Hardware-Software Cosynthesis for Run-time Incrementally Reconfigurable FPGAs [*]

Byungil Jeong   Sungjoo Yoo   Sunghyun Lee   Kiyoung Choi

Design Automation Lab.
School of Electrical Engineering
Seoul National University
Seoul 151-742, Korea
e-mail: {bijeong,ysj,pontneuf,kchoi}@poppy.snu.ac.kr

Abstract

This paper presents a method for hardware-software cosynthesis with run-time incrementally reconfigurable FPGAs. To reduce the run-time overhead of reconfiguring FPGAs, we present a concept called early partial reconfiguration (EPR) which minimizes the overhead by performing reconfiguration for an operation (or a task in our terms) mapped to an FPGA as early as possible so that the operation is ready to start when its execution is requested. For further reduction of the overhead, we integrate the incremental reconfiguration (IR) of FPGAs with the EPR concept. We present an ILP formulation and an efficient heuristic algorithm based on the EPR and IR concepts. Experiments on embedded system examples and synthetic examples show the efficiency of the proposed method

I. Introduction

As multi-million gate FPGA era is expected, the integration of FPGA with CPU, DSP, and memory is an appealing solution to the implementation of embedded systems. Especially, by integrating FPGAs into a system on a chip (FPSC: Field Programmable System on a Chip), we can achieve performance and flexibility at the same time. We can achieve performance through parallel and accelerated execution of operations on the FPGAs. We can achieve flexibility through reconfiguration of the FPGA.

However, we can expect a problem due to the limited FPGA resource that can be integrated on a chip. Such a problem can be alleviated by increasing the utilization of the limited FPGA resource through dynamic change of the FPGA configuration for newly requested operations. However, changing the configuration of an FPGA requires mega bits of configuration data and can consume most of system cycles (25 % to over 70 % [1]), which we call *reconfiguration time overhead.* Thus, to fully utilize the computing power of reconfigurable FPGAs, we need to devise new methods that minimize the reconfiguration time overhead.

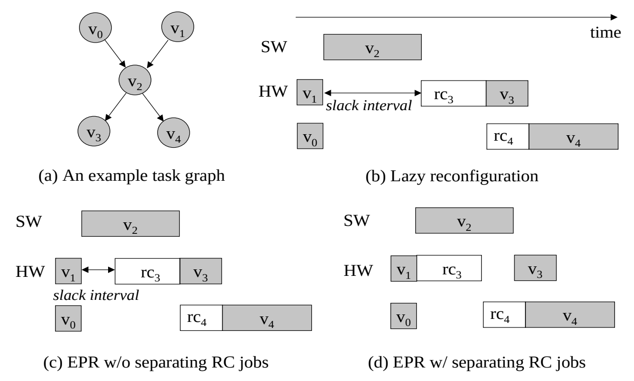
In designing a system having a reconfigurable FPGA and processor core(s), the problem that which part of the system functionality should be mapped to the reconfigurable FPGA (i.e. the HW part) or the processor core(s) (i.e. the SW part(s)) is a well-known HW-SW partitioning problem. To minimize the reconfiguration time overhead and to better utilize the reconfigurable computing power of FPGA, the reconfiguration time overhead should be considered in the HW-SW partitioning and scheduling, i.e. cosynthesis step.

To minimize the reconfiguration time overhead we apply two concepts called *early partial reconfiguration* (EPR) and *incremental reconfiguration* (IR) to the cosynthesis step. With the EPR concept, we perform reconfiguration for an operation mapped to an FPGA as early as possible so that the operation is ready to start when its execution is requested. This strategy saves much time compared to the lazy reconfiguration which starts reconfiguration of an operation when its execution is requested.

The idea of reducing the reconfiguration time overhead by performing configuration earlier than when computation is required is similar to that of Hauck's work [2]. The target architecture of his approach has a processor and a reconfigurable FPGA as a coprocessor. He performs the next needed reconfiguration in advance to overlap the computation of the processor and the reconfiguration of the FPGA. The approach assumes single context of FPGA configuration, i.e. there are no multiple computations running concurrently on the FPGA. In addition, the computation of the FPGA and that of the processor do not run concurrently. We apply the notion of prefetching configuration to the target architecture that is composed of a CPU and a reconfigurable FPGA, where multiple computations can run concurrently on the FPGA along with the computation of CPU.

