PAPER Special Section on Reconfigurable Systems

Temporal Partitioning to Amortize Reconfiguration Overhead for Dynamically Reconfigurable Architectures

Jinhwan KIM[†a], ***Student Member***, **Jeonghun CHO**[††b], ***and*** **Tag Gon KIM**[†c], ***Nonmembers***

SUMMARY In these days, many dynamically reconfigurable architectures have been introduced to fill the gap between ASICs and software-programmed processors such as GPPs and DSPs. These reconfigurable architectures have shown to achieve higher performance compared to software-programmed processors. However, reconfigurable architectures suffer from a significant reconfiguration overhead and a speedup limitation. By reducing the reconfiguration overhead, the overall performance of reconfigurable architectures can be improved. Therefore, we will describe temporal partitioning, which are able to amortize the reconfiguration overhead at synthesis phase or compilation time. Our temporal partitioning methodology splits a configuration context into temporal partitions to amortize reconfiguration overhead. And then, we will present benchmark results to demonstrate the effectiveness of our methodology.

key words: *temporal partitioning, reconfigurable architecture, partial reconfiguration, dynamic reconfiguration, run-time reconfiguration and high-level synthesis*

1. Introduction

Recently, *reconfigurable architectures (RAs)* become one of the most promising computing devices to fill the gap between *application specific integrated circuits (ASICs)* and software-programmed processors such as microprocessors, *general-purpose processors (GPPs)*, and *digital signal processors (DSPs)*, because they can provide both flexibility and performance [10], [20]. Since general RAs contain an array of configurable computing units and programmable routing resources, they are able to have flexibility along with performance. RAs are classified into *statically reconfigurable architectures (SRAs)* and *dynamically reconfigurable architectures (DRAs)* based on their reconfiguration scheme. In DRAs, there are two different configuration memory styles; *partial reconfiguration* [3], [7], [23] and *multi-context switching* [16], [21].

To most of *statically reconfigurable architectures (SRAs)*, the ratio of reconfiguration overhead in run-time seems out of concern. They assume that thousands of iterations of single kernel may amortize long configuration overhead, eventually their RAs outperform microprocessors. However, in the case of *dynamically reconfigurable architectures (DRAs)*, the reconfiguration overhead dominates run-time and greatly degrades the overall performance because the configuration may change multiple times even in a single application execution. According to Callahan [24], configuration overhead in Garp was up to 69% (including data transfer). MorphoSys, in spite of its coarse granularity, also wasted 1.6–41% (excluding data transfer) of execution time to configuration [21], depending on loop iteration counts or complexity in applications.

In order to deal with the reconfiguration overhead, there had been intensive research such as *configuration caching* [8], *configuration compression* [5], [9], *configuration prefetching* [12], [15], and temporal partitioning [6], [14]. However, many researches such as configuration caching, configuration compression, and configuration prefetching require the additional hardware units onto reconfigurable architectures. It is too difficult to apply these techniques to various DRAs without the modification of architectures. Ganesan's work [6] and Cardoso's [14] have focused on software efforts such as temporal partitioning. But these works target specific architecture models and have some limitations as described later. In this paper, we will propose a partitioning method which is able to amortize the reconfiguration overhead without additional hardware units. Therefore, our method can be applied to various DRAs.

The paper is organized as follows. First of all, Sect. 2 briefly describes software efforts to hide the reconfiguration overhead in RAs domain. Section 3 outlines our target architecture and execution model. Section 4 explains the proposed temporal partitioning for configuration contexts. Section 5 presents experimental results. Finally, Sect. 6 concludes with a description of our on-going research for this work.

2. Related Work

Ganesan and Vemuri [6] presented a novel partitioning methodology that temporally partitions a design for a partially reconfigurable processor and improves design latency by minimizing reconfiguration overhead. This methodology are integrated in the SPARCS (Synthesis and Partitioning for Adaptive Reconfigurable Computing Systems) design environment [4]. To minimize reconfiguration overhead is achieved by overlapping execution of one temporal partition with the reconfiguration of another, using the partial

Manuscript received April 16, 2007.
Manuscript revised June 29, 2007.

