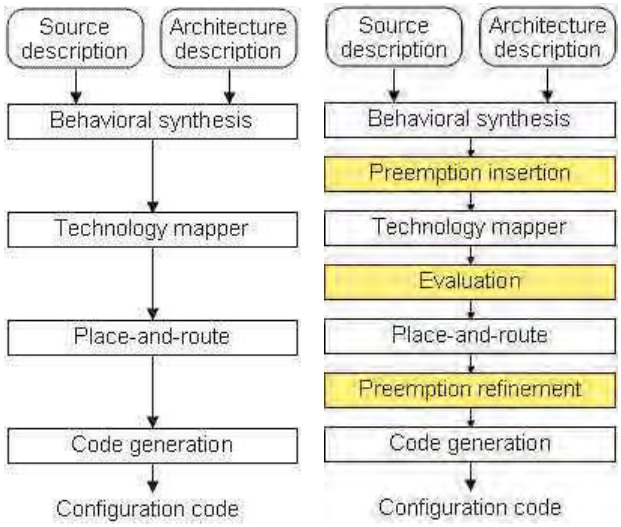


(a) Origin design flow (b) Modified design flow

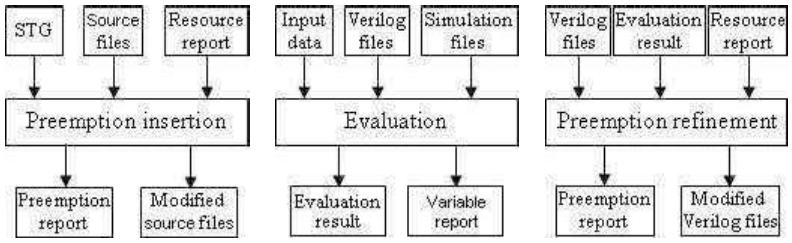
Fig. 5.3: System design flow

At first (Fig. 5.4(a)), the source files, the STG and the resource report of a target application are the basis for identifying possible preemption points. While the resource report is produced by the behavioral synthesis, the STG can be automatically generated by analyzing Verilog source files to identify computation states and their relationship. The generation of an STG could be integrated into the compiler/synthesis when the proposed method is applied in the future. The life time and scope of variables in source programs are analyzed to determine state data necessary to be saved at each computation state according to the STG. Computation states where resource demand is less than a certain threshold are marked as potential preemption points, then extra states for capturing and restoring state data are inserted into source programs at such preemption points.

Next (Fig. 5.4(b)), after the technology mapper step, produced Verilog files, simulation test bench and input data could be provided to an RTL simulator to evaluate how added states at preemption points affect the implementation. It is also possible to simulate and analyze how variables are initialized, used and discarded over each execution clock cycle. Accordingly, a variable report and an evaluation result report may be produced after the evaluation step.

Finally (Fig. 5.4(c)), based on the user-specified latency τ_{input}, the evaluation result and the place-and-route resource report, preemption points generated previously are modified in a way that the maximum preemption latency is satisfied. If the preemption latency cannot be achieved or hardware overhead is too large, another τ_{input} should be given. This process can be repeated until the list of generated preemption points can satisfy the following equation:

$$\tau_{max} \leq \tau_{input}$$



(a) Insertion (b) Evaluation (c) Refinement

Fig. 5.4: Input and output information for proposed method

where: τ_{max} denotes the maximum preemption latency, and τ_{input} specifies a constraint value of maximum preemption latency.

5.4.2 Preemption algorithm

The proposed algorithm achieves the target of minimizing context switch overhead by:

1. allowing preemption only at computation states where used resources are small, and
2. inserting special states for saving and restoring state data. Since including just input/output instructions, so these added states require a small number of resources, and no extra register files as well as memory modules.

The algorithm proposed here is consisting of three stages according to modified steps in Fig. 5.3: inserting preemption states into the original STG (Algorithm 1), evaluating, and refining (Algorithm 2).

5.4.2.1 Inserting preemption states (Algorithm 1)

Using the resource report generated by the behavioral synthesis as an estimation, the insertion algorithm (Algorithm 1) tries to find out potential preemption points.

Variable analysis (lines 2-8): It is necessary to analyze the variables of the target application from the source program in order to find all global and local variables (both register and memory variables) to all computation states. Global and static variables are often saved when the task is preempted unless they are not yet initialized and used. Local variables to a given computation state will only be saved if the associated state becomes a preemption point. In Algorithm 1, P holds a set of preemption points; PS is a set of special states where preemption is allowed, but state data do not need to be saved, as mentioned in the next step; S is a set of computation states, or in other words, it is a set of nodes in the STG of a target application. After variables are analyzed (line 5), each computation state is associated with correspondent variables (line 6), that means if state s_i becomes a preemption point, the correspondent set of variables $v_i(k_i)$ needs to be captured.

input : None
output: List of preemption points P

```
1  begin
2      P ← ∅;
3      PS ← ∅;
4      V ← ∅;
5      L ← ∅;
6      S ← {s_1, s_2, .., s_n};
7      V ← Variable_Analysis(S);
8      (s_1, s_2, ..., s_n) = (v_1(k_1), v_2(k_2), ..., v_n(k_n));
       /* Look for potential preemption points PS where no data is saved */
9      foreach s_i ∈ S do
10         if Initial_State(s_i) ! = 0 then
11             PS ← PS ∪ s_i;
12         end
13     end
       /* Detect computation loops and sort incrementally                 */
14     L ← Loop_Detection();
15     L ← Incremental_Sort(L);
       /* Look for preemption points inside and outside loops             */
16     foreach l_i ∈ L do
17         foreach s_i ∈ l_i do
18             if s_i ∉ PS then
19                 if Required_Resource(s_i) ≤ θ then
20                     P ← P ∪ s_i;
21                 end
22             end
23         end
24     end
25     foreach s_i ∈ (S \ L) do
26         if Required_Resource(s_i) ≤ θ then
27             P ← P ∪ s_i;
28         end
29     end
       /* Insert states for capturing and restoring state data at p_i ∈ P  */
30     P ← P ∪ PS;
31     foreach p_i ∈ P do
32         Insert a new state for capturing state data;
33         Insert a new state for restoring state data;
34         C ← ∅;
35         R ← {r_1, r_2, ..., r_p};
36         while R ≠ ∅ do
37             C ← {r_k, r_{k+1}, ...};
38             R ← R − C;
39         end
40     end
41 end
```

Algorithm 1: Insertion Algorithm

Potential preemption points analysis (lines 9-13): Certain computation states where variables are initialized may become potential preemption points since they can be easily re-executed without saving variables. These states can become preemption points with a special handle: instead of saving variables in states, these states will be simply re-executed when being restores. Such states are often found at the beginning of programs and at the end of loops. According to Algorithm 1, all states $(s_1, s_2, ..., s_n)$ are searched to find potential states, which are kept in PS. These states will become preemption points by being re-executed when the task is resumed.

Loop detection (line 14) At the beginning, Algorithm 1 detects all computation loops $L = l_1, l_2, ..., l_n$ using the given STG. Each loop l_i contains a number of states $l_i = s_{ij}, s_{ij+1}, ...$ The detection of computation loops is important since they are likely to take a considerable amount of time. Taking into account preemption latency, a loop without any preemption points inserted could violate a required preemption latency. For instance, a simple loop for resetting an array variable to a certain value is common in programs. Regardless of being unrolled manually or by the compiler, this kind of loop often contains only one or two states being executed repeatedly for a number of times. If no preemption is allowed in the loop, the given preemption latency might not be satisfied. Fortunately, instead of analyzing a complicated source program, it is more convenient to deal with the STG represented in the form of a flowgraph. Applying a loop detection algorithm [108] [109] on the STG of an implementation, all loops are identified and marked in order that the insertion algorithm will analyze and insert at least one preemption point among states constituting a loop.

Sorting (line 15): Using a suitable sort algorithm, loops l_i are sorted incrementally according to numbered states, i.e. $\forall s_u \in l_i \; and \; s_v \in l_k : \; s_u \leq s_v \; (i < k)$.

Preemption point finding (lines 16-29): Based on the resource report generated by the behavioral synthesis, loops are searched for possible preemption points where used resources are within a given threshold θ (lines 14-22). States that do not belong to any computation loops are also investigated to find out preemption points using the threshold θ (lines 23-27).

New states insertion (lines 30-40): At preemption points, new states are inserted for transferring necessary resources to an outside memory. Since the input/output interface of DRPAs often consists of a certain number of bits, resources are grouped into packets of those bits for output. Depending on the amount of resources and how memories are allocated, it would take a number of clock cycles for transferring. This contributes to reduce the preemption latency and affects the overall performance of the task.

5.4.2.2 Refining preemption points (Algorithm 2)

In order to prepare for refining the list of preemption points generated by 1, it is critical to quantitatively evaluate how preemption states inserted affect the implemented application. This can be done by executing simulations on the original and modified implementations of the application. Using design tools of most DRPAs, the RTL simulation can be performed in order to obtain the critical path, the execution clock cycles, and the used resources at the technology mapper level. With suitable computations, other parameters such as the operating frequency, the throughput and the preemption latency at a given state can be determined.

input : Input preemption latency ψ_{given}
output: None

1 **begin**
2 $S \leftarrow \{s_1, s_2, .., s_n\}$;
3 $k_1 \leftarrow 0$;
4 $M \leftarrow \emptyset$;
5 **for** $k_2 \leftarrow 1$ **to** P **do**
6 **while** $Preemption\_Latency(P_{k2}) > \tau_{input}$ **do**
7 $M = M \cup \{(s_{k_1} < s_i < s_{k_2})$ where required resource is minimized
8 **end**
9 $P \leftarrow P \cup M$;
10 $k_1 \leftarrow k_2$;
11 **end**
12 $B \leftarrow \emptyset$;
13 **for** $i \leftarrow 0$ **to** P **do**
14 **if** $Preemption\_Latency(P_i) \leq \tau_{input}$ **then**
15 $B \leftarrow B \cup P_k$;
16 **else**
17 **if** $n(B) \geq 2$ **then** $P \leftarrow P - B$;
18 $B \leftarrow \emptyset$;
19 **end**
20 **end**
21 **foreach** $p_i \in P$ **do**
22 $R \leftarrow \{r_1, r_2, ..., r_p\}$;
23 **while** $R \neq \emptyset$ **do**
24 $C \leftarrow \{r_k, r_{k+1}, ...\}$;
25 $R \leftarrow R - C$;
26 **end**
27 **end**
28 **end**

Algorithm 2: Refinement Algorithm

Preemption points generated by the previous step could be improved since Algorithm 1 often generates more than necessary. Moreover, the estimation of the critical path at the early stage of the design flow, which is basic for computing the preemption latency, is usually larger than the real one.