With incremental reconfiguration, we can reduce the amount of reconfiguration data required during the reconfiguration. Dick and Jha [3] present a scheduling method which tries to schedule the tasks of the same type con-

[*]This work was supported in part by KOSEF, Korea.



(a) An example task graph (b) Lazy reconfiguration

(c) EPR w/o separating RC jobs (d) EPR w/ separating RC jobs

Fig. 1. Early partial reconfiguration and splitting reconfiguration jobs.

secutively thereby eliminating the reconfiguration of the successor tasks of the same type. We further reduce the reconfiguration time overhead by performing incremental reconfiguration of tasks which partially share configuration data with tasks that have already been configured.

In this paper, we present an Integer Linear Programming (ILP) formulation and an efficient heuristics for the HW-SW cosynthesis exploiting the EPR and IR concepts. In Section II, we describe the EPR and IR concepts. We define the cosynthesis problem and give a brief description of our ILP formulation in Section III. We present a cosynthesis heuristics in Section IV. We compare the experimental results of the heuristics and the ILP formulation in Section V. We conclude in Section VI.

II. Motivation

A. Early Partial Reconfiguration

Figure 1 illustrates our basic idea of EPR. We assume that a task graph is given as a part of the system specification as shown in Figure 1 (a). An acyclic task graph G(V,E) consists of V (set of tasks) and E (set of edges). An edge between two tasks v_i and v_j represents that the execution of v_j gets ready to start only after that of v_i finishes. The computation time of a task graph is the time duration it takes to perform the computations of all tasks in it. We also assume that there is only one reconfiguration controller which performs reconfiguration jobs (in our example, rc_3 and rc_4).

Figure 1 (b) shows an example of mapping tasks to reconfigurable HW or SW and scheduling them. In the figure, blank and shaded rectangles represent reconfiguration and computation jobs of tasks, respectively. In this example, task v_2 is assumed to be mapped to SW (probably due to the high HW implementation cost of v_2) and tasks v_0, v_1, v_3 and v_4 to HW (probably thanks to their shorter computation times when mapped to HW). Since tasks v_3 and v_4 are reconfigured in a lazy manner, there is a slack interval in HW during the computation of SW task v_2 as shown in Figure 1 (b).

The EPR concept exploits such slack interval. Figure 1 (c) shows that early execution of the reconfiguration job rc_3 of task v_3 during the slack interval can hide the reconfiguration time of task v_3. For further reduction of reconfiguration time, as shown in Figure 1 (d), we separate the reconfiguration jobs and their corresponding computation jobs, to allow another reconfiguration job to be performed utilizing the remaining slack interval. In our example, we separate rc_3 and v_3 to execute rc_4 during the remaining slack interval.

B. Incremental Reconfiguration

In Figure 1 (d), to further reduce the reconfiguration time overhead, we can perform reconfiguration of task v_3 or v_4 incrementally. To do that, we should consider the previous configuration of the FPGA (v_0 and v_1 in this example) before v_3 or v_4 can be reconfigured. In the example of Figure 1 (d), if we assume that task v_3 shares more configuration data, for example, 70 % of configuration data of task v_3 with task v_1 than task v_4 (for example, 20 %), we can reduce the reconfiguration time overhead of task v_3 by reconfiguring only the remaining 30 % of its configuration utilizing the configuration of task v_1, i.e. by incremental reconfiguration.

In our work, we incorporate the EPR and IR concepts into the trade-off calculation of hardware-software partitioning thereby the early partial and incremental reconfiguration of FPGA can be best exploited.

III. Problem Definition and An ILP Formulation

A. Problem Definition

We assume that the target architecture has a processor and a reconfigurable FPGA both of which are connected by a communication bus. The FPGA can be an isolated chip or a core in an FPSC (Field Programmable Systems on a Chip) implementation. We also assume that reconfiguration is performed by an independent controller.

In this section, we present an ILP formulation to solve the following HW-SW cosynthesis problem.

Problem 1 *Given G(V,E) and HW resource constraint A^{HW}, map tasks to HW and SW and schedule them to minimize T^G the computation time of G.*

B. An ILP Formulation

In this subsection, we give a brief explanation of our ILP formulation. We formulate the ILP solution with constraints on (1) task start time and finish time constraints, (2) data dependency constraints, (3) serialization constraints on the SW processor, the communication bus, and the reconfiguration controller, and (4) FPGA resource constraint. In this paper, we explain a key constraint related to the EPR and IR concepts. [4] gives a detailed description of the ILP formulation.

