

A Hardware-Software Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems

Bingfeng Mei^{†‡} Patrick Schaumont Serge Vernalde[†]

[†] IMEC vzw, Kalpedreef 75, B-3001, Leuven, Belgium

[‡] Department of Electrical Engineering, K. U. Leuven, B-3001, Leuven, Belgium

Phone: +32(0)16 281 835 Fax: +32(0)16 281 515

E-mail: {bennet, schaum, vernalde}@imec.be

Abstract— Dynamically reconfigurable embedded systems (DRESSs) target an architecture consisting of general-purpose processors and field programmable gate arrays (FPGAs), in which FPGAs can be reconfigured in run-time to achieve cost saving.

In this paper, we describe a hardware-software partitioning and scheduling approach for DRESSs. Previous work only took configuration time into account, without considering partial reconfiguration. With partial reconfiguration, the scheduling on FPGAs becomes a constrained placement problem, whereas scheduling on application-specific integrated circuits (ASICs) is a serialization problem. Here we present a method, based on genetic algorithm (GA) and an improved list scheduling algorithm, which can tackle multi-rate, real-time, periodic systems. The different variants of algorithm are evaluated by a randomly generated testbench.

Keywords— Dynamic reconfiguration, FPGA, co-design, Partitioning, Scheduling

I. INTRODUCTION

Embedded systems perform application-specific functions using both programmable processor and hardware. Here the hardware can be ASICs, gate arrays or FPGAs. Until recently, FPGAs are only used in prototyping of ASIC designs and low-volume production, mostly because of their low speed and high per-unit cost. However, with the technology improvement of FPGAs, soaring nonrecurring engineering (NRE) cost and shorter time-to-market requirements, there is an increasing interest of using FPGAs in embedded systems due to their advantages over ASICs in terms of flexibility and zero NRE cost[1]. Embedded systems employing FPGAs have the potential to be reconfigured in run-time to achieve further cost saving. A number of companies have released products which are both partially and dynamically reconfigurable[2], [3], [4], and some dynamically reconfigurable systems [5], [6], [7] have been suggested. However, most of above work focused on architecture design, ignoring design methodol-

ogy from a system point of view.

Hardware-software co-design research deals with how to design heterogeneous systems. The main goal is to shorten the time-to-market while reducing the design effort and costs of designed products. The normal design flow includes system specification, cost estimation, hardware-software partitioning, co-synthesis and co-simulation. Since DRESSs consist of both processor and FPGAs, they fit into a co-design flow naturally. However, due to reconfigurability of FPGAs, there are some significant differences, especially in partitioning, scheduling and co-synthesis. Since early nineties, there have been considerable research efforts in co-design [8], [9], [10], [11], but none of them can represent the special characteristics of dynamically reconfigurable systems. In this paper, we try to solve hardware-software partitioning and scheduling problems of DRESSs.

In traditional approach, the processor and ASICs can be viewed as sequential processing elements (PEs). Therefore, the task of partitioning and scheduling is to determine task assignment and execution order on the PEs while respecting time constraints. They are known as NP-complete problems, and numerous heuristics have been developed to solve them. In DRESSs, the problems are further complicated by consideration of configuration delay and partial reconfiguration. With the configuration delay, the amount of configuration time a task requires depends on the previous and next task in the FPGAs schedule. With partial reconfiguration, FPGAs are more like resource pools rather than sequential PEs, so the scheduling on FPGAs becomes a constrained placement problem, which is more complicate than the counterpart in general co-design.

Some recent efforts address automatic partitioning and scheduling to reconfigurable architectures. Lin *et al.*[12] developed a compiler aimed on Garp[6] architecture. R.Harr present a compiler called Nimple [13]. These com-

pilars were able to automatically partition a C-like specification on processor-FPGA platform, with the exploration of *instruction level parallelism*(ILP). The methods used by them are not suited to task level partitioning. Kaul *et al.* [14] proposed a ILP based algorithm for temporal partitioning. The algorithm didn't allow hardware-software partitioning and partial reconfiguration, therefore a simple FPGA model is employed. Dave[15] proposed a system called CRUSADE to synthesize tasks on distributed system consisting dynamically reconfigurable FPGAs. Dick and Jha [16] also proposed a similar system called CORDS. Both of them only took configuration delay into account, without regarding partial reconfiguration.