Therefore, based on the reports of the place-and-route phase and the evaluation results, redundant preemption points can be eliminated. Different requirements for preemption could become criteria for removing preemption points from the list generated by Algorithm 1. In the simplest case when the preemption latency can be tolerated, the refining algorithm just tries to eliminate preemption states consuming a larger number of clock cycles. In many cases, a user-specified constraint on preemption latency may be given. The following refining algorithm applies an input preemption latency τ_{input} as a criterion for optimize preemption points. In other words, τ_{input} is a constraint value for preemption latency. As a result, Algorithm 2 should generate such a list of preemption points that guarantees following condition.

$$\tau_{max} \leq \tau_{input}$$

where: τ_{max} denotes the maximum preemption latency or the maximum time to switch to a new task once a preemption is requested to the current running task. If such a list of preemption points, which guarantees above condition, could not be generated, the algorithm will report to designers and let them select another τ_{input}. This process can be repeated several times until a given τ_{input} is satisfied.

Preemption point scanning (lines 2-12): First of all, the algorithm scans the list of preemption points P generated at the previous stage to find out if the condition $\forall i : \tau_i \leq \tau_{input}$ $(i = 0, ..., n)$ is satisfied. If not, extra preemption points are inserted at states where required resources are small.

Unnecessary preemption point elimination (lines 13-21): The algorithm tries to remove unnecessary preemption points using the given preemption latency τ_{input}. Any preemption points in the list P between two preemption points t_1 and t_2 that satisfy $\forall p_i \in P : \tau_{pi} \leq \tau_{input}$, will be eliminated.

Preemption state modification (lines 20-26): Next, preemption states for saving and restoring context data are inserted or modified based on the accurate resource report generated by the place-and-route tool.

Management information: Necessary information should be defined for correctly managing data when saving and restoring, and for the outside operating system to control and schedule tasks. Therefore, while looking for preemption points and inserting preemption states, a data structure containing information such as the amount and the order of data saved and restored must be defined. By modifying the design at high levels (C-based and Verilog files), there is no need to know exactly very detailed information about the hardware architecture like in which PE a specific register variable is assigned to, or which memory modules hold a memory array variable.

5.4.3 Illustrative example

The example code in Fig. 5.6 is implemented on the NEC‘s DRP architecture to show how the proposed method works with the STGs of Task 1 and Task 2 (Fig. 5.5). The algorithm traverses

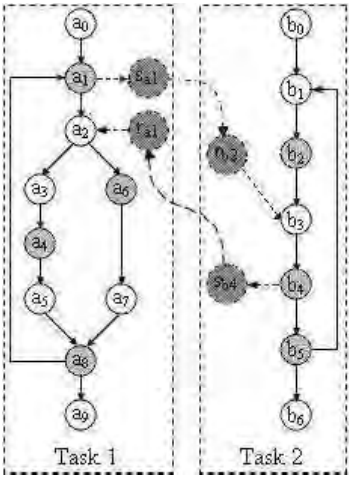


Fig. 5.5: Proposed solution

Saving states	Restoring states
reg(0:16) r[4]; mem(0:16) m0[8], m1[8]; mem(0:16) m2[8], m3[8]; ... dout = r[3]::r[2]::r[1]::r[0]; DataOut(dout); m3[0], m2[0], m1[0], m0[0]; $ dout = m3[0]::m2[0]::m1[0]::m0[0]; DataOut(dout); m3[1], m2[1], m1[1], m0[1]; $ ...	reg(0:16) r[4]; mem(0:16) m0[8], m1[8]; mem(0:16) m2[8], m3[8]; ... din = DataIn(); r[3]::r[2]::r[1]::r[0] = din; $ din = DataIn(); m3[0]::m2[0]::m1[0]::m0[0] = din; $ ...

Fig. 5.6: Example code

STGs to find out the states where used resources are smaller than a certain limit; for example, states (a_1, a_4, a_6, a_8) of Task 1 and (b_2, b_4, b_5) of Task 2 with gray circles are such states. These states are marked as preemption points, and preemption states for saving (states s_{ai}, s_{bj}) and restoring (states r_{ai}, r_{bj}) data are inserted, and correspondent STGs are modified. These inserted states are assumed to be executed only when their coupled states are preempted. For example, when Task 1 is executing at state a_0, a preemption request occurs. Since a_0 is not a preemption point, Task 1 is not interrupted but continues to run until a_1, a nearest preemption point. Task 1 is stopped after a_1, and the execution is transferred to the correspondent state for saving the state information of a_1 (s_{a1}). After r_{b2}, which restores the state data of b_2, is executed, the execution is moved to Task 2, and Task 2 starts to run from b_3 (assume that Task 2 was preempted before at b_2). Switching from Task 2 back to Task 1 is handled in the similar way, for instance, when Task 2 is preempted at b_4.

The example code also illustrates how state data can be saved and restored by states s_k and r_k. The example is described in NEC's BDL [59], a C-based language. r, l_i $(i = 0,\ 1,\ 2,\ 3)$ are arrays of *register* and *memory* types with 16-bit width. Symbol :: shows a concatenation operator, which links variables together to form a larger bit width result. For example, the statement *r[3]::r[2]::r[1]::r[0]* concatenates four 16-bit elements of array r to create a 64-bit output. *DataOut* and *DataIn* are output and input functions working with 64-bit output/input interfaces via 64-bit variables *dout* and *din*. Symbol $ presents a timing descriptor to manually divide the codes into different states.

In the example, when all register and memory variables r, l_i $(i = 0,\ 1,\ 2,\ 3)$ need to be saved, it takes 17 clock cycles to transfer outside. However, the outside controller does not need to know about where saved data come from (registers or memories), and if saved values are 8, 16 or 32 bits. What the controller has to do is to allocate 64-bit buffers to hold data, to maintain the order of saving data, and to send in exactly the same amount in the type of 64-bit packets and order of data when data are restored.

The example shows that by modifying source programs at high level to insert preemption states,

it is possible to avoid the details of the underlying hardware architecture like the exact place of a register variable.

5.5 Target Device

Although the preemption algorithms proposed here can be extended to apply on other reconfigurable devices, in this research, I focus on DRP-1 from NEC Electronics as the target model. This device and the architecture of DRP-1 have been mentioned in Section 3.2.3. As mentioned before, applications designed for DRP platform can be described in the C-based hardware description language called BDL. Although BDL supports pointers, dynamically memory allocation is not allowed. All memory and register assignment are done at the compile time. Also, state registers allowing the DRP to transition from one state to another are determined at that time. The input/output interface of the DRP-1 is performed via two 64-bit separated channels. One input and one output operation can be executed concurrently in a clock cycle.

Currently, the DRP has no multitasking capability. At one time, only one application can be configured and executed on the whole 8-tile reconfigurable array. The basic operation model on the DRP is *time-division execution*, where an application is divided into multiple contexts, and one context at a time is activated and executed. The other operation model the DRP supports is *multi-process execution*, where an application is divided into several processes, each of which could be mapped into a group of Tiles and executed in parallel. Although the multi-process execution allows several threads of control (processes) to be present at the same time and run concurrently, it is not a true multitasking execution since processes belong to one application and no preemption is allowed.

5.6 Evaluation

5.6.1 Target applications

The proposed algorithm in this paper was evaluated on a number of real applications shown in Table 5.1. For each application, different cases are implemented and shown in column $Version$. Cases include implementations when no preemption points are inserted A_0; when preemption points are inserted without any constraint on preemption latency A_1 (only Algorithm 1 (Section 5.4.2.1) is applied); when the latency is specified $A_{\tau 1}$ and $A_{\tau 2}$ (both Algorithm 1 (Section 5.4.2.1) and Algorithm 2 (Section 5.4.2.2) with different input preemption latencies are applied); and when preemption is allowed at every state (A_n). (A represents the name of target applications). Columns 4 and 5 show the number of nodes and edges in the STG of each application. Columns 6-9 denote the input preemption latency (τ_{input}) (input parameter for Algorithm 2, Section 5.4.2.2), the critical path ($Delay$), the maximum preemption latency of each implementation (τ_{max}), and the total number of required PEs ($Used\ PEs$) respectively.

In Table 5.1, *Used PEs* shows the total number of required PEs in every state. τ_{input} specifies a constraint value of maximum preemption latency. If the proposed algorithm cannot generate a list of preemption points satisfying a certain τ_{input}, another value of τ should be given. It is repeated until the given τ_{input} is satisfied. τ_{max} specifies the longest response capability to switch to a new task from a currently running task. In the case of A_0, τ_{max} can be considered to be equal to execution time since when an application cannot be interrupted while running, switching to another application may only be possible when the application terminates.

Taking into account the delay from a moment a preemption request arriving to the moment the execution reaching the nearest preemption point, the calculation of preemption latency according to Fig. 5.1 and equation 5.1 becomes as follows:

$\tau = T_p + T_s + T_r$

where: T_p, T_s and T_r are time to reach the closest preemption point from the moment a preemption request is issued, time to save the state data of the preempted task, and time to restore the previously captured data of the preempting task respectively. T_p, T_s and T_r correspond to t_2, t_3 and t_4 in equation 5.1.