[†]The authors are with the Electrical Engineering & Computer Science Department, Korea Advanced Institute of Science & Technology, Daejeon 305–701, Korea.
[††]The author is with the School of Electrical Engineering & Computer Science, Kyungpook National University, Daegu 702–701, Korea.
a) E-mail: jihkim@smslab.kaist.ac.kr
b) E-mail: jcho@ee.knu.ac.kr
c) E-mail: tkim@ee.kaist.ac.kr

DOI: 10.1093/ietisy/e90–d.12.1977

reconfiguration capability. They have incorporated *block-processing* [6] in the partitioning framework for reducing reconfiguration overhead of partitioned designs. Their proposed methodology was tested on several examples on the Xilinx 6200 FPGA. However, this work uses the very simple model of splitting the available FPGA resources into two same parts and mainly performing temporal partitioning using half of the total available area as the size constraint. Moreover, it is difficult to reduce reconfiguration overhead without block-processing,

The work by Cardoso [14] introduced a temporal partitioning methodology with a *loop dissevering* technique [13] for the coarse-grained XPP architecture [19]. Even though the automated temporal partitioning is supported, his framework requires manual temporal partitioning in order to reduce the reconfiguration overhead. Also, if a loop is not mappable to a reconfigurable array due to resource constraints, a loop dissevering is used. This technique temporally partitions only the loop that oversize the physically available hardware resources. In this case, because the inner loops are partitioned (no more potential for loop pipelining) and each partition may not have require enough computation time over reconfiguration time, the reconfiguration time cannot be hidden by computation time. Thus, the performance is degraded significantly. However, his work shows that a judicious selection of configurations might reduce the overall execution time by overlapping the execution stages needed for each configuration.

3. Reconfiguration Architectures and Execution Model

Usually, RAs are a regular array of configurable computing units, which can be configured to perform different operations and different data routings. Each unit is called differently as Processing Array Element in [19], Reconfigurable Cell in [21], Configurable Logic Block in FPGA domain, or simply Tile in [22]. Afterwards, we will use a term of *Reconfigurable Cells (RCs)* as Singh et al. did in their work. A typical reconfigurable system consists mainly of a host processor and a two-dimensional array of RCs. The host processor controls data and configuration transferred over RC arrays, or executes parts of applications, especially sequential or small parts which are too expensive to run on RC arrays. Each RC may have one or more Arithmetic Logic Units (ALUs) or Look-Up Tables (LUTs) to compute operations, and may have one or more registers to temporally store computation results as operands of successive operations or inputs of other RCs. Also, each RC have switching matrices, segmented buses or simply multiplexers to route data.

In this paper, in order to make experiments practical, we designed an experimental architecture which is very similar to a commercial coarse-grained RA, PACT XPP [19]. Our architecture is a regular array of RCs as shown in Fig. 1 (a). Along both sides of array, there are I/O ports granting accesses to the shared global memory and the configuration memory. Also, our architecture provides partial reconfiguration and its atomic configuration unit is a single RC. As illustrated in Fig. 1 (b), each RC includes a main ALU and two side ALUs, one for forward routing and the other for backward. The main ALU is able to perform most low-level SUIF operations including arithmetic, logical, shift, and multiplier operations. However, each side ALU has a basic arithmetic and logical operators for data routing. In addition, each RC contains two separate routing layers, one for data routing and the other for configuration routing.

Figure 2 shows execution models for static reconfiguration and partial reconfiguration. In these models, the execution of a configuration requires three stages: *fetch*, *configuration*, and *computation*. Fetch stage is related to the load of the configuration data to the RC array. In configuration stage, the download of the configuration data from the configuration memory in the RC array onto the array structures is performed and a local memory or registers in the RC array is initialized with data to be used. In computation stage, the RC array performs pre-defined operations by configuration generation stage in design flow, and generates requested data. In temporal partitioning methods for SRAs, each partition should be occupied onto nearly full reconfigurable array as possible (e.g., [4]). Those schemes only consider another partition after the current one has filled the

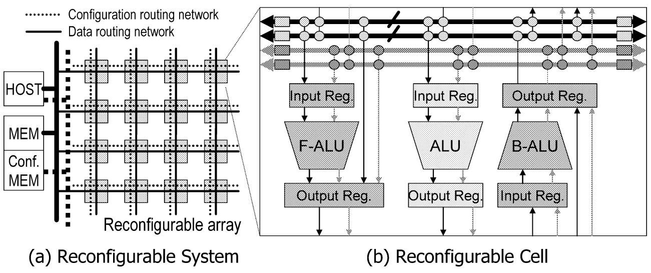

Fig. 1 Typical reconfigurable system.

