iteration and n is the size of the population that is usually fixed for the run of the GA. Note that solutions may repeat in a population. The initial population, p_1 may be created randomly or by using some deterministic heuristic [28, 29]. From the optimization perspective, genetic algorithms attempt to discover an optimal – least cost – solution to the problem. Let the cost of the solution x be designated by $cost(x)$. *Cost* function is supplied by the user as part of the problem specification. The GA is oblivious to the properties of cost functions. Without loss of generality, we assume that the cost function is defined such that for any solution, $cost(x) \geq 0$. Let the $fitness$ of a solution be defined by,

$$fitness(x) = \frac{1}{1+cost(x)} \quad (3)$$

Note that for any solution x, $0 \leq fitness(x) \leq 1$. In our case, the cost of a chromosome is evaluated by the codesign performance estimator which will be discussed in the next section. GA uses an *evolution function* to generate a new generation p_{i+1} from an existing generation p_i. The evolution function usually consists of three components, called *operators*:

- *Selection:* Selection operators probabilistically select some of the members of the current population to move to the next generation. The selection heuristic is such that highly fit solutions in the population are most likely to be selected. The function *select_chromosome*() probabilistically selects a highly fit chromosome. The selection operator is used to create a small number (10-30%) of the members in the next generation. The remaining are provided by the crossover operator.

- *Crossover:* Crossover operators attempt to com-

bine the features of highly fit members of a population in the hope of creating new members that are likely to be better fit than either of their parents. Crossover operators attempt to enhance the genetic mix of the population by mating two highly fit parents. It is believed that this is the key to success of natural evolution and hence might be to artificial evolution as well.

- *Mutation:* Mutation operators randomly alter the structure of the chromosomes. The user specifies a mutation probability, P_m. Mutation probability is usually set to low values in the range [0.05;0.2]. Mutation attempts to introduce new features into the population that did not exist in the previous generation. Mutation alone would represent random walk through the solution space.

Following the generation of the new population, the current population is discarded and the new population becomes current. This evolution process continues until a user specified termination condition has reached. The termination condition is either a constraint satisfying solution or an upper limit of the number of generations that the GA explores.

4.2 GA for HW/SW Partitioning

Encoding: As discussed earlier in Section 3.2, the hardware performance and area estimator generates a table of design options for every task (Table 1). The size of the table is fixed for all tasks. Any feasible encoding for the solution must provide information about the binding of the each task to hardware or software and if the binding is to hardware, it must point to an entry in the hardware performance table.

We use an integer array to represent each chromosome (hardware-software partition). The length of the array is equal to the number of tasks. Let N be the number of tasks and let the integer array $c[1 \,..\, N]$ denote a chromosome. If k_i is the number of entries (i.e. number of hardware implementation options) in the hardware performance table of task i, then for $1 \leq i \leq N$, $1 \leq c[i] \leq 2.k_i$. In other words, each entry in the chromosome array can range form 1 to twice the number of entries in the hardware performance table of the corresponding task.

Associated with each task i is a *mapping array*, $\mathcal{M}^i$, of size $2.k_i$. The mapping array is an array of pointers. Of the $2.k_i$ pointers in the mapping array k_i randomly chosen pointers are NULL. The remaining k_i pointers point to one of the rows in the hardware performance table of the task i. No two pointers in the map array can point to the same row of the table. Creating the mapping array for each task is a one time operation.

The interpretation of the encoding is as follows. If $\mathcal{M}^i[c[i]]$ is NULL then the task i is bound to software, else, the task is bound to hardware and $\mathcal{M}^i[c[i]]$ points to the implementation (hardware design point) chosen for the task i. The reason of having $2.k$ entries in $\mathcal{M}$ and leaving k of the NULL is to provide equal opportunity for a task to either be in hardware or software. Thus, there is no bias towards a hardware or a software solution. Consider a case when $N = 4$, $k_1 = 4$ and $k_4 = 2$. The Figure 5 pictorially illustrates our encoding scheme.

