PAPER

A Preemption Algorithm for a Multitasking Environment on Dynamically Reconfigurable Processors

Vu Manh TUAN^{†a)}, *Nonmember* and Hideharu AMANO[†], *Member*

SUMMARY Task preemption is a critical mechanism for building an effective multi-tasking environment on dynamically reconfigurable processors. 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 Dynamically Reconfigurable Processing Array (DRPAs) devices 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, this paper aims at studying a general method for capturing the state data of hardware tasks targeting coarse-grained DRPAs. 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 Electronics' DRP-1 shows that the proposed method can produce preemption points satisfying a given preemption latency with reasonable hardware overhead (from 6% to 15%).

key words: *dynamically reconfigurable processor, preemption algorithm, preemption latency, hardware overhead*

1. Introduction

In order to further exploit the flexibility of reconfigurable devices, operating systems for managing task allocation, scheduling and configuration have been introduced. One of the focus areas of many studies is to build a multitasking environment to allow different tasks to efficiently share a piece of reconfigurable hardware. Among problems for realizing such an environment, a task preemption mechanism, where a higher priority task is allowed to interrupt an executing task, plays an important role. However, such a mechanism for tasks running on hardware circuits is not trivial involving the question of how to suspend and resume a hardware execution, and especially, how to capture the state data of a given task within a certain latency while guaranteeing a reasonable hardware overhead.

A considerable number of coarse-grained DRPAs such as DRP, DAPDNA-2, FE-GA and SAKE [1], which exploit the multi-context architecture to reduce configuration overhead, have been developed and commercialized. By providing storage for multiple configurations in each processing element, hardware configuration can be changed quickly

often in a clock cycle. Compared with fine-grained Field-Programmable Gate Arrays (FPGAs), since data and time for setting configuration from outside are small, task switching involving preemption in DRPAs seems to be more realistic than those for FPGAs. Nonetheless, while scheduling without preemption has been researched [2], and several methods to approach the problem of task preemption on FPGA-based devices have been proposed, only a few research efforts to implement such functions into DRPAs have been carried out.

In this paper, we propose a method for preempting a hardware task and capturing its state data based on the analysis of resource usage at the design time. By modifying the state transition graph of applications implemented on dynamically reconfigurable systems at computation steps where the requirement of resources is small, the impact of preemption on performance and cost for task switching could be reduced.

The rest of the paper is organized as follows. Section 2 briefly introduces related work and research contribution. Section 3 overviews the solution proposed for hardware task preemption. The details of the proposal is described in Sect. 4. The DRP architecture, which is the target device of this study, is presented in Sect. 5. Experimental results are presented in Sect. 6; and finally, Sect. 7 is for summary.

2. Related Work and Research Contribution

2.1 Related Work

Hardware multitasking on FPGAs have been a challenging subject for many researches. [4] deals with the support of concurrent applications in a multi-FPGA system by reconfiguring entire FPGAs. Storing the state information of a hardware task during interruption is discussed in [5]. The outline of a multitasking environment, which enables several tasks to run in parallel, is introduced in [6]. A hardware check-pointing approach for reconfigurable devices is proposed in [10] to divide a task into smaller modules to minimize the lost of computation when failures occur. In order to do so, the state of a task must be saved frequently at pre-defined checkpoints during execution.

One of the most important features to enable a preemptive multitasking environment is task preemption. In a such environment on reconfigurable devices, several solutions for saving and restoring the state data of a hardware task have

Manuscript received May 9, 2008.

[†]The authors are with the Graduate School of Science and Technology, Keio University, Yokohama-shi, 223-8522 Japan.

a) E-mail: vmtuan@am.ics.keio.ac.jp

DOI: 10.1093/ietisy/e91-d.12.2793

been proposed. However, they mainly target fine-grained FPGAs.

- *Readback*: This solution is based on the configuration read-back capability, which allows to read the content of both registers and internal memories [5], [7]–[9]. Although demanding no extra hardware, the solution is slow due to a great amount of data, and requires additional computation to filter out useless information. More importantly, the format of the configuration stream is crucial to extract useful data; so, this makes the solution depend on specific FPGAs.
- *Internal state supervision*: By adding extra interfaces to registers and internal memories, it is possible to access these elements when saving and restoring context data. This solution could be implemented as a scan chain, a memory-mapping structure, or a scan chain with shadow registers [9]–[12]. Although achieving data efficiency as only needed information is saved, this approach demands extra resources and design efforts to implement interfaces to registers and memories.