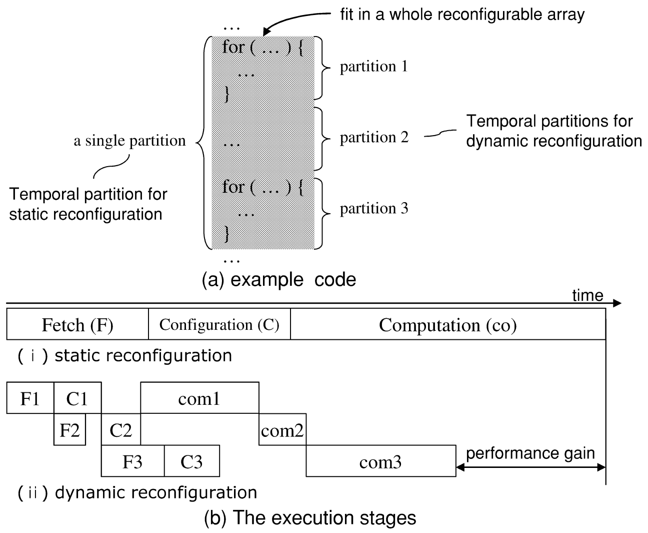

Fig. 2 Execution model for target architectures.

available resources and are insensible to the optimization that must be applied to reduce the overall execution by overlapping the fetching, configuring and computing steps. The execution of this case results in Fig. 2 (b)(i). If this partition is split into three smaller partitions for dynamic reconfiguration as shown in Fig. 2 (a). In DRAs with partial configuration, by overlapping each stage with different stages of a previous partition, execution time is reduced as shown in Fig. 2 (b)(ii). This execution model with partial reconfiguration is our target execution model.

4. Temporal Partitioning

4.1 Framework Overview

Figure 3 shows our compiler framework for reconfigurable systems. This is very similar to a general HW/SW co-design framework. A bold box of this figure represents partitioning process, the scope of this paper. Our framework begins with C Program and partitions it into HW and SW part. The SW part is compiled into the executable by a regular compiler. This executable is executed on host processor simultaneously with collecting results of HW execution. The HW part is first transformed into *SUIF* IR [25] by SUIF front-end. Then, the temporal partitioning splits the whole kernel into a sequence of configuration contexts. After temporal partitioning, *Place & Route* process is performed with SUIF IR and partition information in order to generate configuration contexts for RC array.

Temporal partitioning is divided into two major processes, generating configuration units and merging, as shown in Fig. 3. In the generating process, control-data flow graph (CDFG) of *configuration units* is constructed from SUIF IR. A configuration unit will be explained later in this paper. After the generating process, the merging process begins with CDFG of configuration units. In merging process, each node of CDFG means a single configuration context and grows by merging with other nodes until constraints are fulfilled. Merging process is terminated when all nodes are traversed. Then, a sequence of configuration contexts are generated from SUIF IR with information of configuration partitions through Placement and Routing (P&R) tool. A sequence of configuration contexts lead the amortization of reconfiguration overhead. The detailed discussion of the partitioning process follows.

4.2 Generating Configuration Units

In this step, data flow graphs (DFGs) are extracted from CDFG generated in preprocessor and *configuration units* are generated from these DFGs. A configuration unit represents a certain quantity of configuration data, which can be solely assigned to a single reconfigurable cell (RC). Because a regular coarse-grained DRA can execute a sequence of operations or a complex operation on a single RC in contrast to a regular fine-grained RA, the process of generating configuration units is needed.