In a multitasking environment, the calculation of preemption latency depends on the combination of applications, and the combination of saving/restoring states of preempted and preempting applications. The former is difficult to determine and depends on specific scenarios; and, the latter causes preemption latency to vary even if there are only two applications executing in a system. In this paper, the calculation of preemption latency is performed at every state on the STG of a single application with the assumption that the same set of resources is applied for both saving and restoring. Though not being the exact situation in a real system, this gives us a relative overview on how preemption latency may vary.

All implementations do not pack multiple states into a single context (in the current DRP-1, maximum four states can be assigned to a context) in order to see the impact of inserted states on the performance. Therefore, the number of states is also the number of contexts after synthesizing. Although this prevents implementations with more than 16 contexts from executing on the real chip, it is still possible to complete the place-and-route phase, to execute simulations and to achieve suitable reports. The delay shown in Table 5.1 is the critical path of implementations. Since the option to pack multiple states into a context is not used, added states containing only input/output instructions for capturing and restoring state data do not have any influence on the critical path mainly formed by other main computation states.

5.6.2 Hardware overhead

Hardware overhead or context switch overhead H specifies the amount of additional resources required by added preemption states. If H_{PE} and H^*_{PE} denote the number of required PEs in the original and modified implementations respectively, H can be represented by: $H = H^*_{PE} - H_{PE}$.

Table 5.1: Target applications and evaluation results

Applications	Abbr.	Version	Number of nodes	Number of edges	τ_{input} [ns]	Delay [ns]	Max. latency τ_{max}[ns]	Used PEs
Discrete Cosine Transform [20]	DCT	DCT_0	28	32	-	67.3	6393.5	1105
		DCT_1	-	-	-	-	431.2	1197
		$DCT_{\tau 1}$	-	-	250	-	247.3	1274
		$DCT_{\tau 2}$	-	-	300	-	296.6	1218
		DCT_n	-	-	-	-	180.0	1459
Inverse Modified DCT [20]	IMDCT	$IMDCT_0$	40	48	-	129.5	174695.5	2582
		$IMDCT_1$	-	-	-	-	827.5	2854
		$IMDCT_{\tau 1}$	-	-	750	-	748.5	2820
		$IMDCT_{\tau 2}$	-	-	820	-	815.0	2943
		$IMDCT_n$	-	-	-	-	810.0	3714
Viterbi decoder [20]	Viterbi	$Viterbi_0$	10	10	-	30.4	334.4	843
		$Viterbi_1$	-	-	-	-	273.6	903
		$Viterbi_{\tau 1}$	-	-	200	-	194.8	959
		$Viterbi_{\tau 2}$	-	-	230	-	217.2	938
		$Viterbi_n$	-	-	-	-	828.0	1431
JPEG encoder [94]	JPEG	$JPEG_0$	62	76	-	54.7	34689.2	1851
		$JPEG_1$	-	-	-	-	436.9	2078
		$JPEG_{\tau 1}$	-	-	300	-	289.4	2113
		$JPEG_{\tau 2}$	-	-	350	-	344.1	2052
		$JPEG_n$	-	-	-	-	378.0	2742
Turbo encoder [95]	Turbo	$Turbo_0$	35	41	-	77.8	1249857.0	3008
		$Turbo_1$	-	-	-	-	712.4	3169
		$Turbo_{\tau 1}$	-	-	400	-	389.6	3387
		$Turbo_{\tau 2}$	-	-	450	-	413.4	3287
		$Turbo_n$	-	-	-	-	324.0	4232
MPEG-2 decoder [96]	MPEG	$MPEG_0$	89	101	-	67.9	106195.6	2787
		$MPEG_1$	-	-	-	-	751.0	3200
		$MPEG_{\tau 1}$	-	-	380	-	351.8	3311
		$MPEG_{\tau 2}$	-	-	450	-	419.7	3141
		$MPEG_n$	-	-	-	-	288.0	4445
G721 encoder	G721	GSM_0	12	17	-	93.6	50169.6	1085
		GSM_1	-	-	-	-	392.4	1126
		$GSM_{\tau 1}$	-	-	250	-	237.6	1206
		$GSM_{\tau 2}$	-	-	300	-	295.2	1150
		GSM_n	-	-	-	-	252.0	1376

Although containing only input/output instructions, additional states inserted into the STG of an application for capturing state data still require a number of PEs for concatenating data into *n-bit* packets. This causes a certain hardware overhead. Column *Used PEs* in Table 5.1 shows the required hardware resource in term of PEs for my implementations. Using implementations without preemption points (A_0) as the basic, Fig. 5.7 presents how the hardware overhead varies when preemption points are inserted. In Fig. 5.7, symbols A_1, $A_{\tau 1}$, $A_{\tau 2}$ and A_n denote implementations corresponding to Table 5.1 where A is the name of applications. The hardware overhead varies from 4% to 15% for the A_1 case, from 11% to 15% for the $A_{\tau 1}$ case, from 6% to 13% for the $A_{\tau 2}$ case, and from 27% to

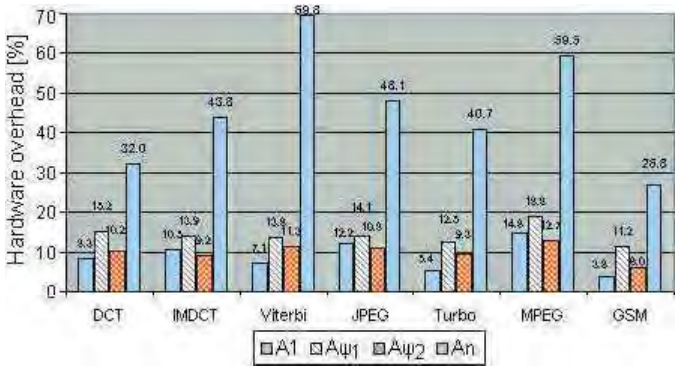


Fig. 5.7: Hardware overhead

70% for the A_n case.

The hardware overhead of implementations according to the proposed algorithm is not large. More importantly, they are even smaller than the A_1 implementation in some cases ($A_{\tau 2}$ vs. A_1 for IMDCT, JPEG and MPEG). This results from the optimization performed by the refining algorithm to eliminate redundant preemption points using a given preemption latency as a criteria. Although some additional preemption points need to be inserted in order to satisfy the given preemption latency, other unnecessary preemption points could be removed. As a result, the hardware overhead could be reduced.

5.6.3 Preemption latency

Preemption latency τ can be defined as the time from a preemption request until a preempting task is ready to run, and it can be computed according to equation 5.6.1.

Fig. 5.8 shows the maximum preemption latency for each implementation. Basically, τ_{input} is used as a constraint for optimizing generated preemption points in Algorithm 2 (Section 5.4.2.2). Such implementations ($A_{\tau 1}$ and $A_{\tau 2}$) have better preemption latency over correspondent versions without such constraints (A_0). In some implementations (IMDCT, Viterbi, JPEC and GSM) the preemption latency of the A_τ versions is even smaller than that of the correspondent A_n although the latter has no delay for reaching a preemption point (A_n versions can be preempted at any state). This means in those cases, time to save and restore state data at some points dominates the total preemption latency.

5.6.4 Hardware overhead vs. preemption latency

At the first sight, hardware overhead seems to be reduced when the number of preemption points is eliminated; or, in other words, preemption latency is increased. In order to see the trade-off between these two parameters, different preemption latencies are provided as the input parameter to the refinement algorithm (Algorithm 2, Section 5.4.2.2) and results are presented on Fig. 5.6.4, where preemption latency is on X-axis and hardware overhead in the number of PEs is on Y-axis.

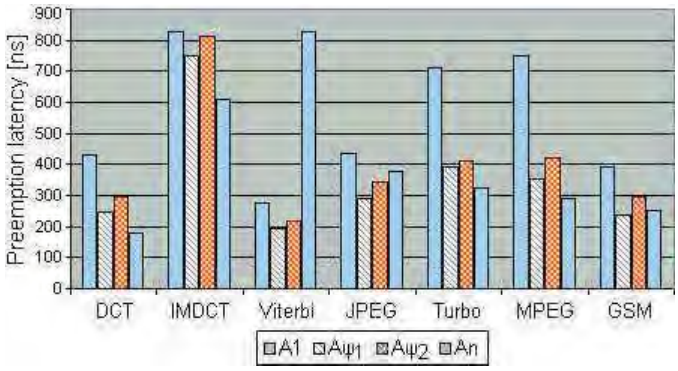


Fig. 5.8: Maximum preemption latency

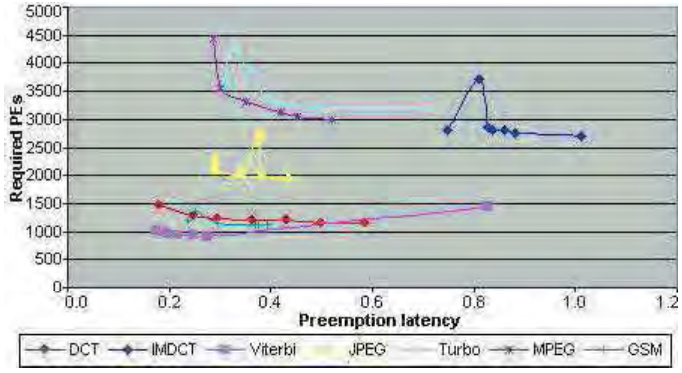


Fig. 5.9: Hardware overhead vs. Preemption latency

When preemption latency becomes larger, hardware overhead tends to reduced. Nonetheless, some implementations show more complicated relationship when preemption latency increases, hardware overhead also increases at some points. Looking into more details, these points correspond to the situation where preemption is allowed at every state (A_n version). In this case, both preemption latency and hardware overhead are influenced by the amount of state data necessary to save. When this amount is large, both these parameters also grow. Therefore, it may not be a good solution to allow preemption at every state, and my proposed method achieves its merit, which satisfies a constraint on preemption latency with reasonable hardware overhead.