Here we present an algorithm based on GA and an improved list scheduling algorithm. It can automatically partitioning and scheduling multi-rate, periodic systems to processor-FPGA architecture, while respecting time constraints and taking both configuration delay and partial reconfiguration into account.

II. PRELIMINARIES

A. Target Architecture

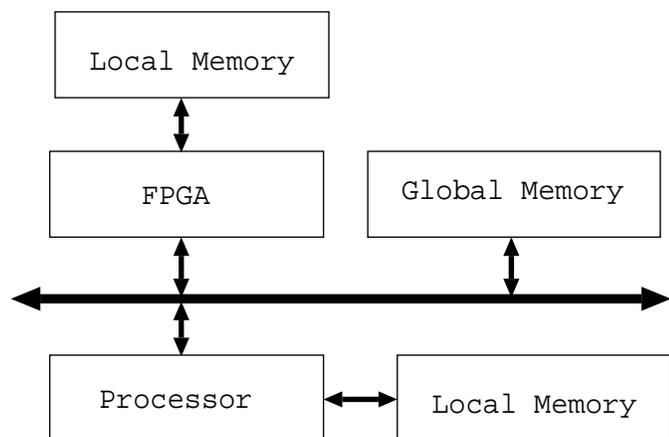


Fig. 1. The target architecture of DRESS

Our algorithm target achitecture is shown in Fig. 1. It contains one processor, one FPGA, one bus and memory hierarchy. The processor can be used to implement control-intensive functions, while an FPGA is employed to accelerate computation-intensive tasks. The bus serves as communication channel connecting processor and FPGA. This architecture and our algorithm can be easily extended to loosely coupled systems, which consist of multiple processors and FPGAs but only one bus.

B. Formulation of Embedded Systems

The functionality of embedded systems is usually described by a set of *task graphs*. The related concepts are

defined as follows.

System: A embedded system is modeled as a set $G = \{G_i; i = 1, \dots, g\}$ of g periodic task graph.

Task graph: A task graph G_i is a directed acyclic graph (DAG) of period τ_i and defined by a 3-tuple $G_i = \{T_i, E_i, \tau_i\}$, where $T_i = \{t_{ij}; j = 1, \dots, n_i\}$ represents the set of tasks, and $E_i = \{e_{ijl}; j, l = 1, \dots, n_i\}$ represents the set of communications.

Task: A task t_{ij} is a node of task graph and defined by a 2-tuple $t_{ij} = \{s_{ij}, h_{ij}\}$, in which s_{ij} and h_{ij} denote the software and hardware implementation respectively.

Communication: The communication e_{ijl} is a directed edge, which means task t_{ij} precedes t_{il} . Each communication is associated with a data transmission time on a specific communication channel.

Deadline: In hard real-time systems, a task t_{ij} might be associated with a deadline $dt(t_{ij}^k)$, which means the task must be finished before the deadline, otherwise the system will fail.

Hyperperiod: Due to the periodic behaviour of task graphs, the study can be restricted to the hyperperiod H , which is the least common multiple of the periods $\tau_i (i = 1, \dots, g)$. Therefore, each hyperperiod will have L/τ_i instances. The instance number $k (k = 1, \dots, L/\tau_i)$ is denoted $G_i^k = (T_i^k, E_i^k)$ with $T_i^k = \{t_{ij}^k\}$ and $E_i^k = \{e_{ijl}^k\}$.

C. Physical Resources

Processor: A processor could be a general-purpose processor, digital signal processor(DSP), or an application-specific instruction set processor(ASIP).It is a sequential device, that says at any time only one task can be carried out on it. For each task, there is an processor execution time $proc_et(t_{ij})$ associated with it.

FPGA: In DRESS, the FPGAs must be dynamically and partially reconfigurable. With partial reconfigurability, a FPGA is essentially different from a processor. It is modeled as a concurrent device, on which some tasks can be performed simultaneously. Each task performed on FPGA has the attributes of execution time $fpga_et(t_{ij})$, length $fpga_l(t_{ij})$, width $fpga_w(t_{ij})$ (if applicable), and configuration time $fpga_ct(t_{ij})$. In some systems [5], [6], to simplify run-time placement, all tasks are specially designed to have same width. Therefore, the width is unnecessary for such case. Similarly, each FPGA is associated with a total length L and a total width W (if applicable).

Bus: The bus connect processor and FPGA. Each communication performed on bus is associated with a data transmission time $bus_et(et_{ijl})$. It is commonly assumed, in distributed systems research, that communication between tasks assign to the same PE is effectively instantaneous, compared with inter-PEs communication. We also

follow this assumption in this paper.