[3] proposes a systematic methodology for incorporating preemption constraints in application specific multi-task VLSI systems. By considering a predetermined set of applications, the method tries to insert preemption points taking into consideration both dedicated and shared registers in order to minimize the task switching overhead. This approach is suitable for fixed hardware platform like Application Specific Integrated Circuits (ASIC) in which registers to be saved can be determined in advance.

2.2 Research Contribution

The method proposed in this research differs from above-mentioned solutions in the following aspects.

- By modifying the source program at high level languages (C and Verilog) during the design time to insert special states for saving and restoring state data, the method is independent from the details of the underlying hardware architecture such as which registers and memories are assigned to variables at the run time.
- The proposal targets coarse-grained DRPA devices, whose system design flow is tightly and automatically integrated with the necessary steps of the method in order for designers to examine and justify how the method affects implemented applications.
- The impact of the method on performance, hardware overhead and preemption latency can be measured at the design time.

The main contribution of this research includes:

- Algorithms to select and optimize the preemption points of a target application subject to given preemption latency constraints.
- Important steps to integrate the proposed method into

the system design flow in order to automate the process of identifying and optimizing the list of preemption points for a target application during the design time.

3. Preemption Analysis

3.1 Task Switching

While tasks running on a general-purpose processor can be considered as software tasks, hardware tasks are the parts of an application implemented in reconfigurable logic. In this paper, since we target coarse-grained DRPAs, a hardware task can be considered as the representative of a part of an application mapped onto on a DRPA for execution.

In a preemptive multitasking environment, a typical task switching process can be illustrated on Fig. 1. While Task 1 is running, an interrupt signal, often caused by a system timer, indicating a possible task switch is generated. Before a new task (Task 2) can be executed, several preparing stages have to be done. First, an interrupt service is called to decide whether a task switch is necessary (Stage A). If it is, it might take some more times to wait because Task 1 may not be suspended immediately (Stage a'). Then, when Task 1 is ready to be stopped, the state data of Task 1 is saved in Stage B; and, that of Task 2, which had been preserved before, is loaded in Stage C.

(t_1) can be considered as interrupt latency; and, $\sum_{i=2}^4 t_i$ is context switch latency. In many systems and depending on certain tasks, there is no Stage a', or in other words, tasks can be usually suspended immediately when receiving a preemption request. In such cases, time according to Stage a' (t_2) is zero. On the other hand, some systems or certain tasks may not be interrupted right after receiving a request, for example, when the current running task is in a critical area, or when tasks only allows to be interrupted at certain points during execution. In this case, t_2 has a certain value. Generally speaking, the sum $\tau = \sum_{i=1}^4 t_i$ can be considered as preemption latency. Usually, Stage A does not take a long time for modern processors and operating systems. In this paper, we do not take Stage A or interrupt latency (t_1) into consideration, so preemption latency can be computed as follows:

$$\tau = \sum_{i=2}^4 t_i \quad (1)$$

However, Stages B and C may take a considerably amount

Fig. 1 Task switching.

Fig. 2 Memory and register usages vs. computation steps.

of time since the information representing the context of a hardware task is specific for a given implementation and may scatter on different state-holding elements. The amount of data captured when preempting a hardware task is often considerably large; so, the cost for a hardware task preemption mechanism should be minimized.

3.2 Our Approach

The amount of internal registers and memories necessary to store intermediate results is considerably varying during execution. Most target applications of dynamically reconfigurable processors are stream processing, that is, data blocks to be processed are iteratively received in a certain interval. Normally, between the processing of two data blocks, the amount of state data is relatively small. This fact can be applied to build a preemption mechanism where preemption is allowed only at predefined steps called *preemption points* [15] or *switchpoints* [16] during execution.