Initial Population: We force one of the chromosomes in the initial population to be an all software solution. The rest of the chromosomes are randomly generated. For the random chromosomes, for each entry $c[i]$ in the array is assigned a random value between 1 and $2.k_i$, where k_i is the number of entries in the hardware performance table for task i.

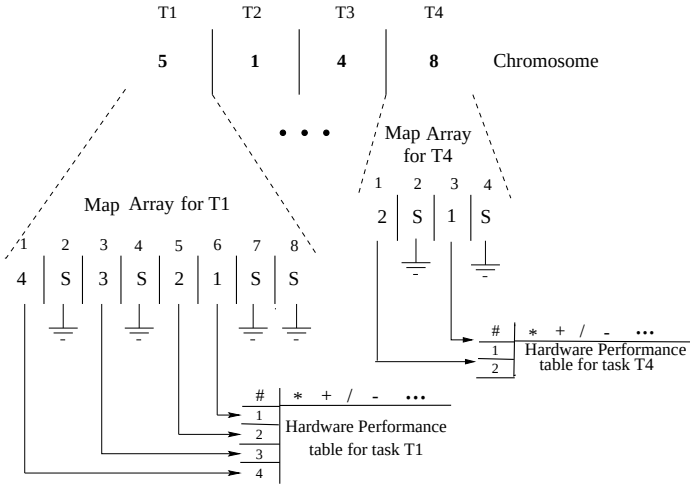


Figure 5: **Chromosome Encoding**

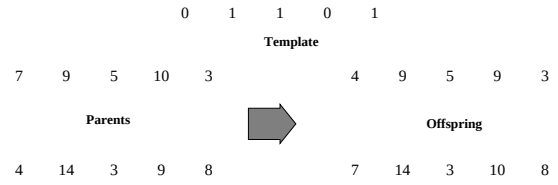


Figure 6: **Effect of Uniform Crossover**

Crossover: We use a uniform crossover operator. In this crossover, a binary string, T, whose length is equal to the number of tasks in the graph is generated. Each bit in this *template* is randomly set to either 0 or 1. Next, two parents p_1 and p_2 are probabilistically selected for mating. Let c_1 and c_2 be the children resulting from the crossover. Then,

$$c_1[i] = \begin{cases} p_1[i] & \text{if } T(i) = 1 \\ p_2[i] & \text{otherwise} \end{cases}$$

$$c_2[i] = \begin{cases} p_1[i] & \text{if } T(i) = 0 \\ p_2[i] & \text{otherwise} \end{cases}$$

Figure 6 shows the effect of uniform point crossover.

Mutation: The mutation operator randomly selects an entry in the chromosome array and changes its value. Effectively, the mapping of a single task is modified. The mapping of the task can change in three ways. The task can move from software to hardware, hardware to software, or the the task may continue to remain in hardware with a change in its implementation.

5 Codesign Performance and Cost Estimation

Each chromosome generated by the genetic partitioning engine corresponds to a solution in the codesign space and hence has to be evaluated for its fitness.

Algorithm 5.1 (Codesign Hardware Area Estimation)

T: set of tasks bound to hardware.
$c_1, c_2 \cdots c_{N-1}, c_N$ are the components in the design library.
t_{xy}: Number of components of type c_y present in the implementation chosen for task x.
Area(c_x): The area of the component c_x

Estimate_hwarea(T)
begin
  for *j in 1 to N* **do**
    $n_j \leftarrow max\{\forall t_i \in T : t_{ij}\}$
  end do
  $Hardware\_area \leftarrow \sum_{i=1}^{N} (n_i \times Cost(c_i))$
  $\mathcal{R}_{hard} \leftarrow \{(c_1, n_1), (c_2, n_2), \cdots (c_N, n_N)\}$
  return(*Hardware_area*, $\mathcal{R}_{hard}$)
end

The fitness of the chromosome depends on whether it satisfies the hardware area and execution time constraints posed on the task graph.