Since the EPR concept permits reconfiguration jobs to be separated from computation jobs and to be placed in advance, we set constraints on the start/finish time when each task starts/finishes occupying the computation resource (HW or SW). For example, in Figure 1 (d), task v_3 has the following constraint.

$$\begin{aligned} T_F(3) - T_S(3) \quad \geq \quad & rc(3) \cdot s(3) + C^H(3) \cdot m(3) + \\ & C^S(3) \cdot (1 - m(3)) - 1 \end{aligned} \tag{1}$$

In Inequality (1), $T_S(3)$, $T_F(3)$, $rc(3)$, $C^H(3)$, and $C^S(3)$ represent, respectively, start time of resource occupation, finish time of resource occupation, reconfiguration run-time, HW computation time, and SW computation time of task v_3, $m(3)$ is a binary variable representing that task v_3 is mapped to HW ($m(3) = 1$) or SW ($m(3) = 0$). If a task v_i is mapped to HW, its start time of resource occupation $T_S(i)$ is the time when the reconfiguration of the task begins. In Inequality (1), $s(3)$ represents the percentage of incremental reconfiguration to be performed for task v_3. For example, if $s(3) = 0.3$, only 30 % of the configuration of task v_3 needs to be reconfigured. For a task v_i, $s(i)$ is defined to be $m(i) - \sum_{j \neq i} \rho(j,i)\pi(j,i)$, where $\rho(j,i)$ is the percentage of shared configuration data between two tasks v_j and v_i in the viewpoint of task v_i (in Figure 1, if $\rho(1,3) = 0.7$, then task v_3 shares 70 % of its configuration data with task v_1), and $\pi(j,i)$ is a binary variable representing that task v_i can be reconfigured just after the computation of task v_j finishes (in Figure 1 (b), (c), and (d), $\pi(1,3) = 1$). If $m(i) = 1$, only one case of $\pi(j,i)$ is set to 1 to allow incremental reconfiguration. Else (i.e. if $m(i) = 0$), all $\pi(j,i)$'s are set to zero. That is, $\forall v_i, \sum_{\forall v_j} \pi(j,i) = m(i)$.

IV. Proposed HW-SW Cosynthesis Heuristics with the EPR and IR concepts

In this section, we present a heuristics to solve the HW-SW cosynthesis problem defined in Section III-A.

A. Overall Strategy

To move tasks between HW and SW, we adopt the Fiduccia/Mattheyses (FM) algorithm-based HW-SW partitioning [5] as shown in Figure 2. We apply the EPR and IR concepts to the scheduling step of GetScheduleLength() (on line 24 in Figure 2). In GetScheduleLength(), we perform a list scheduling based on the earliest computation start times (ECST's) of ready tasks. To be specific, we select the task that can start its computation at the earliest time, i.e. that has the smallest ECST among ready tasks (SelectATask()) and schedule its (reconfiguration job, if it is mapped to HW, and) computation job (ScheduleATask(selected)).

```
1.  currP = bestP = InitialP();
2.  IterationLoop: loop
3.      everbestP = bestP;
4.      while (UnlockedTaskExist) loop
5.          moved = SelectNextMove();
6.          currP = MoveAndLockTask(moved);
7.          bestP = GetBetterPart(currP,bestP);
8.      end loop;
9.      if no improvement of $T^G$ in this pass then
10.         return everbestP;
11.     else // Do another pass
12.         UnlockAllTasks();
13. end loop;

14. SelectNextMove() {
15.     best_move = 0; best_TG = max number;
16.     for task $v_i$ in all tasks
17.         TryMove($v_i$);
18.         TG = GetScheduleLength();
19.         if( best_TG > TG )
20.             best_move = $v_i$; best_TG = TG;
21.         RestoreMove($v_i$);
22.     end for;
23.     return best_move; }

24. GetScheduleLength() {
25.     CalculateECST();
26.     while (UnscheduledTaskExist) loop
27.         selected = SelectATask();
28.         ScheduleATask(selected);
29.         UpdateECST();
30.     end loop;
31.     return $T^G$; }
```

Fig. 2. A pseudo code of the proposed HW-SW cosynthesis heuristics.