3.2.1 Preliminary Evaluation

Figure 2 shows requirement for resources in terms of memories and registers in each computation step when an IMDCT, a JPEG encoder and a Turbo encoder are implemented on NEC's DRP-1. X-axis of the figures shows the computation steps, and Y-axis shows the number of registers and memories. As shown in Fig. 2, the number of registers and memories for storing intermediate results and switching contexts greatly varies from step to step. For example, in IMDCT, steps 0, 1, 2, 3, 4, 6, 13, 19, 22, 29 and 34 do not require too many memories and registers; in addition, steps 10, 11, 23, 24 and 29 do not use a lot of memories though the number of registers is remarkable. Accordingly, by only allowing preemption at steps corresponding to steps where the requirement for memories and registers is minimized, the amount of data necessary to be saved can be dramatically reduced.

For a given task, in order to identify which data should be the target for capturing if the task is preempted at a certain computation step, it is critical to examine what resources are required and how they are used. The former can be done by source program analysis to identify variables and

their life-time; and, the latter can be determined by an appropriate simulation. In a typical program, there are two types of variables: global and local, which specify the scope of variables. When a task is preempted, basically, all global variables should be saved; furthermore, local variables relating to the preempted place should also be captured. In addition, during execution, the use of variables is varying based on execution time and computation step.

3.2.2 Solution

In our approach, it is critical to evaluate the usage of variables in order to specify potential locations to be preemption points. Specifically, the proposed method is based on the state transition graph and the resource examination of a target application. This kind of resource evaluation can be done at the early stage of design by synthesis tools. The solution should be automatically done with a certain algorithm, since it must be combined into the design tool in the future. That is, the steps of the proposed method are automatically attached while an application is designed. The following policies are adopted.

- Steps where preemption is allowed are limited at pre-determined points called *preemption points* where the demand of registers and memories is smaller than a certain limitation.
- At preemption points, special states tailored to current contexts are added for flushing and restoring state data.
- Performance degradation resulting from preemption is evaluated in order to optimize preemption points.
- Proposed algorithm are integrated into the system design flow.
- Algorithms for selecting and inserting preemption points are based on the design tools of a target architecture.

3.3 State Transition Graph

The homogeneous synchronous data flow [17] is used as the computational model for applications. The model iteratively

processes semi-finite data blocks arriving in a certain interval. Stream applications, which are the main target of DRPAs, are suitable to be represented by this model. The behavior of a hardware task is often represented in the form of a State Transition Graph (STG) $G(N, E, START, END)$, where N is the set of nodes representing computation states; E is the set of edges showing the transition and data dependence between states; $START \in N$ is a distinguished start node without incoming edges; and $END \in N$ is a distinguished end node without outgoing edges. Figure 8 shows the STG of two tasks, where (a_i, b_i) ($i = 0 \dots n$) shows numbered states, arrows represent possible transitions from states to states, (a_0, b_0) and (a_9, b_6) are start and end states. Transition can be switched conditionally as in state a_2 .

4. Preemption Algorithms

4.1 System Design Flow

Figure 3 shows a typical design flow for DRPAs, in which the compiler or the behavioral synthesis, the technology mapper, the place-and-route and the code generation tool are assumed to be design tools available for the target DRPA. First, a source C-based program and an architecture description are taken as inputs for the behavioral synthesis, which extracts control flows as well as data flows, allocates operation resources, and produces reports about required resources. Then, the technology mapper actually produces the code in the form of hardware description language (HDL) for processing elements, and, a functional simulation at the register transfer level (RTL) can be executed. The place-and-route tool compiles the HDL code into a netlist. Exact reports about resource usage and critical path can be obtained at this step. Finally, the code generator produces configuration code for the underlying reconfigurable hardware.

Figure 3 (b) presents a modified design flow with the proposed preemption algorithm. Since most current DRPAs do not support dynamic memory allocation, the resource re-

Fig. 3 System design flow.

port produced by the compiler could describe quite exactly how variables are allocated and which resources are necessary for task switch. This is the basic for preliminarily analyzing preemption points and inserting preemption states at the step *preemption insertion*. The *evaluation* step is based on the RTL simulation for evaluating how added preemption states affect the implementation. The last step is the *preemption refinement* where preemption points are modified according to the exact report of resource usages and the evaluation result. A modified source program will be fed back to the technology mapper for re-compiling and re-evaluating.