5.7 Conclusion

A method for identifying preemption points and inserting extra states to capture and restore state data of applications implemented on coarse-grained dynamically reconfigurable devices based on resource requirements is proposed to enable a preemptive multitasking environment where a running task can be preempted. Evaluation results on the DRP architecture show that the proposed method may satisfy a user-specified preemption latency within a reasonable amount of hardware overhead. Moreover, the steps of the proposed method are integrated into the system design flow to assist

designers in developing applications on dynamically reconfigurable devices. Also, the trade-off between preemption latency and hardware overhead is also presented and discussed.

Chapter 6

Multicore Reconfigurable Architecture

6.1 Problem

Recently, multicore processors have emerged as a dominant trend in the chip making industry due to their advantages over the single-core architecture [15] [110] including: (1) to overcome the limitation of single-core architecture relating to CMOS process, wiring delay and power consumption; (2) to improve performance based on thread-level parallelism in addition to ILP, which has been exploited for improving the performance of the single-core architecture for the last two decades; (3) and, to bridge the gap of memory wall. In embedded devices, in which low power coupled with high performance is quite critical, a trend to shift to multicore can be observed as well [111].

Recently, dynamically reconfigurable accelerators [14] that use an array of simple coarse-grained PEs, have been popularly used, and some of them are embedded in SoCs toward electronic appliances [112], [113] and [114]. A PE is often composed of a simple shift and mask unit, an arithmetic and logic unit, and a register file. Each PE has a memory component to hold multiple circuit configuration data referred as contexts, each of which is constructed by operational instructions for the PE and connection instructions for routing resources. By rapidly switching contexts, the PE operations and intra/inter-PE connections can be dynamically reconfigured every one or few clock cycles.

For efficient use of such accelerators, since a single hardware task often lacks enough degree of parallelism, executing multiple tasks is advantageous. In Chapter 4, a task-level parallelism mechanism has been investigated by mapping the tasks of an application into a reconfigurable array that is partitioned into clusters of hardware execution units (tiles or tile groups). However, because of the limitation of the number of tiles and tile groups in a currently available DRPA, only a limited number of tasks can be implemented. Big applications composed of many tasks, or tasks whose sizes are large, or multiple applications each of which consists of a number of tasks are difficult and almost impossible to be implemented. Furthermore, it is hard to enlarge the reconfigurable array of a chip because of matters such as the size of the chip, power consumption, clock control and chip complexity. To further exploit a task-level parallelism, toward an efficient multitasking environment on DRPAs, it is necessary to investigate a reconfigurable multicore structure.

In Chapter 5, a mechanism for preempting a task executing on a DRPA has been proposed and examined. However, evaluation results obtained come from a simulation environment that allows to execute one task at a time. As being mentioned in such a situation, preemption latency is not the one in real systems but the worst case. In order to provide an environment where multiple applications can be assigned and executed in parallel, a multicore architecture is a good platform to experience and to further extend the proposed preemption algorithm.

In conventional FPGAs, multi-task systems with their partial reconfigurability have been proposed. The architecture of FPGAs allows to divide the reconfigurable array into one-dimensional (column oriented) slots, each of which can hold and execute a task; more importantly, that helps the partial reconfiguration mechanism be implemented [115] [116]. Using partial reconfiguration capability on such FPGAs, while many tasks are executing, it is possible to stop a certain task, free the slot occupied by the task, load a new task into this slot, and execute the new task [115] [117] [118]. Slots may either be equal in size, as used in [119,103], or have variable sizes, as proposed in [120,121,122].

Another example comes from the architecture of NEC Electronics' DRP-1, in which the reconfigurable array is divided into eight tiles, which can work independently and communicate with each other [63]. Multiple hardware tasks can be implemented on one or several numbers of tiles, and executed in parallel or pipelined manner.

Using a Network-on-Chip (NoC) for inter-task communications in reconfigurable platforms has been proposed. In an NoC, source nodes generate packets that consist of a header and a payload data, which are transferred by on-chip routers through connected links, and decomposed by destination nodes. Since different packets can be simultaneously transferred on multiple links, the bandwidth of NoCs is much larger than that of buses. In addition, the wire-delay problem is resolved, since each flit of a packet is transferred on limited length point-to-point links, and buffered in every router along the routing path. By introducing error detection and re-transmission protocols, dynamic transmission errors caused by crosstalk, which will come up in future smaller process technologies, can be also solved [123].

In this work, I propose a mutlicore reconfigurable architecture in which each core can be assigned a task for execution. Many multiprocessor systems equip an on-chip bus for inter-core communication because of the relatively small number of cores. Nonetheless, in the proposed architecture, a regular NoC is introduced in order to ensure each core scalability and modularity. Different from the tile-based architecture introduced in Chapter 4 in which tiles are directly connected and hardware tasks assigned to tiles for execution can communicate to one another using embedded memory modules arranged as FIFOs, the proposed multicore architecture is equipped an NoC to connect computational cores. These two architectures have not been well evaluated and compared with real application programs. In this section, I build correspondent architectures based on NEC's DRP-1 and implement applications on each architecture in order to see how these architectures affect performance and resource usage.

6.2 Related Work

Techniques on mapping multiple tasks into a single FPGA has been widely researched. The architecture of many FPGAs permits to execute multiple tasks based on the one-dimensional partitioning model where an FPGA's reconfigurable area is partitioned into the stripes of full array height or slots. Slots are placed side by side and may be equal in size [103] [119] or have variable widths [121] [120]. By exploiting the partial reconfiguration capability, slots can be individually reconfigured. In these systems, however, tasks communicate by means of programmable routing resources of the FPGA. Concerning resource management, the one-dimensional partitioning model is equivalent to a homogeneous multi-processor model.

The reconfigurable surface can also be divided into predefined slots, which are allocated to tasks for execution according to a scheduling algorithm. Due to the fragmentation problem, many solutions have been proposed. [21] tries to rearrange tasks executing on a partially reconfigurable FPGA by local repacking and ordered compaction. Exploiting the technique of task relocations and transforms to reduce fragmentation is discussed in [40]. [121] investigates different placement techniques for coarse-grained, non-rectangular tasks on partially reconfigurable FPGAs.

Unlike FPGAs, a DRPA supports a high speed dynamic reconfiguration mechanism during execution. In order to map multiple tasks into such devices, special architectural support is required. NEC's DRP-1 [63] is consisting of eight tiles each of which can control dynamic reconfiguration independently. Several tasks can be assigned into one or a number of tiles, and executed in parallel communicating with each other. Hitach's FE-GA [67], SANYO's car turner DRPA [114], and Toshiba's SAKE [113] uses a multi-core structure consisting of relatively small DRPAs for special mapping of multiple tasks. Such types of DRPAs are designed based on an observation that a group of relatively small sized arrays is efficient in various applications [82].

Using an NoC in a reconfigurable fabric has been proposed by [28] in order to improve the interconnection network in terms of structure, performance and modularity. [118] [124] [106] [125] have proposed an FPGA-based multi-task system in which an NoC is used for inter-task communications. In [118], a two-dimensional torus is introduced as a network topology. Each router for inter-task communications has two virtual channels, and provides wormhole packet switching and dimension-order routing, which is one of the deterministic routing algorithms. This system has been implemented on the Xilinx Virtex-II FPGA (XC2V6000).

6.3 Evaluated Architectures

6.3.1 Target Device

In order to evaluate and compare both architectures, DRP architecture from NEC Electronics is selected as the target device for this study. The detail of DRP can be referred from Section 3.2.3.

In this research, a *tile* of DRP is used as a basic unit to compute the size of computational cores.

Therefore, one DRP tile equivalent to 64 PEs is the smallest size a core could be. To describe different core sizes, different number of tiles arranged in a certain shape is used. Generally speaking, the size of a core is expressed as follows:

$$S_{core} = n \times \text{PEs/tile} \tag{6.1}$$

$$1 \leq n \leq 8 \tag{6.2}$$

where: S_{core} denotes the size of a core. n shows the number of tiles, and $PEs/tile$ is the number of PEs in a DRP tile. Since there are eight tiles in a DRP-1 chip, so the maximum core size is eight tiles. Fig. 6.6(a) - Fig. 6.6(d) show examples for the proposed architecture with different core sizes from one to four tiles. In the multi-process execution, there is a way to specify how tiles are joined together to form a certain shape, which is assigned to a task. I adopt this method to correctly specify the size of cores in implemented variants for a target application as show on Fig. 6.6. A target application is partitioned into tasks, whose sizes are small enough to be able to fit into cores. As a result, the smaller the core size is, the more tasks an application should be divided into.

6.3.2 Tile-based architecture

The tile-based architecture studied in this section is assumed to be a two-dimensional (2D) multicontext coarse-grained DRPA consisting of $M \times N$ hardware execution units, each of which is called a tile. A tile consists of a certain number of PEs and may also have other components such as memory modules, multipliers, register files and flip-flops. The size of a tile is computed as the number of PEs it contains. Generally speaking, tiles could have different sizes, but in order to simplify the mapping process, which is not the object of this study, in the thesis, tiles are supposed to be identical, or have equal sizes. This assumption creates a homogeneous architecture, which allows more flexible in mapping.

A hardware tasks can be mapped to a tile for execution providing that the size of the task is equal or smaller than that of a tile. Furthermore, several neighboring tiles can be joined together to form a tile group, where a task whose size is larger that the size of a tile can be mapped into. Embedded memory modules within the reconfigurable array are used to form a communication mechanism between tasks assigned to two tiles, a tile and a tile group, or two tile groups. When two tasks implemented on two different tiles want to exchange data, they can declare a shared memory module arranged as a FIFO to use as an inter-task communication method. FIFOs use a simple handshake mechanism in order for two tasks involving in communication to determine if an FIFO is full or empty. If there is no data in the input FIFO, or the output FIFO is full, the execution of receiving and sending tasks, respectively, is stalled.