Hardware Area Estimation: Since the tasks assigned to the co-processor do not execute in parallel, hardware resources can be fully shared among the hardware tasks. Hence, adding the areas of the chosen resource bags of all hardware tasks may be an over estimation of the hardware area. For this reason we use the procedure in Algorithm 5.1 to estimate the hardware area. The hardware required for communication is estimated as discussed in Section 3.3 and is not shown in this figure.

Codesign Runtime Estimation: The area estimation procedure in Algorithm 5.1 returns the the hardware resource set available ($\mathcal{R}_H$). The performance tables of all hardware tasks are revisited to find the fastest feasible implementation constrained by the resource set $\mathcal{R}_H$. The communication time penalties as estimated in Section 3.3 are added to the respective tasks. Now that the run times and binding of each task is available, we use a list scheduler [22] to schedule the task graph. The priority function of our list scheduler implementation is based on the mobility values of the tasks. We used list scheduler because it is very fast and efficient. However, changes can be made to the priority function at will and the scheduler itself can be modified, if need be, without affecting the rest of the partitioning environment.

The schedule time gives us the estimated codesign execution time of the task graph.

Codesign Cost Function: Our goal is to produce a codesign implementation which satisfies the conflicting constraints. Accordingly, the cost function, which determines the cost of a partition is a combination of the two conflicting cost factors. Let A_{est} and T_{est} be the estimated area and execution time of the codesign and let A_{max} and T_{max} be the respective user specified constraints. Then our cost function is:

$$cost = \frac{\Delta A}{A_{max}} + \frac{\Delta T}{T_{max}}$$

where, $$\Delta T = \begin{cases} 0 & \text{if } T_{est} \leq T_{max} \\ T_{est} - T_{max} & \text{otherwise} \end{cases}$$

Example Name	Num. Tasks	Num. Edges	Num. Library Components
DCT-1	9	20	4
DCT-2	36	76	4
Reg_Anal	8	8	6
FFT	15	22	5
LT	9	12	6
Mean	9	12	10
LUD	9	11	6
rand_20	20	40	8
rand_50	50	100	10
rand_75	75	100	6
rand_100	100	500	6
rand_400	400	800	15
rand_500	500	1500	20

Table 2: **Design Data for Test Examples**

$$\Delta A = \begin{cases} 0 & \text{if } A_{est} \leq A_{max} \\ A_{est} - A_{max} & \text{otherwise} \end{cases}$$

Fitness of the chromosome is calculated according to the equation (3) in Section 4.1.

6 Experimental Results

In this section we present results of the genetic algorithm (GA) for hardware software partitioning. We have implemented all algorithms in C++ on a Sparc 5 Unix workstation at clock-speed 143 Mhz. We compare the results of the GA with an implementation of simulated annealing (SA) [30]. The reason for picking SA for our comparison is because SA is a well established randomized algorithm and is widely used by several researchers in various problem domains, including hardware software partitioning [1, 8].

Table 2 shows the design data for the various test graphs we used to study the effectiveness of our methodology. *DCT-1* and *DCT-2* are both task graphs for a 4x4-matrix discrete cosine transform operation. The tasks in DCT-2 are fine grain resolution in comparison with DCT-1. *Reg_Anal* was generated from a *C* code which computes the line of best fit using linear regression analysis. *FFT* is a fast Fourier transform butterfly network model. *LT* is a Laplace Transform task graph. *Mean* is a task graph for performing mean value analysis and *LUD* performs LU decomposition. The four task graphs models FFT, LT, Mean and LUD are the ones developed by Ahmad et. al. in [13]. Since we wanted to verify the performance of the GA on large task graphs, we implemented a task graph generator to generate synthetic graphs of arbitrary sizes. Designs *rand_20* through *rand_500* are randomly generated graphs of varying sizes. Column 4 of Table 2 gives the number of different component types that we used in the design. For all the tasks belonging to the task graphs in Table 2, the size of the hardware performance table ranged from 10 to 50 entries depending on the granularity of the task. For the synthetic benchmarks, the performance tables for all the tasks involved was also randomly generated.