D. Formulation of Solution

With the definition of systems and physical resources, the solution to our problem consists of four parts ($A, S1, S2, S3$):

Task partitioning A , which determines the assignment of each task to either processor p_1 or FPGA p_2 .

$$\forall t_{ij}, t_{ij} \mapsto A(t_{ij}) = p_u, u \in \{1, 2\}$$

Scheduling on processor $S1$, which associates each task instance assigned on processor a start time st .

$$\forall A(t_{ij}) = p_1, t_{ij} \mapsto S1(t_{ij}^k) = st(t_{ij}^k)$$

Scheduling on FPGA $S2$, which associates each task instance assigned on FPGA a start time st , coordinates x and y (if applicable), and a flag f indicates whether the task need configuration before execution.

$$\forall A(t_{ij}) = p_2,$$

$$t_{ij} \mapsto S2(t_{ij}^k) = (st(t_{ij}^k), x(t_{ij}^k), y(t_{ij}^k), f(t_{ij}^k))$$

Scheduling on bus $S3$, which associates each inter-PEs communication a start time st .

$$e_{ijl}^k \mapsto S3(e_{ijl}^k) = st(e_{ijl}^k),$$

$$\{e_{ijl}^k \in E_i^k | A(t_{ij}) \neq A(t_{il})\}$$

while respecting **following constraints**:

Period: Every task can't start earlier than current period.

$$\forall s_{ij}^k, st(t_{ij}^k) \geq (k-1) \times \tau_i$$

Deadline:

$$\forall s_{ij}^k, ft(t_{ij}^k) < \min(k \times \tau_i, dt(t_{ij}^k))$$

with ft as the finish time of the task and dt as the deadline time.

Precedence:

$$st(t_{ij}^k) \geq \begin{cases} ft(t_{il}^k) & \forall t_{il} \in pred(t_{ij}) \cap A(t_{il}) = A(t_{ij}) \\ ft(e_{ilj}^k) & \forall e_{ilj}^k \end{cases}$$

with the $pred(t_{ij})$ as the predecessors of t_{ij} .

III. PARTITIONING

GA is one class of optimization algorithm, which exploits an analogy to natural evolution. Some researchers have applied it in similar problems[17], [18], [19]. However, we have different problem definition and employ a new list scheduling algorithm in the context of DRESSs. In this section, main aspects of our algorithm are described.

A. Chromosome Coding

GA employs the concept *chromosome* to code a solution instead of manipulating the solution itself. The choice of this coding is a main feature of a GA. It will affect the quality of solutions and constrain the choice of genetic operators. As we aim at one-processor-one-FPGA architecture, the chromosome coding in our algorithm is straightforward and easy to extend. Each task t_{ij} is represented by a binary *gene* which denotes the allocation $A(t_{ij})$ for this task (Fig. 2).

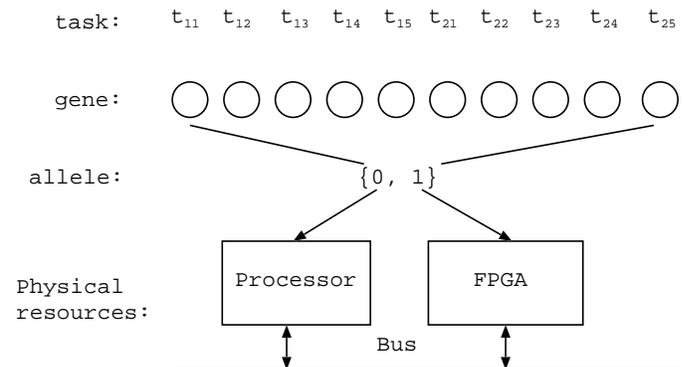


Fig. 2. A chromosome for our partitioning algorithm

In heterogeneous systems, one task has different versions of executable code. For example, the executable code could be a collection of instructions for processor, or bitstreams for FPGAs. If different instances of same task are allocated on different PEs, we have to maintain multiple versions for the same task. It imposes extra requirement on limited memory. Thus, we assign one *gene* to each *task* instead of each *task instance*.

Another advantage of this coding strategy is that genetic operators, such as crossover and mutation, can easily be applied without generating invalid chromosomes. It is also easy to extend to loosely coupled multiple-processor-multiple-FPGA architecture.