In our cosynthesis heuristics, since the key point is applying the EPR and IR concepts to the calculation of the ECST of each task, we focus on the description of calculating the ECST.

B. Calculating Earliest Computation Start Time

To exploit the EPR and IR concepts, we define a *present HW configuration set* PCS^{HW} as the set of HW tasks which currently occupies HW resource. For example, in Figure 3 (a), since tasks v_0 and v_1 (in the task graph of Figure 1 (a)) are mapped to HW, $PCS^{HW} = \{v_0, v_1\}$. In Figure 3, the height of each rectangle represents the size, i.e. the implementation cost of each HW task.
The ECST of task v_i is determined by the maximum of the time when data from its direct predecessors are ready to be used (which we call *data ready time* $DRT(i)$) and the time when the computation resource (HW or SW) is ready

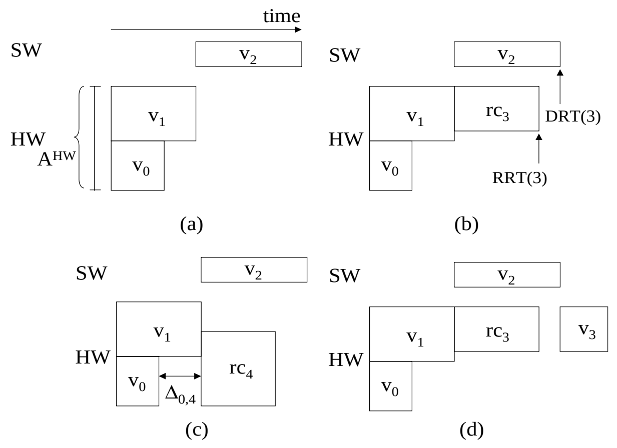


Fig. 3. Examples of scheduling HW tasks.

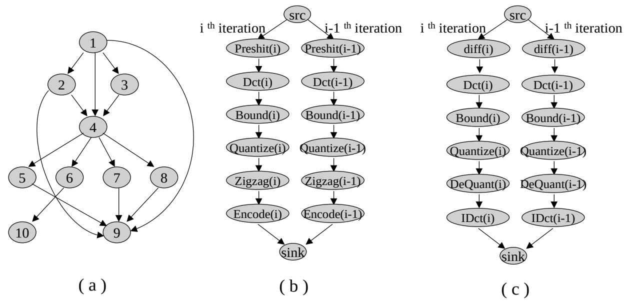


Fig. 4. Task graphs : (a) a synthetic task graph, (b) the JPEG example, and (c) the H.263 example.

to be used (which we call *resource ready time* $RRT(i)$). For HW task v_i, $RRT(i)$ is the time point when the reconfiguration of HW task v_i is completed. Note that **when $\mathbf{DRT(i) \geq RRT(i)}$, the reconfiguration overhead of HW task v_i is totally hidden by EPR**.

Figure 3 (b) illustrates examples of DRT and RRT values for task v_3 in the task graph of Figure 1 (a). In the figure, $DRT(3)$ is the finish time of the predecessor task v_2 and $RRT(3)$ is the finish time of reconfiguration job rc_3 of task v_3 assuming communication time between tasks v_2 and v_3 is zero. Since $DRT(3) > RRT(3)$, $ECST(3) = max\{DRT(3), RRT(3)\} = DRT(3)$, i.e. the reconfiguration time overhead for task v_3 is totally hidden by EPR as shown in Figure 3 (b).

For a task v_i, $DRT(i)$ is determined as follows.

$$DRT(i) = \max_{v_j \in Pred(i)} \{T_F(j) + comm(j,i)\} \qquad (2)$$

where $Pred(i)$ is the set of direct predecessor tasks of task v_i and $comm(j,i)$ is the communication time between task v_j and v_i. Since the predecessor task v_j is already scheduled, its finish time $T_F(j)$ is obtained.