Fig. 6.1 shows an example of a tile-based architecture using the target device of NEC's DRP-1 with 4×2 tiles. Tile groups can be formed by combining close tiles together; so, they may have different shapes. TG_1 and TG_2 are created by the different number of tiles with different shapes.

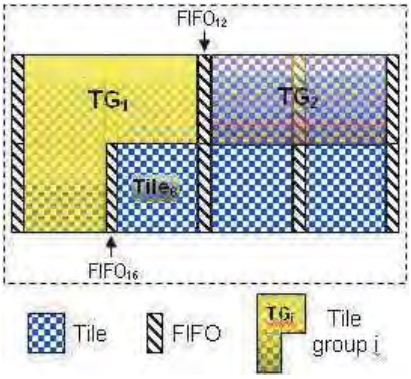


Fig. 6.1: Tile-based architecture

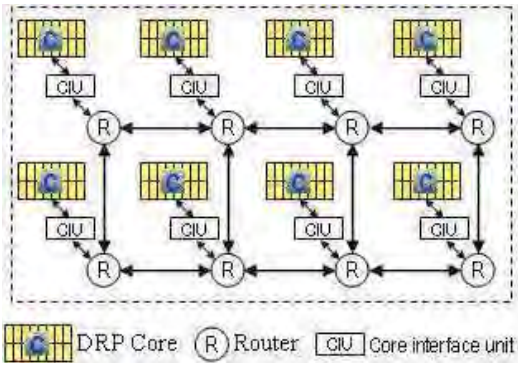


Fig. 6.2: Multicore architecture

$FIFO_{12}$ can be used to exchange data between tasks mapped to TG_1 and TG_2; and, $FIFO_{16}$ is for communicating between TG_1 and $Tile_6$;

For DRP-1, applications are manually partitioned and assigned to tiles or tile groups. Since having eight tiles, the maximum number of tasks that can be concurrently executed is eight. However, in most cases, the number of tasks a target application is partitioned into is less than eight since tile groups are often formed to accommodate tasks that are larger than the size of a tile. A FIFO mechanism, which employs VMEMs between tiles, is used as a inter-task communication method. A FIFO is for one-way communication and acts like a pipe. Writing to and reading from a FIFO are blocking, that is, a task needs to be stalled because of the data shortage.

6.3.3 Multicore architecture

NEC's DRP-1 is used as computational cores for the multicore architecture proposed in this section. Cores are connected by an NoC composing of routers. In order to compare with the tile-based architecture, only 4×2 cores with a two-dimensional mesh topology are used in the study as shown on Fig. 6.2.

The network uses the wormhole switching technique with dimension-order routing. A wormhole routing allows data packets to be pipelined through the network and requires only a small buffer in a router to store a part of a packet (flits). Dimension-order routing, which uses Y-dimension channels after X-dimension channels in a 2D mesh and torus, can be implemented with simple combination logic on a router and does not demand to store routing information in a routing table. In the network, a data packet is broken into flits, which belong to one of three types: header flits, body flits and tail flits. In order to avoid deadlocks, virtual channels are employed.

Fig. 6.3 shows the router architecture used in this study. A router consists of a crossbar switch, an arbitration unit (ARB), input and output physical channels. Each physical channel has two virtual channels, each of which has a FIFO buffer for storing four flits. The router architecture is fully pipelined, and allows to transfer a header flit through three pipeline stages that include routing computation, virtual channel and switch allocation, and switch traversal.

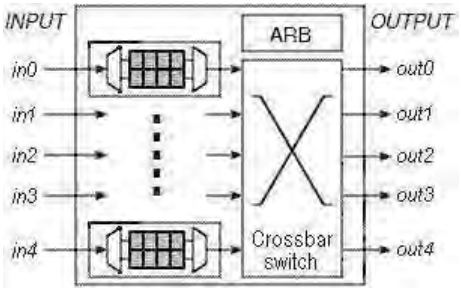


Fig. 6.3: Router architecture

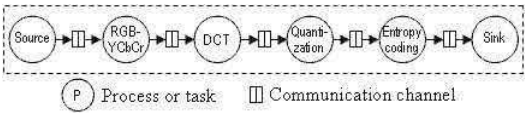


Fig. 6.4: Representation of JPEG encoder

Table 6.1 shows the implementation of a router. A router is synthesized, placed, and routed with a 90nm standard cell library.

Table 6.1: Router implementation

Parameters	Value
Process	ASPLA 90nm
Operating frequency	Maximum up to 500 MHz
Flit size	64/128 bits
Number of ports	5
Number of virtual channels	2
Buffer size	4-flit for each virtual channel
Packet length	4-flit data + 1-flit header

In order to connect cores to routers, a fixed interface, which is referred as a core interface unit (CIU), is used (Fig. 6.2). In this study, the input/output interface of DRP-1, which consist of two 64-bit separated channels, one for input and another for output, is exploited as a CIU. CIUs serve two purposes. First, a CIU can convert data exchanging between a core and a router. The channel width of the network W can be expressed as: $W(bit)$ = flit size + 2 , where *flit size* is the size of a flit or the width of a physical channel, and *2* represents two bits for flit header information. For example, since the input/output interface of DRP-1 with 64 bits is employed to connect to a router, the actual channel bit width between routers is 66 bits. A CIU is used to convert from core's 64-bit channels to router's 66-bit channels. When data are transferred from a DRP core to a router, the correspondent CIU adds two bits containing a flit type into 64 bits; on the way back, when data are sent to a DRP from a router, those two bits of a flit type are removed.

Second, another important reason for using CIUs is to allow tasks to be independent from physical locations. In a multitasking environment where tasks are dynamically assigned into and removed from cores, the exact place of a certain task can only be determined at the run time by the operating system, and it is often changed from time to time either when the system is defragmented or when the task is removed and later resumed. In a general case, for an application made of multiple tasks implemented on a such environment, the physical place of tasks are not known at the design time.

This raises the question of how tasks can communicate to one another if their places are not determined in advanced. My solution is to use CIUs. At the run time, when a task is assigned into a core, the operating system also updates the routing table of the correspondent CIU to clearly show where other related tasks are placed. When task A needs to communicate to task B, for example, the routing information of task B will be added into header flit coming out from task A. Therefore, by using CIUs, it is possible for tasks to communicate independently from their physical places.

6.3.4 Application model

A target application is represented with multiple tasks that can be arranged to execute in a pipelined manner. That is, computations can be specified as a data flow graph with streams of data items (edges) flowing between computation stages (nodes) (Fig. 2.2.1). Basically, a task gets a data stream from its input communication channel, executes its own computation, and produces the correspondent result to the output communication channel. Fig. 6.4 shows an example of a representation for a JPEG encoder.

With the tile-based architecture, each task of a target application is mapped onto a tile or a tile groups depending on their size. With the multicore architecture, tasks are mapped onto cores. In both architectures, all tasks belong to a target application are arranged for executing in a pipelined manner.

6.4 Evaluation

6.4.1 Simulation environment

Fig. 6.5 shows the simulation environment used in this study. A target application is partitioned into multiple tasks *(Task 1, Task 2, ..., Task i, ..., Task n)*, which are described in BDL. Configuration data for DRP-1 generated by the compiler/synthesizer can be loaded onto a DRP-1 chip for execution or for online debug. Apart from configuration data, the compiler also generates Verilog description files corresponding to BDL source files. These Verilog descriptions with Verilog files implementing router architecture, network architecture, core interface and test bench become the input for an RTL simulation. With relevant input data, the RTL simulator can run a simulation and produce correspondent results and reports such as critical path, resource usage, execution clock cycle for evaluation.

6.4.2 Two architectures

6.4.2.1 Target applications

Several real applications shown in Table 6.2 have been implemented in order to compare between the two mentioned architectures shown in Fig. 6.1 and Fig. 6.2.

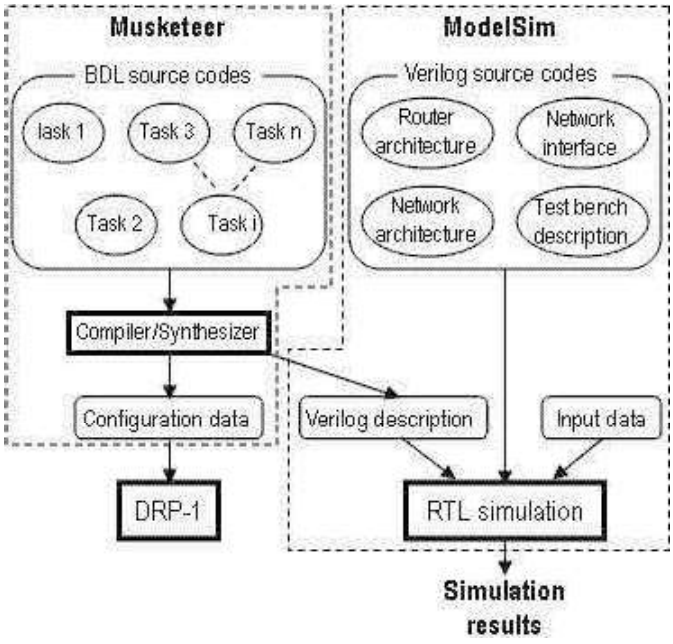


Fig. 6.5: Simulation environment

Target applications are manually partitioned into tasks in such a way that suits specific implementations. In the tile-based architecture, applications are modeled, partitioned into tasks and mapped onto the reconfigurable array of DRP-1 according to the optimized algorithm presented in [16]. Since data are exchanged using FIFOs, the communication channel bit width can be varied from 8 to the maximum of 64 bits. The same configuration is applied to the multicore architecture in order to evaluate the effect of using an NoC for inter-task communication instead of FIFOs. More importantly, the channel bit width of the NoC in the multicore architecture is evaluated with 64 bits and 128 bits in order to see how the width of communication channels influences on performance.