B. Cost Function

As GA is an optimization method, one cost function must be defined. Unlike ASICs, where the hardware area is often a cost to be minimized, in DRESSs, we often know which FPGA is on the board. Consequently FPGA area minimization doesn't seem important to take into account. Furthermore, if the area is an objective of optimization, it is difficult to deal with the placement issue, where the physical size of FPGA should be prior knowledge. Here we use the similar cost function as [18], which is based on *accumulated violation*, called *tardiness*. For a partitioned and scheduled system, the violation of task instance t_{ij}^k can be computed as Eq. 1.

$$vio(s_{ij}^k) = \begin{cases} 0 & \text{if } ft(t_{ij}^k) < dt(t_{ij}^k) \\ ft(t_{ij}^k) - dt(t_{ij}^k) & \text{if } ft(t_{ij}^k) > dt(t_{ij}^k) \end{cases} \quad (1)$$

Then the tardiness can be computed as Eq. 2. In this way, the goal is to find a zero tardiness scheduling, *i.e.*, a solution meeting the timing constraints, by minimizing the tardiness function.

$$tard(G) = \sum_k \sum_i \sum_j vio(t_{ij}^k) \quad (2)$$

C. Main Loop

Our GA algorithm is implemented using the PGAPack library [20]. It takes many parameters as inputs. These parameters may affect the partitioning quality and running time. The choice of these parameters are often determined by experience rather than theory. Through experiments, we use these parameters: population size is 100; crossover type is uniform; crossover probability is 0.7; mutation probability is 0.3; maximum generation is 200; the size of new individuals is 50. The main loop is described as follows.

- **Step 1: Initialization.** In order to get diversity in the initial population, each individual(chromosome) is randomly generated, where each gene is set to 0 or 1 with equal probability
- **Step 2: Evaluation and Fitness.** Invoke the scheduler (see section IV) to schedule the tasks and apply Eq. 1,2 to compute the tardiness. The fitness is derived from evaluation using a *linear rank fitness function* given by

$$Min + (Max - Min) \frac{rank(p) - 1}{N - 1} \quad (3)$$

- **Step 3: Selection.** The selection phase allocates reproductive trials to strings on the basis of their fitness. Here we follow the normal tournament selection strategy.
- **Step 4: Crossover and Mutation.** In this phase, crossover and mutation operators are applied on selected parent individuals. The crossover type is set as uniform [21], and the probability is 0.5. The mutation probability is set as 0.3.
- **Step 5: Update Population.** Recalculate the evaluation and fitness value of new individuals. Then according to fitness, discard losers and add new individuals to form new generation.
- **Step 6: Stop Criteria.** If the one of stop criteria is met (find solution or reach the maximum generations), stop to export results; Otherwise repeat step 3 to 6.

A. Description of Problem

As mentioned above, the scheduler serves as a subroutine of partitioning algorithm. It is invoked as the evaluation step of GA. In the context of DRESSs, the scheduling can be divided into two parts. On the processor and bus side, the scheduling keeps the traditional meaning, *i.e.*, to determine the execution order and start time on the sequential devices; On the FPGA side, the scheduling not only has to find out the start time, but also the physical position of tasks on the FPGA with respect to the precedence and resource constraints. In the other words, it can be defined as a *constrained placement problem*, where the placement space is 3-dimensional by taking time dimension into account. An example of scheduled system is shown in Fig. 3.

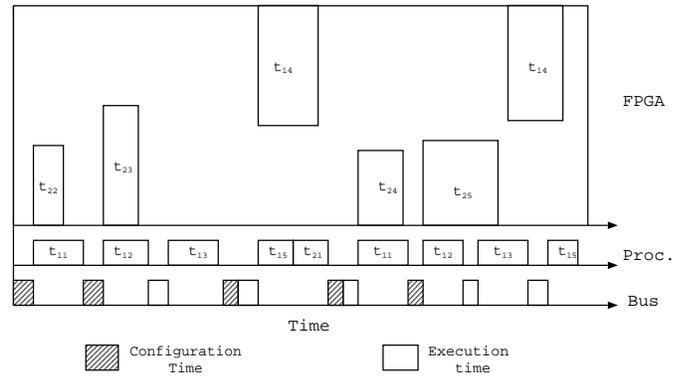


Fig. 3. The scheduling in DRESSs

In Fig. 3, we use tasks of fixed width, therefore, the 3D placement is downgraded to a 2D problem. The task must be placed in the right side of its all predecessors to meet precedence constraints. No two tasks can overlap with each other on placement space. Each task has to be re-configured before execution unless there is same task in same position on FPGA previously.