According to the modified design flow, three additional steps are inserted to support the proposed preemption mechanism. Each step requires different input files and produces correspondent results as shown on Fig. 4. At first (Fig. 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. 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. 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 guarantees τ_{input} .

Fig. 4 Input and output information for proposed method.

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 memories.

The algorithm proposed here is consisting of three stages according to modified steps in Fig. 3: inserting preemption states into the original STG, evaluating, and refining.

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.

(1) Variable analysis

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 this state becomes a preemption point.

(2) Potential preemption points analysis

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.

(3) Loop detection:

At the beginning, Algorithm 1 detects all computation loops $L = l_1, l_2, \dots, l_n$ using the given STG. Each loop l_i contains a number of states $l_i = s_{ij}, s_{ij+1}, \dots$. 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 [18], [19] 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.

(4) Sorting:

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

(5) Preemption point finding:

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 . States that do not belong to any computation loops are also investigated to find out preemption points using the threshold θ .

(6) New states insertion:

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.

4.2.2 Refining Preemption Points (Algorithm 2)

In order to prepare for refining the list of preemption points generated by Algorithm 1, it is critical to quantitatively evaluate how inserted preemption states 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.

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.

(1) Preemption point scanning:

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

(2) Unnecessary preemption point elimination:

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.

(3) Preemption state modification:

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.

(4) 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. Target Device

5.1 DRP Architecture

Although the preemption algorithms proposed here can be extended to apply on other reconfigurable devices, in this research, we focus on DRP-1 as the target model. Being a coarse-grain dynamically reconfigurable processor core released by NEC Electronics in 2002 [13], DRP-1 carries an

Fig. 5 DRP tile architecture.

Fig. 6 Processing element architecture.

on-chip configuration data corresponding to multiple contexts, which are dynamically rescheduled to realize multiple functions with one chip.

The primitive unit of the DRP core is called a tile, and a DRP core consists of arbitrary number of tiles. The primitive modules of a tile are processing elements (PEs), a State Transition Controller (STC), a set of 2-ported vertical memory (VMEM) and a pool of 1-ported memory (HMEM) with their controllers. The structure of a tile is shown in Fig. 5. There are 64 PEs located in one tile. The architecture of a PE is shown in Fig. 6. It has an 8-bit ALU, an 8-bit data manipulation unit, sixteen 8-bit register file units, and an 8-bit flip-flop.

The STC is a programmable sequencer where any finite state machine (FSM) can be stored. STC has 64 states, each of which is associated with an instruction pointer. The FSM of STC operates synchronized to the internal clock, and generates the instruction pointer according to states. Also, STC can receive event signals from PEs to branch conditionally.

VMEM is a 2-port memory unit arranged on both sides of a tile; and HMEM is a 1-port memory unit placed on upper and lower boundary of the reconfigurable array. Contents of memories, flip-flops, and register files are all con-

Fig. 7 Prototype chip: DRP-1.

nected and shared by contexts. As shown in Fig. 7, the prototype chip DRP-1 consists of a 8-tile DRP Core, eight 32-bit multipliers, an SRAM controller, a PCI interface, and 256-bit I/Os. It is fabricated with 0.15- μm 8-metal layer CMOS process. The maximum operation frequency is 100-MHz.

An integrated design environment called Muskeeter [14], which includes a high level synthesis tool, a design mapper for DRP, simulators, and a layout/viewer tool, is provided. Applications can be written in a C-based high level hardware description language called Behavioral Design Language (BDL), synthesized, and mapped directly onto the DRP-1 chip for executing.

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.

Fig. 8 Proposed solution.

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

Fig. 9 Example code.

5.2 Illustrative Example

The example code in Fig. 9 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. 8). The algorithm traverses 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 [14], 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, we can avoid the details of the underlying hardware architecture like the exact place of a register variable.