The GA was executed with population size 200, crossover percentage 80%, and mutation probability 0.15. The GA terminates either if a constraint satisfying solution is reached or if 200 generations have

Example	Area Cnst. CLBs	Time Cont. ns	Genetic Algorithm				Simulated Annealing			
			A_{est} CLBs	T_{est} ns	Fitness	run time hh:mm:ss	A_{est} CLBs	T_{est} ns	Fitness	run time hh:mm:ss
	2800	3000	2744	2820	0.9400	00:00:51	2744	2820	0.9400	00:00:26
DCT-1	5000	2600	4976	2520	1.0000	00:01:03	4976	2520	1.0000	00:00:34
	1000	8000	1212	7980	**0.8251**	00:03:11	1244	7980	0.8039	00:01:34
DCT-2	2000	7000	1276	7980	0.8772	00:03:27	1276	7980	0.8772	00:01:45
	1000	2100	977	2180	0.9633	00:01:44	921	2180	0.9633	00:00:38
Reg_Anal.	700	2200	733	2310	0.9115	00:03:34	733	2310	0.9115	00:00:36
	9000	80	11523	81	**0.7735**	00:06:44	11853	80	0.7593	00:04:13
FFT	10000	90	10603	92	**0.9238**	00:02:37	10772	92	0.9090	00:00:42
	8100	50	10463	47	**0.7742**	00:05:01	46	10717	0.7558	00:03:28
LT	10000	350	358	1119	**0.8813**	00:06:40	12068	349	0.8291	00:03:34
	6900	160	15016	143	**0.4595**	00:06:53	15724	148	0.4388	00:04:13
Mean	15000	150	14785	150	**1.0000**	00:07:26	15281	149	0.9816	00:05:17
	3100	400	3005	404	0.9900	00:03:31	3005	404	0.9900	00:01:54
LUD	7900	160	8222	147	**0.9608**	00:03:08	8594	147	0.9192	00:01:28
	9500	500	13704	500	**0.6856**	00:01:28	14123	500	0.6726	00:00:53
rand_20	7500	1000	7735	1028	**0.9440**	00:01:27	7519	1065	0.9367	00:01:19
	18000	1900	18339	1868	**0.9815**	00:03:34	19190	1898	0.9379	00:02:23
rand_50	17000	2200	17195	2198	**0.9887**	00:07:51	17288	2230	0.9703	00:05:17
	900	13000	867	13215	**0.9837**	00:20:21	13439	857	0.9673	00:12:21
rand_75	9000	900	12831	891	**0.7015**	00:19:09	13081	897	0.6880	00:16:26
	12000	1900	12743	1847	**0.9417**	00:20:19	14096	1859	0.8513	00:14:45
rand_100	31000	2000	30960	1984	1.0000	00:29:06	30994	1983	1.0000	00:29:34
rand_400	42000	10000	45466	9792	**0.9238**	01:31:47	45519	9862	0.9226	01:07:13
rand_500	55000	11000	58526	10900	**0.9398**	01:58:12	58800	10996	0.9353	01:00:05

Table 3: **Results of Genetic Hardware Software Partitioning**

passed without improvement in the best solution. The simulated annealing implementation uses the same chromosome encoding and cost estimation as used by the GA. The perturbation function of the SA modifies the mappings of the tasks in the chromosome. The amount of perturbation depends on the temperature. We start the SA at very high temperature of 100,000 and cool it down to 0.1. The cooling factor is set to a very low value (0.00003) so that the SA had enough opportunity to search for good neighborhood solutions.

Table 3 presents the results of the comparison between the GA and the SA. Columns 2 and 3 are the area and time constraints posed on the codesign respectively. The table reports the estimated area and time of the best solution detected by the GA and SA. It also reports the fitness of the best solution and the cpu time taken to arrive at that solution for both GA and SA. When the constraints were loose, both SA and GA were efficient and arrived at the constraint satisfying solution. For this reason we set very tight constraints on area and time to investigate how the two algorithms perform under tight constraints.