B. Computation of Priority

In list scheduling, there is a list of unscheduled tasks, which are sorted by some metric, called *priority*. When multiple tasks are ready, the scheduler will select one with the highest priority to run. How to compute priority is the key feature of different list scheduling algorithms. Usual metrics include *as soon as possible*(ASAP) time, *as late as possible*(ALAP) time, *mobility*, etc. These priorities are effective to solve some problems, but often suffer from their static nature, which means the priority is computed before scheduling. During the course of scheduling, the statically computed priority cannot accurately describe the dynamic property of tasks. So far various dynamic priori-

ties have been suggested in literatures. Here we propose a new metrics given by

$$priority(t_{ij}^k) = -(dyna_ASAP(t_{ij}^k) + ALAP(t_{ij}^k)) \quad (4)$$

with *dyna_ASAP* denoting dynamic *ASAP* start time. Unlike static metrics, once a task is scheduled, all *dyna_ASAP* values will be recalculated to reflect the current status. Larger *ASAP* time means the task must be scheduled later, thus has a lower priority. Similarly, the larger *ALAP* time means that the task can be executed later, also should be assigned lower priority. Unlike *ASAP* value, the *ALAP* time doesn't change during the scheduling, therefore, it is still computed statically. The algorithm to compute *ALAP* can be found in [22].

```
for(each_unscheduled_task)
{
  check_if_ready();
  if(task_is_ready())
  {
    finish_time = check_predecessor();
    dyna_ASAP =
      search_FPGA_for_space(finish_time);
    dyna_priority = -(dyna_ASAP + ALAP);
  }
}
```

Fig. 4. Compute dynamic priority

Fig. 4 describes the algorithm of computing dynamic priority. For each ready task, it derives latest finish time from all predecessors. Then it searches the FPGA along the time dimension until it finds enough space to contain the task. This moment is assigned to *dyna_ASAP*, and dynamic priority can be computed accordingly. In next step, the scheduler will select task with the highest priority to place on FPGA.

C. Placement Algorithms

After the compute and select task of the highest priority, the next step is to determine where to place the task on the FPGA. Due to the characteristics of list scheduling, this problem is similar to an *online bin-packing problem*, although in fact the scheduling is conducted offline. Two well-known online algorithms for the 1D bin-packing problem are *first fit* (FF) and *best fit* (BF). The FF algorithm puts the arriving task in the lowest indexed bin that can accommodate the task. The BF algorithm choose the smallest bin which can accommodate the task. Intuitively, BF results in less fragments and can get more compact placement, while FF is much faster than BF algorithm. But

in DRESSs, the differences between them are not so apparent (see Sec. V-B). Both algorithms can be used.

Compared with the offline counterparts such as *simulated annealing* and *greedy algorithms*, which employ iterative methods to improve placement results, online placement algorithm often leads to poorer performance. Its decision is based on incomplete knowledge, and has no chance to improve the placement quality during the course of scheduling.

D. Reducing Configuration Overhead

Configuration overhead is a main factor that prevents dynamic reconfiguration from being accepted. With the advance of FPGAs technology, the reconfiguration time is reduced dramatically. However, compared with ever increasing FPGAs speed, the delay is still quite considerable. Though the delay is mainly determined by characteristic of target FPGA, by proper partitioning and scheduling algorithm, we still can reduce some configuration costs. This approach is first exploited by R.P.Dick *et.al.*[16]. Here we extend it to the case of partial reconfiguration.

```
compute_dynamic_priority();
check_same_task_on_FPGA();
if(found_same_task())
{
  dyna_ASAP = end_of_same_task();
  tmp_priority =
    -(dyna_ASAP + ALAP - C * conf_time);
  if(tmp_priority > dyna_priority)
  {
    dyna_priority = tmp_priority;
    save_time_and_position();
  }
}
```

Fig. 5. Ajust dynamic priority with configuration delay

The algorithm is shown in Fig. 5. The basic idea is that priorities are ajusted dynamically regarding configuration possibilities. If a existing configuration can be reused, then this task will be assigned to higher proper priority. The *C* in Fig. 5 is a weight number, which can be choosed in different applications.