6.4.2.2 Results

Table 6.2 shows the throughput of target applications implemented in two mentioned architectures. *Tile-based* column specifies the result obtained with the tile-based architecture. *Multicore (64 bits)* and *Multicore (128 bits)* present implemented results in the multicore architecture with the channel bit widths of 64 bits and 128 bits respectively.

It is easy to see that the bit width of communication channels between tasks plays an important role in improving throughput. While the implementation in the multicore architecture with the 64-bit channel width, which is the same as the communication channel width of the tile-based architecture, results in lower throughput, the same configuration with the 128-bit channel width achieves better throughput in many cases except IMDCT and MPEG implementations. This shows that communication significantly affects throughput. While it takes only one clock cycle to transfer a data packet ($1bit \leq$ size of a data packet $\leq 64bits$) from one task to another in the tile-based architecture using a

Table 6.2: Throughput of two evaluated architecture

Application	Abbr.	Tile-based [Mbps]	Multicore (64 bits) [Mbps]	Multicore (128 bits) [Mbps]
Discrete Cosine Transform	DCT	341.9	297.1	345.0
Inverse Modified DCT	IMDCT	182.3	156.2	178.8
Viterbi decoder	Viterbi	4.32	3.9	4.34
JPEG encoder	JPEG	106.9	89.2	107.3
Turbo encoder	Turbo	4.0	3.4	4.1
G712 encoder	G721en	45.1	33.1	46.0
G712 decoder	G721de	10.3	9.2	10.5
MPEG-2 decoder	MPEG	10.4	8.7	9.7

FIFO, several clock cycles are needed to transfer a packet from one router to another in the multicore architecture depending on the distance between two routers and the amount of data. The further a target router is, the more number of clock cycle is required for communication. Moreover, while it takes at least four clock cycles to transfer one data packet, which corresponds to a flit, between two adjoining routers, time for communication will reduce if the amount of transferring data is increased since it takes a fixed number of clock cycles to set up a communication path between two routers in an NoC.

6.4.3 Evaluation with different core sizes

6.4.3.1 Implementation variants

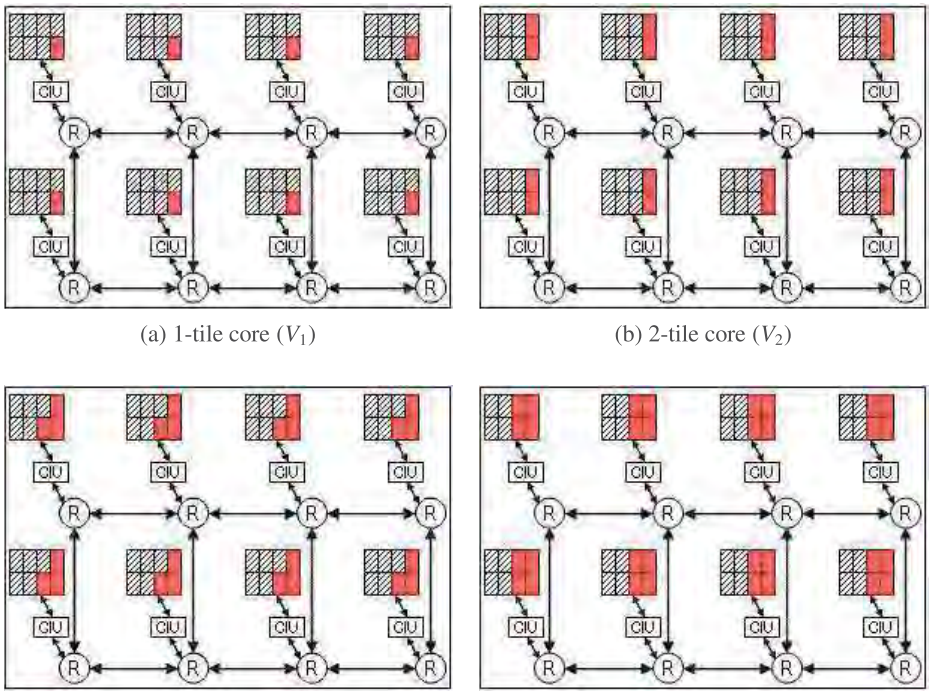
In order to see how the core size influences on the performance of target applications, this evaluation uses the same multicore architecture shown in Fig. 6.2 with variable core sizes. In Fig. 6.2, each core is an 8-tile DRP-1, but Fig. 6.6 show the same architecture with each core has the size of one (Fig. 6.6(a)) to four tiles (Fig. 6.6(d)). For example, Fig. 6.6(b) shows an architecture where each core has the size that is equal to two DRP tiles, or in other words, each core can be considered to contain 128 PEs. Fig. 6.6(a), Fig. 6.6(c) and Fig. 6.6(d) illustrate similar architectures with the core sizes of one, three and four DRP tiles, respectively.

Since the purpose of this evaluation is not to get the maximum performance, but to show the effect of the core size on performance, only 64-bit communication channels are applied for the interconnection network.

6.4.3.2 Application partitioning and mapping

Following rules are adopted when partitioning and mapping target applications onto evaluated architectures.

- Since all evaluated variants only use eight cores (Fig. 6.6(a)-Fig. 6.6(d)), the number of tasks each application is partitioned into must be equal or less than eight.



(a) 1-tile core (V_1)

(b) 2-tile core (V_2)

(c) 3-tile core (V_3)

(d) 4-tile core (V_4)

Fig. 6.6: Implementation variants

- For a specific variant, each application is partitioned into tasks whose sizes are suitable to the core size of the implementation. For example, in V_1 variant where the size of each core is equal to the size of one tile in DRP architecture (Fig. 6.6(a)), a certain target application should be divided into tasks in a way that each of which can be implemented using only 64 PEs. If a task cannot fit within one DRP tile, it must be further partitioned into two or more tasks, each of which is small enough to fit in one tile.

- For variants with large core sizes (V_3 and V_4 variants), the number of tasks each target application is partitioned into may be less than eight; or in other words, in such cases, not all cores are used. The decision of how many tasks an application should be partitioned into depends on performance. For each variant, different cases, which have different number of tasks, have been implemented and evaluated. And, the one with the best performance is chosen and presented on Table 6.3.

- The tasks of an application are mapped in such a way that two tasks that need to exchange data are assigned to two cores close to each other to reduce network delay as the number of routers between two cores decreases.

Table 6.3: Implementation results of target applications

Application	Abbr.	Variant	Symbol	Tasks	Throughput [Mbps]	Fragmentation [%]
Discrete Cosine Transform	DCT	1-tile core	V_1	8	142.4	11.9
		2-tile core	V_2	5	267.8	26.7
		3-tile core	V_3	4	297.1	43.7
		4-tile core	V_4	3	291.4	65.3
Viterbi decoder	Viterbi	1-tile core	V_1	6	2.5	26.7
		2-tile core	V_2	4	3.9	33.6
		3-tile core	V_3	4	3.9	57.1
		4-tile core	V_4	3	3.5	66.0
JPEG encoder	JPEG	1-tile core	V_1	8	39.9	17.1
		2-tile core	V_2	6	66.2	29.1
		3-tile core	V_3	5	89.2	30.2
		4-tile core	V_4	4	92.3	51.1
Turbo encoder	Turbo	1-tile core	V_1	8	1.7	18.1
		2-tile core	V_2	5	3.4	31.2
		3-tile core	V_3	4	3.2	37.7
		4-tile core	V_4	4	3.2	49.7
G712 encoder	G721	1-tile core	V_1	8	3.5	19.8
		2-tile core	V_2	6	5.2	30.2
		3-tile core	V_3	5	9.2	40.3
		4-tile core	V_4	4	8.7	51.2

6.4.3.3 Implementation results

Table 6.3 shows the evaluation results for every implemented variants of target applications. Column "Tasks" denotes the number of tasks each application is partitioned into. As mentioned earlier, for variants V_1-V_4, the number of tasks on column "Tasks" is the best implementation case I can achieve in terms of throughput.

6.4.3.4 Throughput

It is easy to see that the throughput of implementations with different core sizes (V_i variants) are considerably varying. The best throughput when the core size is equal to two tiles (V_2) is the implementation of Viterbi and Turbo; in the case of three tiles (V_3) this corresponds to DCT, Viterbi and G721; and when implementing with four tiles (V_4), JPEG implementation achieves the highest throughput. Apparently, the size of cores influences the throughput of applications implemented in a multicore architecture.

- When the core size is small, for example, (V_1) variants, a target application must be partitioned into multiple small tasks in order to fit small cores. This creates a great amount of communication, which causes throughput to drop significantly. With larger core sizes, the number of tasks each application needs to be divided into is reduced; so, less communication is required, and throughput is improved.

- Nonetheless, if the core size becomes large like (V_4), throughput might not be as good as expected. Among the target applications in this study, only JPEG has the best throughput with V_4 variant. A large core can hold a bigger task, so the partition of an application is likely to generate fewer tasks. This reduces the amount of communication among tasks, and the throughput could become better. However, a fewer number of tasks means a smaller number of pipelining stages, and, the workload balance between computation stages is hard to maintain. Therefore, throughput could be reduced. In addition, large cores do not use resources effectively as discussed in the next section.

6.4.4 Internal fragmentation

Internal fragmentation is the fragmentation inside the boundary of a task. This type of fragmentation comes from the rectangular shape of a task. In the proposed multicore system, each task is mapped onto a separated core with a fixed size; and, since the size of task is often smaller than that of a core, some resources may not be used, or they get wasted. There are several types of resources contained in a tile such as PEs, VMEMs, HMEMs, register files and flip-flops, but in this study, only PEs are taken into account for calculating internal fragmentation. For a multicontext DRPA, the total internal fragmentation of an application (F_{app}) can be computed as follows.

$$F_{context\_i}(\%) = \frac{PE_{not\_used\_i}}{N} * 100$$

$$F_{task\_j}(\%) = \frac{\sum_{i=1}^{k} F_{context\_i}}{k}$$

$$F_{app}(\%) = \frac{\sum_{j=1}^{T} F_{task\_j}}{T}$$

In this equation, $F_{context\_i}$ is the fragmentation in i^{th} context. It is the ratio of the number of unused PEs ($PE_{\text{not\_used\_i}}$) in the context to the size of a core (N). F_{task_j} denotes the fragmentation of j^{th} task; N is the size of a core in terms of the number of PEs; k is the total number of context a task requires to implement; and T is the number of tasks the application is partitioned into. According to the above equation, the internal fragmentation of an application is the average of the fragmentation of its tasks.