The results in Table 3 show that the GA has a definite edge over the SA terms of fitness of the best solution generated. The GA produced better solution than SA 17 out of 24 times (shown bold faced in the GA fitness column). For the remaining cases the GA and SA produced the same best fitness value. The reason for GA performing better than SA may be attributed to the fact that GA maintains a population of solution and has a structured evolution strategy while the SA moves from one point in the solution space to the other in a less structured manner. Moreover the GA operators avoid being trapped in local minima. The GA pays some time penalty for carrying a population of solution along. However, the table shows that for most cases the SA and GA times are in a comparable range. Even for the largest example, *rand_500*, the GA is not much slower than the SA. Given the fact that partitioning a task graph is a one time operation, the additional time spent in generating a better solution most often is worth the effort.

7 Discussion and Conclusions

This paper presented an integrated approach to hardware software partitioning and hardware design space exploration. We proposed a genetic partitioning environment which can partition a task graph into hardware or software while at the same time perform trade-off analysis among the different hardware alternatives available for each task. In the case where there are multiple software alternatives available for the tasks, it is easy to extend our genetic framework to perform a combined hardware software design space exploration problem of which hardware software partitioning becomes a sub-problem. In comparison to a simulated annealing algorithm performing the same task, the genetic algorithm proved much superior in effectively exploring the codesign space.

In the methodology discussed in this paper, a high level synthesis tool initially estimates the performance of all the tasks and generates a tractable number of hardware design points for the genetic algorithm to contemplate. The GA chooses an implementation for each of the tasks. The implementation may be a software implementation or one of the hardware options. Thus the GA performs hardware/software binding and

design space exploration at the same time. Another approach to solve this problem, uses the Problem Space Genetic Algorithm (PSGA) [13] approach. Here the genetic algorithm works on the problem space rather than on the solution space, like it is done normally.

A PSGA model for the integrated partitioning and exploration problem, will start with a genetic algorithm producing chromosomes representing resource sets available to implement the hardware tasks. For each chromosome produced by the PSGA, a hardware performance estimator has to be invoked for all tasks and performance estimates are collected for that resource set. We will then have a standard hardware/software partitioning problem to solve. This approach was discarded because, having a performance estimator inside the genetic algorithm loop will be too time consuming.

Acknowledgements

This work was supported by the ARPA RASSP program and monitored by the Wright Lab, US-AF under contract number F33615-93-C-1316. The authors thank Sriram Govindarajan for his contributions toward hardware design space pruning.

References

[1] R. Ernst, J. Henkel, T. Benner. "Hardware-Software Cosynthesis for Microcontrollers". In *IEEE. Design & Test of Computers*, pages 64–75, December 1993.

[2] R.K. Gupta, G.De Micheli. "Hardware-Software Cosynthesis for Digital Systems". In *IEEE. Design & Test of Computers*, pages 29–40, September 1992.

[3] N. Woo, A.E. Dunlop, W. Wolf. "Codesign from Cospecification". In *IEEE. Computer*, pages 42–47, January 1994.

[4] G. De Micheli. "Computer-Aided Hardware Software Codesign". In *IEEE Micro*, pages 10–16, Aug 1994.

[5] R.K. Gupta, G.De Micheli. "System-level Synthesis using Re-programmable Components". In *Proc. European Design Automation Conference*, pages 2–7, 1992.

[6] J.Hou, W. Wolf. "Process Partitioning for Distributed Embedded Systems". In *Fourth International workshop on Hardware/Software codesign*, pages 70–76, March 1996.

[7] D.E. Thomas J.K. Adams, H. Schmit. "A Model and Methodology for Hardware-Software Codesign". In *IEEE. Design & Test of Computers*, pages 6–15, September 1992.