6. Evaluation

The proposed algorithm in this paper is evaluated on a number of real applications shown in Table 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 (Sect. 4.2.1) is applied); when the latency is specified $A_{\tau1}$ and $A_{\tau2}$ (both Algorithm 1 (Sect. 4.2.1) and Algorithm 2 (Sect. 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, Sect. 4.2.2), the critical path (*Delay*), the maximum preemption latency of each implementation (τ_{max}), and the total

Table 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
		$DCT_{\tau1}$	-	-	250	-	247.3	1274
		$DCT_{\tau2}$	-	-	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
		$IMDCT_{\tau1}$	-	-	750	-	748.5	2820
		$IMDCT_{\tau2}$	-	-	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
		$Viterbi_{\tau1}$	-	-	200	-	194.8	959
		$Viterbi_{\tau2}$	-	-	230	-	217.2	938
		$Viterbi_n$	-	-	-	-	828.0	1431
JPEG encoder [21]	JPEG	$JPEG_0$	62	76	-	54.7	34689.2	1851
		$JPEG_1$	-	-	-	-	436.9	2078
		$JPEG_{\tau1}$	-	-	300	-	289.4	2113
		$JPEG_{\tau2}$	-	-	350	-	344.1	2052
		$JPEG_n$	-	-	-	-	378.0	2742
Turbo encoder [23]	Turbo	$Turbo_0$	35	41	-	77.8	1249857.0	3008
		$Turbo_1$	-	-	-	-	712.4	3169
		$Turbo_{\tau1}$	-	-	400	-	389.6	3387
		$Turbo_{\tau2}$	-	-	450	-	413.4	3287
		$Turbo_n$	-	-	-	-	324.0	4232
MPEG-2 decoder [22]	MPEG	$MPEG_0$	89	101	-	67.9	106195.6	2787
		$MPEG_1$	-	-	-	-	751.0	3200
		$MPEG_{\tau1}$	-	-	380	-	351.8	3311
		$MPEG_{\tau2}$	-	-	450	-	419.7	3141
		$MPEG_n$	-	-	-	-	288.0	4445
G721 encoder	G721	$GS M_0$	12	17	-	93.6	50169.6	1085
		$GS M_1$	-	-	-	-	392.4	1126
		$GS M_{\tau1}$	-	-	250	-	237.6	1206
		$GS M_{\tau2}$	-	-	300	-	295.2	1150
		$GS M_n$	-	-	-	-	252.0	1376

number of required PEs (*Used PEs*) respectively.

In Table 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 terminals.

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. 1 and Eq. (1) becomes as follows:

$$\tau = T_p + T_s + T_r \quad (2)$$

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 Eq. (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 1 is the critical path of implementations. Since we do not use the option to pack multiple states into a context, 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.

6.1 Hardware Overhead

Hardware overhead or context switch overhead H specifies

Fig. 10 Hardware overhead.

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: $H = H_{PE}^* - H_{PE}$.

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 1 shows the required hardware resource in term of PEs for our implementations. Using implementations without preemption points (A_0) as the basic, Fig. 10 presents how the hardware overhead varies when preemption points are inserted. In Fig. 10, symbols A_1 , $A_{\tau 1}$, $A_{\tau 2}$ and A_n denote implementations corresponding to Table 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 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.

6.2 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 Eq. (2).

Figure 11 shows the maximum preemption latency for each implementation. Basically, τ_{input} is used as a constraint for optimizing generated preemption points in Algorithm 2 (Sect. 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, JPEG and GSM) the preemption latency of the A_{τ} versions is even smaller than that of the correspondent A_n

Fig. 11 Maximum preemption latency.

Fig. 12 Hardware overhead vs. preemption latency.

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.

6.3 Hardware Overhead and 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, Sect. 4.2.2) and results are presented on Fig. 12, where preemption latency is on X-axis and hardware overhead in the number of PEs is on Y-axis.

When preemption latency becomes larger, hardware overhead tends to be 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 our proposed method achieves its merit, which satisfies a constraint on preemption latency with reasonable hardware overhead.

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.

Acknowledgments

The authors sincerely express their gratitude to NEC Electronics and NEC laboratories.