E. Complete Algorithm

The inputs of scheduling algorithm are partitioned tasks, while the outputs are scheduled (placed) tasks. The main loop of scheduling algorithm is described as Fig. 6

In first step, all intra-PE communications are removed, and new task graphs are constructed where all communication events are viewed as normal tasks assigned on bus. Then, *ALAP* values are computed, and an unscheduled

```

remove_intra-PEs_communication();
construct_new_task_graphs();
compute_ALAP()
construct_unscheduled_list();
do
{
    search_ready_tasks();
    compute_dyna_priority();
    ajust_priority_with_configuraion();
    select_task_with_highest_priority();
    placement_algorithm();
    remove_task_from_unscheduled_list();
}while(!unscheduled_list_is_empty())

```

Fig. 6. Scheduling algorithm

task list is constructed sorted by decreasing *ALAP* values. In each iterative, the scheduler computes dynamic priority, ajusts priority with configuration delay, and invokes placement subroutine respectively. The finish condition of loop is that all tasks are scheduled.

V. EXPERIMENTAL RESULTS

A. The testbench

Since our algorithm is the first to address similar problem, we can't compare it with other algorithms. Experiments are only conducted on different versions of our algorithm. We constructed a testbench consisting 300 systems, which are randomly generated by TGFF[23]. The systems are composed of 3, 4, or 5 task graphs. Each task graph contains an average of 10 tasks, with the variability of 5. According to A. DeHon[24], FPGA implementation is often 10 times faster than the processor version for the same task. Thus we assume each task takes an average execution time of 200us on the processor, with the variability of 150 us, and the average execution time on FPGA is 20us with the a variability of 15us. We also assume the target FPGA is XC6264, which contains 16384 CLBs and requires 81.92us to configure entire chip. The average number of CLBs occupied by a task is 2500, with a variability of 2000. The communication on bus takes an average time of 15us, with a variability of 10us.

B. The results

We have submitted the same testbench to different versions of our algorithm. They are: ALAP based priority, dynamic priority, dynamic priority with configuration delay, and dynamic priority with both configuration delay and BF strategy.

The overall results are shown in Fig. 7. As expected, the

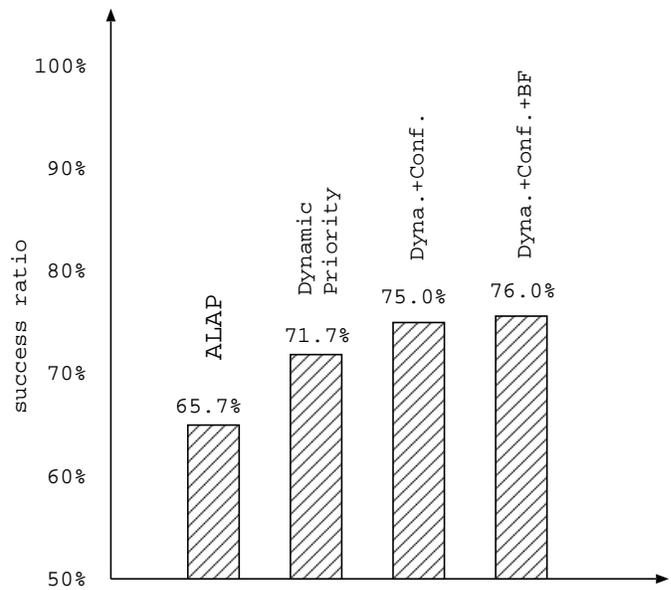


Fig. 7. The results of different variants of our algorithm

results reveal that the last algorithm outperformed other three algorithms, but the difference between these algorithms is not so large. Partially because randomly generated systems are highly diversified, so a large number of systems are infeasible on target platforms, even with the optimal algorithm. Another reason is that all these algorithms employ same GA framework, only vary in the the scheduling algorithms.

VI. CONCLUSIONS

In this paper, we proposed an algorithm to solve the hardware-software partitioning and scheduling problem for DRESSs. At our best knowledge, it is the first algorithm that tackles dynamically reconfigurable systems, while taking hardware-software partitioning, partial reconfiguration and configuration overhead into account. The objective is to find a feasible partitioning and scheduling with the respect to all constraints: precedence, deadline, periods and resource. The uniqueness of this problem is that scheduling on FPGA is a constrained placement problem rather than an ordering issue. We suggest an approach which combines a "standard" *genetic algorithm* and an *improved list scheduling algorithm*. The list scheduling algorithm includes a new metric to compute dynamic priority, dynamic adjustment for configuration delay, and BF placement strategy. The experimental results indicate that the algorithm containing all features outperform other variants.