[8] Petru Eles, Krzysztof Kuchcinski, Zebo Peng, Alexa Doboli. "System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search". In *Design Automation for Embedded Systems, 2, Kluwer Academic Publishers*, pages 5–32, 1997.

[9] V. Catania, M. Malgeri, M. Russo. "Applying Fuzzy Logic To Codesign Partitioning". In *IEEE Micro*, pages 62–70, June 1997.

[10] Stephen A. Blythe, R.A.Walker. "Toward a Practical Methodology for Completely Characterizing the Optimal Design Space". In *9th International Symposium on System Synthesis*, Nov 1996.

[11] Samit Chaudhuri, Stephen A. Blythe, R.A.Walker. "A solution methodology for exact design space exploration in a 3D Design Space". In *IEEE Trans. on VLSI*, March 1997.

[12] J.Septien et. al. "Heuristics for Branch and Bound Global Allocation". In *Proceedings of the European Design Automation Conference*, 1992.

[13] I. Ahmad, M.K. Dhodhi, C.Y.R. Chen. "Integrated Scheduling, Allocation and Module Selection for Design Space Exploration in HLS". In *IEE Proc. on Computers and Digital Tech*, pages 65–71, Jan 1995.

[14] Dutta R., J. Roy and Ranga Vemuri. "Distributed Design Space Exploration for High-Level Synthesis Systems". In *29th Design Automation Conference*, June 1992.

[15] Holland J. *Adaptation in Natural and Artificial Systems.* Ann Arbor: University of Michigan Press, 1975.

[16] F. Sánchez, J. Cortadella. "Resource-constrained software pipelining for high-level synthesis of DSP systems". In *Algorithms and Parallel VLSI Architectures III*, pages 377–388, 1995.

[17] M. Lam. "Software Pipelining: An effective scheduling technique for VLIW machines". In *Proc. of SIGPLAN*, pages 318–328, June 1988.

[18] A. Aiken, A. Nicolau. "Perfect Pipelining: A new loop parallelization technique". In *Lecture notes in Computer Science*, volume 300, pages 221–235, March 1988.

[19] N. Park, A.C. Parker. "Sehwa: A software package for synthesis of pipelines from behavioral specifications ". In *IEEE Trans. on CAD*, volume 7, pages 356–370, March 1988.

[20] *Quantify User's Guide.* Pure Software, Sunnyvale, CA, 1975.

[21] D. Gajski, N. Dutt, A. Wu. *High-Level Synthesis : Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[22] G. De Micheli. *Synthesis and optimization of digital circuits.* Mc Graw Hill, 1994.

[23] Samit Chaudhuri, R.A.Walker. "Computing Lower Bound on Functional Units before Scheduling". In *IEEE Trans. on VLSI*, June 1996.

[24] Sreenivasa Rao D., F.J.Kurdahi. "Heirarchical Design Space Exploration for a Class of Digital Systems". In *IEEE Trans. on VLSI*, 1993.

[25] Cohoon, J., and W. Paris. "Genetic Placement". In *IEEE Trans. on CAD, vol. CAD-6 No. 6*, pages 956–964, Nov. 1987.

[26] Shahookar, K., P. Mazumdar. "A Genetic Approach to Standard Cell Placement using Meta-genetic Parameter Optimization". In *IEEE Trans. on CAD, vol. 9 No. 5*, pages 500–511, Nov. 1990.

[27] Ram Vemuri. *"Genetic algorithms for Partitioning, Placement, and Layer Assignment for Multichip Modules"*. PhD thesis, University of Cincinnati, USA, July 1994.

[28] Goldberg, D.E. *Genetic Algorithms in Search, Optimization, and Machine Learning,.* Addison-Wesley, Reading, MA,, 1989.

[29] Davis, L. *Handbook of Genetic Algotithms.* Van Nostrand, Reinhold, NY, 1989.

[30] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi. "Optimization by Simulated Annealing". In *Science, vol 220, no.4598,*, pages 671–680, 1983.