If task v_i is mapped to SW, $RRT(i)$ is determined to be the maximum of finish times of tasks scheduled on SW before task v_i. If task v_i is mapped to HW, we should consider PCS^{HW} in the computation of $RRT(i)$ as follows.

$$RRT(i) = \min_{v_j \in PCS^{HW}} \{T_F(j) + (1 - \rho(j,i)) \cdot rc(i) + \Delta_{j,i}\} \qquad (3)$$

where $rc(i)$ is the reconfiguration time of task v_i. In the equation, the term $(1 - \rho(j,i)) \cdot rc(i)$ implies the reconfiguration time overhead when task v_i is incrementally reconfigured utilizing the configuration of task v_j. $\Delta_{j,i}$ is exemplified in Figure 3 (c). In the figure, if task v_4 shares more configuration data with task v_0 than with task v_1, we can try reconfiguring task v_4 just after the computation of task v_0 finishes. However, in this case, since the sum of the size of task v_4 and that of task v_1 exceeds the given HW cost constraint A^{HW}. Thus, the reconfiguration of task v_4 is delayed by the amount of $\Delta_{0,4}$ until task v_1 releases the HW resource. For another example, in the case of Figure 3 (b), since the size of task v_3 is smaller than that of task v_1, $\Delta_{1,3} = 0$.

As shown in the above examples, to determine $\Delta_{j,i}$, we should consider two cases. Case I (II) represents a case where task v_i shares configuration data with task v_j $\in PCS^{HW}$ and the HW cost of task v_j is larger than or equal to (smaller than) that of task v_i. In Case I, $\Delta_{j,i} = 0$. In Case II, $\Delta_{j,i}$ is calculated as the time interval for which we have to wait to obtain enough HW area for reconfiguring task v_i. Figure 3 (d) depicts the result of scheduling the reconfiguration and computation jobs of task v_3.

V. Experiments

We applied the ILP formulation and the proposed heuristics to a set of synthetic task graphs and two real examples (a JPEG encoder [6] and an H.263 encoder [7]).

A. Synthetic Task Graphs

We used synthetic task graphs to investigate (1) the performance gain obtained by applying the EPR and IR concepts varying HW resource constraints and reconfiguration time overhead and (2) the run-time of the heuristics for large task graphs. To generate synthetic task graphs, we used a task graph generator, TGFF [8]. TGFF also generates parameters for each task (HW and SW computation times, HW implementation cost, etc.). We added reconfiguration time to each task in the generated task graphs. Since reconfiguration time is generally proportional to the hardware resource usage, reconfiguration time was assigned to each task in proportion to the hardware implementation cost of the task. We used reconfiguration-to-computation ratio denoted by R to vary the reconfiguration time overhead in the synthetic task graphs. We set $\rho(j,i)$ for each pair of tasks v_j and v_i to a value between 0 % and 40 %. We also varied the value of A^{HW}.

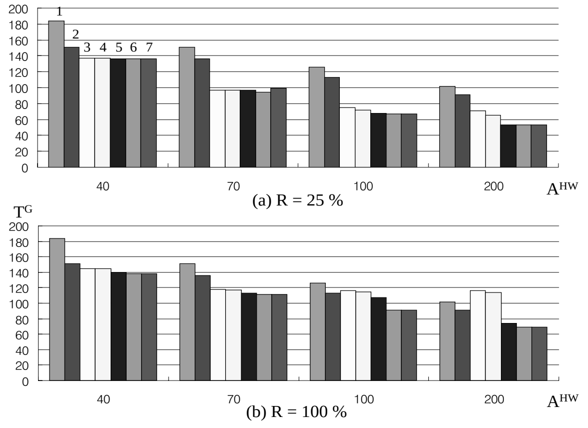


Fig. 5. Comparison of the finish times of a synthetic task graph.

We applied the ILP formulation to the generated task graph under six different conditions. Table I shows the condition of each case. In Table I, RRC represents run-time reconfiguration. In the cases that run-time reconfiguration is not exploited, we used static configuration i.e. HW resource occupied by a HW task is not reconfigured for another HW task throughout the run-time. However, HW resource occupied by a HW task may be re-used by another task of the same type if the Same Type condition is 'Yes' (fourth column in the table). To solve ILP problems, we used a commercial ILP solver, CPLEX. We also applied the proposed heuristics (case No. 7 in Table I).

Figure 5 (a) and (b) show the comparison of the finish time (T^G) of a synthetic task graph (shown in Figure 4 (a)) for two cases of R (=25 % and 100 %). R=25 % represents the average reconfiguration time of a HW task is 25 % of its HW computation time. Each vertical bar represents T^G for each of the seven cases with given A^{HW}.