From the internal fragmentation of implementations in Table 6.3, It is easy to see that the fragmentation is growing as the core size increases, or in other words, the larger a core size is, the less effective resource usage becomes. As a result, the size of cores in a multicore system plays an important role in balancing between throughput and resource usage.

6.5 Conclusion

This section examines two architectures, a tile-based architecture using embedded memory modules arranged as FIFOs for inter-task communication, and a multicore architecture employing an NoC to connect computational cores. Target applications are partitioned into a number of tasks, each of which is mapped onto a tile or a tile group in the tile-based architecture, and onto a core in the multicore architecture. All tasks of an application are arranged to execute in a pipelined manner. Based on NEC's DRP-1, the evaluation result shows that the width of communication channels largely affects on performance. Also, using an NoC to form a multicore architecture proves to be an effective method toward improving the performance of implemented applications, which can be modeled using a pipelined processing manner. Moreover, this section tries to investigate how the size of cores influences the performance of target applications. Evaluation results received from a number of real applications implemented on the system with different core sizes show that the size of core plays an important role in balancing between throughput and resource usage. The most suitable core size in many cases is about the size of two or three tiles of DRP architecture.

Chapter 7

Conclusion and Future Work

7.1 Thesis Summary

The focus of this thesis is to propose, implement and evaluate feasible mechanisms that support to create a multitasking environment for coarse-grained dynamically reconfigurable processors. This is achieved by firstly introducing the methodology to guide research and to organize the thesis. Several concepts including the definitions of applications, hardware tasks and inter-process communications, which are somehow different from that of the microprocessor domain, are described. An outline of a system using a coarse-grained DRPA supporting a multitasking environment is also explained. Moreover, three multitasking models based on resource sharing are introduced.

Secondly the background of coarse-grain DRPAs is examined based on general architectural aspects available on currently existing devices. These aspects can be used as criteria to categorize coarse-grained DRPAs both currently available and future released devices into different classes. Then a number of DRPAs released by consumer-electronics makers in the last decade including not only commercially available products but also new devices under research and development is introduced. Although the target device of research in this thesis is NEC Electronics DRP-1, others described in Table 3.2 could become potential target hardware to apply proposed solutions with relevant changes. Specifically, the proposed preemption algorithms (Chapter 5 and the multicore reconfigurable system (Chapter 6) can be implemented with other DRPAs with minor modifications; the proposed mapping method (Chapter 4) needs to be modified to adapt to other platforms because the method assumes the two features specific to NEC's DRP architecture: using embedded memories as FIFO to exchange data between hardware execution units (tiles), and forming tile groups from separated tiles. As a result, the method can directly apply on platforms that support these two features; and for other devices without either such two characteristics, the mapping algorithm has to be changed.

Thirdly, a systematic method for mapping an application modeled as a Kahn Process Network onto a target DRPA in order to enhance throughput by trying to exploit more inherent parallelism of target applications is proposed and evaluated. By exploring the multi-process execution in dy-

namically reconfigurable processors, which is a technique to enhance throughput through exploiting more inherent parallelism of applications. Basically, a total process for an application is divided into small processes, assigned into limited areas of a reconfigurable array, and concurrently executed in a pipelined manner. In order to do that, the size of tiles, which is a unit area of dynamically reconfigurable array, and the grouping of processes are adjusted. Using real applications such as DCT, JPEG encoder and Turbo encoder, the impact of different versions mapped onto the NEC Dynamically Reconfigurable Processor on performance is evaluated. Evaluation results show that the proposed mapping algorithm achieves the best performance in terms of the throughput and the execution time.

Fourthly, a research for a suitable mechanism to preempt tasks executing on dynamically reconfigurable processors is performed. Basically, when a task is preempted, its necessary state information must be correctly preserved in order for the task to be resumed later. Not only do coarse-grained DRPAs have different architectures using a variety of development tools, but the great amount of state data of hardware tasks executing on such devices are usually distributed on many different storage elements. To address these difficulties, a general method for capturing the state data of hardware tasks targeting coarse-grained DRPAs is proposed and evaluated. Based on resource usage, algorithms for identifying preemption points and inserting preemption states subject to user-specified preemption latency are proposed. Moreover, a modification to automatically incorporate proposed steps into the system design flow is also discussed. The performance degradation caused by additional preemption states is minimized by allowing preemption only at predefined points where demanded resources are small. The evaluation result using a model based on NEC ElectronicsĄf DRP-1 shows that the proposed method can produce preemption points satisfying a given preemption latency with reasonable hardware overhead (from 6% to 15%).

Last, a new multicore reconfigurable system, which is a potential architecture candidate for realizing a multitasking environment, is proposed. As an extension of Chapter 4, a comparision between application impementations using a multi-thread execution model on a tile-based architecture and a multicore architecture is performed. Based on such a multicore reconfigurable architecture, a research on how the size of cores in a multicore reconfigurable system influences on the performance and the resource utilization of target applications is examined and discussed. Recently, an efficient multicore architecture for processors toward multitasking design has emerged as a dominant trend in the chip making industry. As reconfigurable devices gradually prove their capability in improving computation power while preserving flexibility, I examine a multicore reconfigurable architecture composed of multiple hardware execution units or cores connected by an interconnection network. An application is partitioned into a set of tasks, each of which is mapped onto a core for execution in a pipelined manner. Using real applications implemented on the proposed architecture in which cores are based on the tile structure of NEC Electronics' DRP-1, the evaluation result shows that the size of core is a trade-off between throughput and resource usage, and the size equals to two or three DRP tiles is an appropriate choice for many cases.

7.2 Suggestions for Future Research

This thesis presents several key methods toward building a multitasking environment on coarse-grained DRPAs. Although this topic has been the main research subject for GPPs for a long time, and a lot of research results have come into real life through implementations in processor architectures, operating systems, software developments and user realization, related studies in the field of reconfigurable computing are still immature. To date, an effective reconfigurable architecture, an operating system and a software framework that can support a true multitasking environment have not yet appeared. Apart from having a few researches in this field, it is necessary to take into consideration a lot of difficulties to transform equivalent concepts and implementations from software world based on GPPs to hardware world based on reconfigurable devices. In this thesis, I developed only some basic methods for a simple multitasking computational model. As a result, the work opens up a great field for future research. At this point, I recommend following directions for future work

- **Mapping method for hardware tasks on multicore reconfigurable architecture:** When proposing the mapping algorithm in Chapter 4, I assume the two following conditions: (1) Embedded memories are used as FIFOs to exchange data between tiles since a target application is modeled as a KPN. (2) Neighboring tiles can be joined together to form tiles groups. However, it is not always the case for many DRPAs, some of which might not allow embedded memories to work as FIFOs for data communication between hardware execution units (HEUs), others may not have capability to join HEUs and allows them to work independently and in parallel. Moreover, in the proposed multicore reconfigurable system (Chapter 6), cores are connected by an international network not FIFOs; so, an original KPN cannot be directly applied. Besides, it is often difficult to join several cores together to create a core group, a concept similar to a tile group described in Chapter 4. As a result, a new mapping method is necessary for a multicore reconfigurable architecture, where cores cannot communicate using the FIFO mechanism.

- **Simulation environment for improving the proposed preemption algorithm under real situation:** the preemption algorithm introduced in Chapter 5 was evaluated in a limited scenario where only one application occupies and executes on the whole target reconfigurable array. That is why in the evaluation part of Chapter 5, only the maximum preemption latency was provided, not the real preemption latency. In a real situation, multiple applications can be assigned to different execution units for execution in parallel. Then a certain running application may be preempted; the resources occupied by the application are freed; and a new application could be brought in for execution. In such a circumstance, the actual preemption latency of a target application will vary depending on the number of applications in the system at a given time, the moment when a preemption request is issued, and the condition of the system. In addition, the result given in Table 5.1 does not take into account time for transferring

data between the target reconfigurable device and the outside memory. Since the input/output channel of the target device is shared with other applications, sometimes, it could be blocked. All of this happen in a real system and make the actual preemption latency increase. In order to evaluate preemption latency more elaborately, it is necessary to build a simulation environment that can capture all aspects of a real system.

- **Design issues for multicore reconfigurable architectures:** The multicore reconfigurable system proposed in Chapter 6 is just a first step. The next step is to evaluate different structures and configurations such as how different topologies of an interconnection network influence on performance, area overhead and the structure of the system; how large an interconnection network occupies or how much area overhead caused by a network is; how different channel bit widths affects area overhead and throughput; how to map an application consisting of multiple tasks onto a multicore architecture to achieve certain goals; and, how to defragment the system when tasks are dynamically allocated and freed.

- **Design issues for interconnection networks:** As interconnection networks may have great impact on performance, area overhead, wire delay and power of a multicore reconfigurable architecture, they need more attention [28, 118, 124, 123]. One of the solutions to the wire delay problem is three-dimensional IC technology that stacks multiple wafers or dice using vertical interconnects [126, 127, 128, 129]. Stacked dynamically reconfigurable processors are an innovative approach to solve the issue of stretched programmable wires.

- **Prototype of a reconfigurable architecture that supports a multitasking and multi-thread model:** In spite of introducing several mechanisms toward a multitasking environment on DRPAs, a complete system has not yet been build and evaluated. It is crucial to design such a system with proposed solutions in order to examine how well proposed methods work in a general architecture and in a real situation.