However, our algorithm has some obvious disadvantages: It is only suited to loosely coupled system, but cannot handle hierarchical systems; Unlike simulated an-

nealing placement method, it suffers from possible inefficiency; it cannot automatically allocate FPGAs and processors, etc. We are going to overcome these drawbacks in future work.

In future, we also plan to integrate the algorithm to our OCAPI-XL tools. It is supposed to become one step of our C++ based hardware-software co-design flow.

REFERENCES

- [1] S. Davis, *Total Cost of Ownership: Xilinx FPGAs vs. Traditional ASIC Solutions*. Xilinx, Inc., Feb. 2000.
- [2] Xilinx, Inc., *Vitex 2.5V Field Programmable Gate Arrays*, May. 2000.
- [3] Atmel, Inc., *AT40K FPGAs with FreeRAM*, Jan. 1999.
- [4] Atmel, Inc., *Coprocessor Field Programmable Gate Arrays*, Oct. 1999.
- [5] M. J. Wirthlin and B. L. Hutchings, "A dynamical instruction set computer," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 99–107, 1995.
- [6] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 12–21, 1998.
- [7] J. Villasenor, B. Schoner, K.-N. Chia, C. Zapata, H. J. Kim, C. Jones, S. Lansing, and B. Mangione-Smith, "Configurable computing solutions for automatic target recognition," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 70–79, 1996.
- [8] R. Ernst, J. Henkel, and T. Benner, "Hardware/software cosynthesis for microcontrollers," *IEEE Design & Test of Computers*, vol. 10, pp. 64–75, Dec. 1993.
- [9] R. K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. PhD thesis, Stanford University, 1994.
- [10] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Prentice Hall, 1994.
- [11] I. Bolsens, H. De Man, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest, "Hardware-software codesign of digital telecommunication systems," *Proceeding of IEEE*, vol. 85, pp. 391–418, Mar. 1997.
- [12] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-software co-design of embedded reconfigurable architectures," in *Proc. Design Automation Conference*, 2000.
- [13] R. Harr, "The nimble compiler for agile hardware: A research platform," in *Proc. 13th International Symposium on System Synthesis*, 2000.
- [14] M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouais, "An automated temporal partitioning and loop fission approach for fpga based reconfigurable synthesis of dsp applications," in *Proc. Design Automation Conference*, pp. 616–622, 1999.
- [15] B. P. Dave, "CRUSADE: Hardware/software co-synthesis of dynamically reconfigurable heterogeneous real-time distributed embedded systems," in *Proc. Design, Automation and Test in Europe Conference*, pp. 97–104, 1999.
- [16] R. P. Dick and N. K.Jha, "CORDS: Hardware-software co-synthesis of reconfigurable real-time distributed embedded systems," in *Proc. Int. Conf. on Computer-Aided Design*, pp. 62–68, 1998.
- [17] R. P. Dick and N. K.Jha, "MOGAC: A multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems," *IEEE Trans. on Computed-Aided Design of Integrated Circuits and Systems*, vol. 17, pp. 920–935, Oct. 1998.
- [18] Y. Monnier, J.-P. Beauvais, and A.-M. Deplanche, "A genetic algorithm for scheduling tasks in a real-time distributed system," in *Proc. 24th EUROMICRO Conf.*, pp. 708–714, 1998.
- [19] M. Grajcar, "Genetic list scheduling algorithm for scheduling and allocation on a loosely coupled heterogeneous multiprocessor system," in *Design Automation Conference*, pp. 280–284, 1999.
- [20] D. Levine, "Users guide to the PGAPack parallel genetic algorithm library," tech. rep., Argonne National Laboratory, 1996.
- [21] R. Niemann, *HW/SW Co-design for data flow dominated embedded systems*. Kluwer Academic Publishers, 1998.
- [22] M. you Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 1, pp. 330–343, Jul. 1990.
- [23] R. P.Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task graphs for free," in *Proc. Int. Workshop Hardware/Software Codesign*, pp. 97–101, 1998.
- [24] A. DeHon, "Comparing computing machines," in *Configurable Computing: Technology and Applications*, vol. 3526 of *Proc. of SPIE*, pp. 124–133, 1998.