TABLE I
Types of HW-SW cosynthesis.

No.	ILP/Heu	RRC	Same Type	EPR	IR
1	ILP	No	No	No	No
2	ILP	No	Yes	No	No
3	ILP	Yes	No	No	No
4	ILP	Yes	Yes	No	No
5	ILP	Yes	Yes	Yes	No
6	ILP	Yes	Yes	Yes	Yes
7	Heu	Yes	Yes	Yes	Yes

Figure 5 (a) shows that as the EPR and IR concepts are applied, we obtain shorter T^G, i.e. more performance improvement. In some case (A^{HW}=100 and 200) of Figure 5 (b), since the reconfiguration time overhead gets comparable to HW computation times of tasks (R=100 %) and large A^{HW} values allow more tasks to be mapped on HW, when the EPR and IR concepts are not used, run-time reconfiguration yields worse performance than the static configuration. However, by applying the EPR and IR concepts, we could obtain performance improvement.

The algorithm complexity of the proposed heuristics is O(n^5).[1] Table II shows the run-times of the proposed heuristics for large-sized task graphs generated by TGFF on an Ultra II workstation (200 MHz, 512 MB main memory). The ILP formulation for EPR and IR concepts hardly gives solutions for task graphs having more than 10 tasks.

TABLE II
Run-times of the proposed heuristics.

No. of tasks	50	100	150	200
Run-time (sec)	22	930	6,034	16,954

B. JPEG and H.263 Examples

The JPEG and H.263 examples contain, respectively, 6 sequential tasks in each iteration of encoding. Since we perform two iterations of encoding concurrently, there are 12 tasks as shown in Figure 4 (b) and (c), respectively.

For the JPEG and H.263 examples, we used a 32bit RISC microprocessor (ARM7 [9]) as the SW processor and a run-time reconfigurable FPGA (AT40K40 from Atmel Co. [10]). Table III shows the parameters of tasks in the JPEG example (and some tasks in the H.263 example) : HW cost (size of configuration data in bytes), HW computation time (C^H), SW computation time (C^S), and reconfiguration time (rc). C^H, C^S and rc are in microseconds.[2] Since the same amount of data (64x16 bits) are transferred between neighboring tasks in the JPEG and H.263 examples (Figure 4 (b) and (c)), communication time between a SW task and a HW task is also the same and set to 10 μs. Table IV gives the ratio of configuration data sharing ρ between every pair of tasks (previous tasks are in the first column) in the JPEG example.

TABLE III
Task parameters of the JPEG example.

	HW cost	C^H	rc	C^S
Pre	3,545	16	53	65
DCT	26,203	273	543	355
Bnd	3,689	73	70	98
Qtz	14,107	23	211	233
Zig	6,527	59	97	80
Enc	36,371	62	545	551

Figure 6 (a) and (b) show the comparison of T^G in the JPEG and H.263 examples. For the case of A^{HW} = 37 Kbytes, the proposed heuristics gives up to 41.1 % and

[1]In Figure 2, since the two while loops (line 4 to 8 and line 26 to 30) and the for loop (line 16 to 22), respectively, have O(n) complexity and UpdateECST() has O(n^2) complexity.

[2]We set the system clock frequencies of ARM7, FPGA computation, and FPGA reconfiguration to 25 MHz, 5 MHz, and 33 MHz, respectively. We obtained the HW and SW computation times of each task and communication times by running the instruction-set simulation of ARM7 processor [11] and VHDL simulation.

TABLE IV
THE RATIO OF CONFIGURATION DATA SHARING BETWEEN EVERY PAIR OF TASKS IN THE JPEG EXAMPLE.

	Pre	DCT	Bnd	Qtz	Zig	Enc
Pre	1	0	0.09	0.02	0.07	0
DCT	0.35	1	0.32	0.28	0.31	0.21
Bnd	0.33	0.02	1	0.08	0.27	0.02
Qtz	0.27	0.06	0.33	1	0.29	0.08
Zig	0.27	0.02	0.30	0.07	1	0.02
Enc	0.31	0.18	0.37	0.30	0.31	1

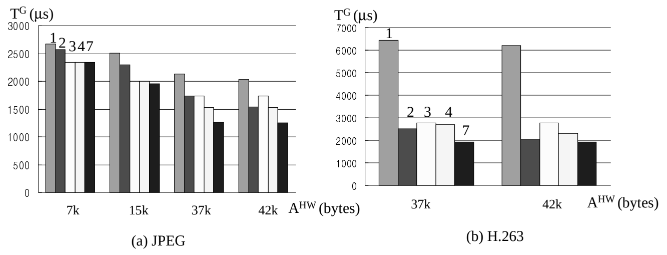


Fig. 6. Comparison of the finish times of the JPEG and H.263 examples.