Abbreviation and Acronyms

2D	Two-dimensional
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BDL	Behavioral Design Language
CIU	Core Interface Unit
CPU	Central Processing Unit
DCT	Two-dimensional Discrete Cosine Transform
DMU	Data Manipulation Unit
DPU	Datapath Unit (CS2112)
DRP	Dynamically Reconfigurable Processor
DRPA	Dynamically Reconfigurable Processing Array
DSP	Digital Signal Processor, Digital Signal Processing
FFU	Flip-Flop Unit
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
GPP	General Purpose Processor
HDL	Hardware Description Language
HEU	Hardware Execution Unit
HMEM	Horizontal Memory
I/O	Input/Output
ILP	Instruction-Level Parallelism
IMDCT	Inverse Modified Discrete Cosine Transform
IP	Intellectual Property
IRAM	Internal Random Access Memory
JPEG	JPEG encoder
KPN	Kahn Process Network
LUT	Look-up-Table
MAC	Multiply-Accumulate

MEM	Memory module
MIMD	Multiple Instruction Multiple Data
MIPS	Million Instructions Per Second
MLT	Multiply and Accumulate Cell (Hitachi FE-GA)
MPEG	MPEG-2 decoder
MuCCRA	Multi-Core Configurable Reconfigurable Architecture
MULT	Multiplier
NoC	Network-on-Chip
OS	Operating System
PAE	Processing Array Element
PC	Program Counter
PCI	Peripheral Component Interconnect
PE	Processing Element
RC	Reconfigurable Cell (Mophosys)
RCFU	Reconfigurable Functional Unit
RFU	Register File Unit
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SE	Switching Element
SIMD	Single Instruction Multiple Data
SMU	Shift and Mask Unit
SoC	System-on-Chip
STC	State Transition Controller
STG	State Transition Graph
Turbo	Turbo encoder
Viterbi	Viterbi decoder
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
VMEM	Vertical Memory
XPP	eXtreme Processing Platform

Bibliography

[1] R.W. Hartenstein and H. Grunbacher. The Roadmap to Reconfigurable computing. In *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications (FPL'00)*, pages 27–30, August 2000. 1

[2] S. Brown, R. Francis, J. Rose, and Z. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, 1992. 1

[3] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999. 1, 48

[4] S.M. Arnold D. Buell and W.J. Kleinfelder. SPLASH 2: FPGAs in a Custom Computing Machine. *IEEE Computer Society Press*, 1996. 2

[5] E. Tau, I. Eslick, D. Chen, J. Brown, and A. DeHon. SPACE 2 as a Reconfigurable Stream Processor. In *Proceedings of 4th Australasian Conference on Parallel and Real-time Systems (PART'97)*, pages 286–297, September 1997. 2

[6] J. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the 5th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, pages 12–21, April 1997. 2, 26

[7] ITRS. *International Technology Roadmap for Semiconductors 2007 Edition*, 2007. available at `http://www.itrs.net/`. 2

[8] Timothy Mark Pinkston and Jeonghee Shin. Trends Toward On-Chip Networked Microsystems. *International Journal of High Performance Computing and Networking*, 3(1):3–18, September 2005. 2

[9] J. Patterson and H. Agah. Synopsys and Xilinx Unveil Next Generation Flow for Platform FPGAs. *Xcell Journal Online*, 41, September 2001. 2

[10] Xilinx. *Spartan-3A FPGA Family: Data Sheet*, 2005. 2

[11] Xilinx. *Virtex-4 Family Overview*, 2005. 2

[12] Xilinx. *Virtex-5 Family Overview*, 2007. 2

[13] Altera. *Stratix IV Device handbook, Volume 1*, 2008. 2

[14] Hideharu Amano. A Survey on Dynamically Reconfigurable Processors. *IEICE Transactions on Communications*, E89-B(12):3179–3187, December 2006. 2, 48, 67, 86

[15] David Geer. Chip makers turn to multicore processors. *IEEE Computer*, 38(5):11–13, May 2005. 3, 86

[16] V.M. Tuan and H. Amano. A Mapping Method for Multi-Process Execution on Dynamically Reconfigurable Processors. *IEICE Transactions on Information & Systems*, E91-D(9):2312–2322, September 2008. 5, 93

[17] V.M. Tuan and H. Amano. A Preemption Algorithm for a Multitasking Environment on Dynamically Reconfigurable Processors. *IEICE Transactions on Information & Systems*, E91-D(12), December 2008. 5

[18] Ivica Crnkovic. Software Engineering and Science. Presentation on Research Methodology for Natural Sciences and Technology, 2006. `http://www.idt.mdh.se/kurser/ct3340/ht07/se-methods2003.pdf`. 7

[19] S. Hauck and A. Dehon. *Reconfigurable Computing: the theory and practice of FPGA-based computation*. Morgan Kaufmann Publishers, 2008. 9

[20] M. Suzuki, Y. Hasegawa, Y. Yamada, N. Kaneko, K. Deguchi, H. Amano, K. Anjo, M. Motomura, K. Wakabayashi, T. Toi, and T. Awashima. Stream Applications on the Dynamically Reconfigurable Processor. In *Proceedings of International Conference on Field Programmable Technology (FPT 2004)*, pages 137–144, December 2004. 9, 61, 82

[21] O. Diessel, H. Elgindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic scheduling of tasks on partially reconfigurable FPGAs. *IEE Proceedings on Computers and Digital Techniques*, 147(3):181–188, May 2000. 11, 88

[22] Grant Wigley and David Kearney. The management of applications for reconfigurable computing using an operating system. *Australian computer Science Communications*, 24:73–81, January 2002. 11

[23] H. Walder, C. Steiger, and M. Plazner. Fast online task placement on FPGAs: Free space partitioning and 2D-Hashing. In *Proceedings of Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 178–185, April 2003. 11

[24] M. Handa and R. Vemuri. Area fragmentation in reconfigurable operating systems. In *Proceedings of the International Conference Engineering of Reconfigurable Systems and Algorithms (ERSA'2004)*, pages 77–83, June 2004. 11

[25] Scott Hauck. The Roles of FPGAs in Reprogrammable Systems. *Proceedings of the IEEE*, 86(4):615–639, April 1998. 12

[26] H. Krupnova and G. Saucier. FPGA-based emulation: Industrial and custom prototyping silutions. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 68–77, August 2000. 12

[27] Altera Corp. Altera Stratix Device Handbook, July 2005. `http://www.altera.com`. 13

[28] W.J. Dally and B. Towles. Route Packets not Wires: On-Chip Interconnection Networks. In *Proceedings of Design Automation Conference*, pages 684–689, June 2001. 14, 88, 102

[29] J. S. N. Jean, K. Tomko, V. Yavagal, J. Shah, and R. Cook. Dynamic reconfiguration to support concurrent applications. *IEEE Transactions on Computers*, 48:591–602, June 1999. 17, 68

[30] L. Levinson, R. Manner, M.Sesler, and H. Simmler. Preemptive Multitasking on FPGAs. In *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 301–302, April 2000. 17, 68

[31] H. Simmler, L. Levinson, and R. Manner. Multitasking on FPGA Coprocessors. In *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications (FPL'00)*, pages 121–130, January 2000. 17, 68

[32] H.K.H. So and R.W. Brodersen. Improving usability of fpga-based reconfigurable computers through operating system support. In *Proceedings of the 16th International Conference on Field Programmable Logic and Applications (FPL'06)*, pages 1–6, August 2006. 17

[33] M. Middendorf, B. Scheuermann, H. Schmeck, and H. Elgindy. An evolutionary approach to dynamic task scheduling on FPGAs with restricted buffer. *Journal of Parallel and Distributed Computing*, 62:1407–1420, September 2002. 19

[34] J. Teich, S. Fekete, and J. Schepers. Compile-time optimization of dynamic hardware reconfigurations. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTAĄE9)*, pages 1097–1103, June 1999. 19

[35] J. Teich, S. Fekete, and J. Schepers. Optimization of dynamic hardware reconfigurations. *The Journal of Supercomputing*, 19:57–75, May 2001. 19

[36] PACT Inc. *XPP-III Processor Overview*, version 2.0 edition, July 2006. 21, 29, 48

[37] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Transactions on Computers*, 49(5):465–481, May 2000. 22

[38] Guy Lemieux and David Lewis. *Design of interconnection networks for programmable logic*. Kluwer Academic Publishers, 2004. 22

[39] M. Kato, Y. Hasegawa, and H. Amano. Evaluation of MuCCRA-D: A Dynamically Reconfigurable Processor with Directly Interconnected PEs. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'2008)*, pages 353–359, July 2008. 23, 44, 47, 48

[40] K. Compton, J. Cooley, S. Knol, and S. Hauck. Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(3):209–220, June 2002. 23, 88

[41] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A Time-Multiplexed FPGA. In *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM1997)*, pages 22–28, April 1997. 23

[42] E. Tau, I. Eslick, D. Chen, J. Brown, and A. DeHon. A First Generation DPGA Implementation. In *Proceedings of the 3rd Canadian Workshop on Field-Programmable Devices*, May 1995. 23

[43] X. P. Ling and H. Amano. WASMII: A Data Driven Computer on a Virtual Hardware. In *Proceedings of the 1st IEEE Symposium on FPGAs for Custom Computing Machines (FCCM1993)*, pages 33–42, April 1993. 25

[44] Y. Shibata, M. Uno, H. Amano, K. Furuta, T. Fujii, and M. Motomura. A Virtual Hardware System on a Dynamically Reconfigurable Logic Device. In *Proceedings of the 8th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM2000)*, pages 295–296, April 2000. 25

[45] S. Goldstein, H. Schmit, M. Moe, M. Budiuy, S. Cadambi, R. Taylor, and R. Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA1999)*, pages 28–39, May 1999. 25

[46] J. Arnold, D. Buell, and E. Davis. Splash 2. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–322, June 1992. 25