References

- [1] H. Amano, "A survey on dynamically reconfigurable processors," *IEICE Trans. Commun.*, vol.E89-B, no.12, pp.3179–3187, Dec. 2006.
- [2] J. Noguera and R.M. Badia, "Multitasking on reconfigurable architectures: Microarchitecture support and dynamic scheduling," *ACM Trans. Embedded Computing Systems*, vol.3, no.2, pp.385–406, 2004.
- [3] K. Kim, R. Karri, and M. Potkonjak, "Micropreemption synthesis: An enabling mechanism for multitask VLSI systems," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol.25, no.1, pp.19–30, Jan. 2006.
- [4] J.S.N. Jean, K. Tomko, V. Yavagal, J. Shah, and R. Cook, "Dynamic reconfiguration to support concurrent applications," *IEEE Trans. Comput.*, vol.48, no.6, pp.591–602, June 1999.
- [5] H. Simmler, L. Levinson, and R. Manner, "Multitasking on FPGA coprocessors," *Proc. 10th International Workshop on FPGAs*, pp.121–130, 2000.
- [6] G. Brebner, "The swappable logic unit: A paradigm for virtual hardware," *IEEE Symposium on FPGAs for CCMs*, pp.77–86, 1997.
- [7] L. Levinson, R. Manner, M. Sesler, and H. Simmler, "Preemptive multitasking on FPGAs," *Proc. 2000 IEEE Symposium on FCCM*, 2000.
- [8] S.A. Guccione, D. Levi, and P. Sundararajan, "JBits: A Java-based interface for reconfigurable computing," *Proc. Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, Sept. 1999.
- [9] H. Kalte and M. Porrmann, "Context saving and restoring for multitasking in reconfigurable systems," *Proc. 15th International Conference on FPL*, pp.223–228, Aug. 2005.
- [10] D. Koch, C. Haubelt, and J. Teich, "Efficient hardware checkpointing: Concepts, overhead analysis, and implementation," *Proc. FPGAs*, pp.188–196, Feb. 2007.
- [11] S. Jovanovic, C. Tanougast, and S. Weber, "A hardware preemptive multitasking mechanism based on scan-path register structure for FPGA-based reconfigurable systems," *Proc. NASA/ESA Conference on AHS*, pp.358–364, Aug. 2007.
- [12] M. Ullmann, M. Huebner, B. Grimm, and J. Becker, "An FPGA run-time system for dynamical on-demand reconfiguration," *Proc. 11th*

- Reconfigurable Architectures Workshop (RAW), April 2004.
- [13] M. Motomura, "A dynamically reconfigurable processor architecture," Microprocessor Forum, Oct. 2002.
 - [14] T. Toi, N. Nakamura, Y. Kato, T. Awashima, and K. Wakabayashi, "High-level synthesis challenges and solutions for a dynamically reconfigurable processor," Proc. CICC, Nov. 2006.
 - [15] J. Simonson and J.H. Patel, "Use of preferred preemption points in cache-based real-time systems," Proc. International Computer Performance and Dependability Symposium, pp.316–325, April 1995.
 - [16] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Designing an operating system for a heterogeneous reconfigurable SoC," Proc. International Parallel and Distributed Processing Symposium, Paris, June 2003.
 - [17] E.A. Lee and D.C. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," IEEE Trans. Comput., vol.36, no.1, pp.24–36, 1987.
 - [18] A.V. Aho, R. Sethi, and J.D. Ullman, Compiler: Principles, Techniques, and Tools, Addison Wesley, 1986.
 - [19] V. Sreedhar, G.R. Gao, and Y. Lee, "Identifying loops using DJ graphs," ACM Trans. Programming Languages and Systems (TOPLAS), vol.18, no.6, pp.649–658, Nov. 1996.
 - [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," Proc. International Conference on FPT, pp.137–144, Dec. 2004.
 - [21] Independent JPEG Group, <http://www.ijg.org/>
 - [22] MPEG Software Simulation Group (MSSG), <http://www.mpeg.org/MPEG/MSSG>
 - [23] C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding: Turbo codes," IEEE Trans. Commun., vol.44, no.10, pp.1261–1271, Oct. 1996.

Vu Manh Tuan received the Bachelor degree at Hanoi University of Technology, Vietnam in 1993. In 2000, he received the Master of Science from University of New South Wales, Australia. He is now a third-year Ph.D student at AMANO Laboratory, Department of Information and Computer Science, Keio University. His research interests are reconfigurable architectures and algorithms.

Hideharu Amano received the Ph.D degree from Keio University, Japan in 1986. He is now a Professor in the Department of Information and Computer Science, Keio University. His research interests include the area of parallel architectures and reconfigurable computing.