27.4 % (70.1 % and 23.4 %) performance improvement for the JPEG (H.263) example compared to the ILP solutions of static configuration without same type (vertical bar numbered 1) and with same type (vertical bar numbered 2), respectively. Due to the long run-time, we could not obtain ILP solutions for case No. 5 and 6 where the EPR and IR concepts are applied.

Figure 7 shows the result of HW-SW partitioning and scheduling for the JPEG example obtained by the proposed heuristics when $A^{HW} = 42$ Kbytes. In the figure, dashed areas in HW represent reconfiguration jobs and shaded areas in HW represent computation jobs. The figure shows that the reconfiguration times of HW tasks Bound(i-1), Quantize(i-1) and Zigzag(i-1) are hidden by EPR. Note that the reconfiguration time of HW task Encode(i) is eliminated since the configuration of Encode(i-1) is re-used for Encode(i). The reconfiguration time of HW task Encode(i-1) is also reduced by IR utilizing the configuration of HW task Quantize(i-1) (8 % of configuration of Encode(i-1) is shared by Quantize(i-1) as shown

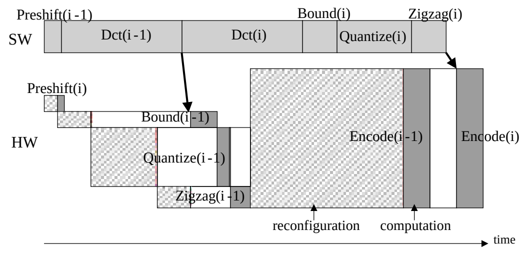


Fig. 7. Result of HW-SW partitioning and scheduling of the JPEG example (A^{HW}=42 Kbytes).

in Table IV). Bold arrows (e.g. the one from the end of Dct(i-1) computation job to the start of Bound(i-1) computation job) represents communication between SW and HW.

VI. Conclusion

In this paper, we have presented a new method for HW-SW cosynthesis with the run-time incrementally reconfigurable FPGA. Experiments show the EPR and IR concepts and the proposed heuristics give significant performance improvement for synthetic examples and real embedded system examples compared to the optimal solutions without the concepts.

Currently, we are working on extending the EPR and IR concepts to such systems that have computation-intensive loops and conditional executions.

References

[1] S. Hauck, "The Future of Reconfigurable Systems", *5th Canadian Conference on Field Programmable Devices*, June 1998.

[2] S. Hauck, "Configuration Prefetch for Single Context Reconfigurable Coprocessors", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74, 1998.

[3] R. P. Dick and N. Jha, "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems", *Proc. Int. Conf. on Computer Aided Design*, Nov. 1998.

[4] B. Jeong, "Hardware-Software Partitioning for Reconfigurable Architectures", *M.S. thesis, School of Elec. Eng., Seoul National Univ.*, Feb. 1999.

[5] F. Vahid, "Modifying Min-Cut for Hardware and Software Functional Partitioning", *Proc. Int. Workshop on Hardware-Software Codesign*, pp. 43–48, Mar. 1997.

[6] Portable Video Research Group, *PVRG-JPEG CODEC*, ftp://havefun.stanford.edu/pub/jpeg/JPEGv1.2.1.tar.Z.

[7] Telenor, *Telenor's H.263 Software*, http://www.nta.no/brukere/DVC/h263_software/.

[8] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task Graphs for Free", *Proc. Int. Workshop on Hardware-Software Codesign*, pp. 97–101, Mar. 1998.

[9] ARM Ltd., "ARM7 Data Sheet", *http://www.arm.com/Documentation/UserMans/PDF/ARM7vC.pdf.*

[10] Atmel Co., "AT40K FPGAs with FreeRAM", *http://www.atmel.com/.*

[11] ARM Ltd., *Software Development Toolkit*, http://www.arm.com/products/SDT/.