The algorithm to generate configuration units works as in Fig. 4. This algorithm traverses DFGs in a depth-first manner. It starts by generating a new configuration unit, and the algorithm adds visited operation nodes to configuration unit until the RC resource constraints are satisfied. The satisfaction of the resource constraints indicates that a configuration unit can be mapped into a RC. These RC resource constraints are the number of operations, the number of input / output ports, and the number of memory ports of a single RC and obtained from machine description written by users. If these constraints are not fulfilled or the current node is not the successor of the previously-visited node, the current configuration unit is completed and the algorithm continues with a new configuration unit including the current configuration unit. When all nodes of DFGs are visited, the algorithm finally generates CDFG of configuration units. In this CDFG, each node runs on a single RC and data flow

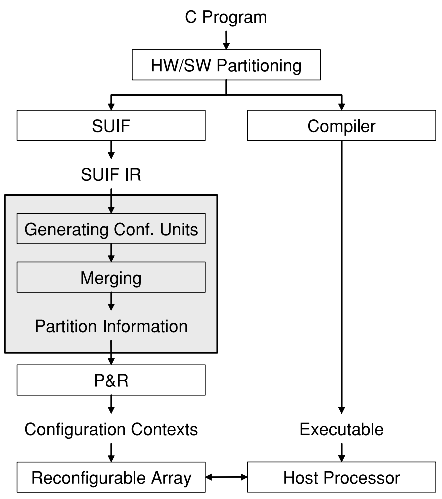

Fig. 3 Compiler framework.

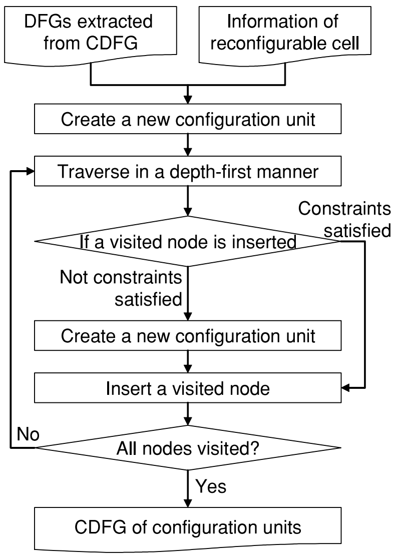
DFGs extracted from CDFG
Information of reconfigurable cell
Create a new configuration unit
Traverse in a depth-first manner
Constraints satisfied
If a visited node is inserted
Not constraints satisfied
Create a new configuration unit
Insert a visited node
No
All nodes visited?
Yes
CDFG of configuration units

Fig. 4 Algorithm to generate configuration units.

indicates routing information between RCs.

4.3 Merging of Candidate Partitions

Merging process begins with CDFG of configuration units. In merging process, each node of CDFG represents a candidate partition and grows by merging with other candidates while constraints are fulfilled. Merging is repeated from node of maximal loop-nesting level (innermost loop) to one of minimal level (outside of outermost loop) in structural order [18] until all nodes are traversed. At each iteration, a speedup due to merging is estimated. If a speedup is achieved, a candidate partition is added to a set of good candidate partitions. Otherwise, a candidate is added to a set of bad candidate partitions. Merging process makes use of a set of bad candidate partitions to only merge with other candidate partitions in order to easily estimate cost.

Before the merging process will be described in detail, we give the following notations.

P_i : a candidate partition i
MEM : total size of configuration memory
M_i : size of configuration memory required by P_i
RC : total number of a reconfigurable cells
RC_i: number of reconfigurable cells required by P_i
T_{F_i} : fetching time of P_i
T_{C_i} : configuration time of P_i
T_{co_i} : computation time of P_i
T_i : exposed execution time of P_i
- the estimated execution time excluding the time overlapped with the preceding partition.

4.3.1 Merging and Time Estimations

Two or three candidate partitions to be merged should be formed as one of patterns in Fig. 5. Each pattern can be merged into a new single candidate partition. In the case of sequential candidate partitions, two adjacent candidate partitions are selected to be merged. A loop can be merged only when each of a loop control part and a loop body part is represented by a single candidate partition, as shown in Fig. 5 (b). In the case of branches, it is possible to merge only when conditional branches can be represented by IF-THEN (-ELSE) in high-level language and each of branch control (IF), THEN, and ELSE part can be represented by a single candidate partition as illustrated in Fig. 5 (c). In order to determine that a new candidate partition can be properly configured onto a reconfigurable array, each pattern should be tested with the following resource constraints. In the following, we write MP to denote a set of all candidate partitions to be merged.

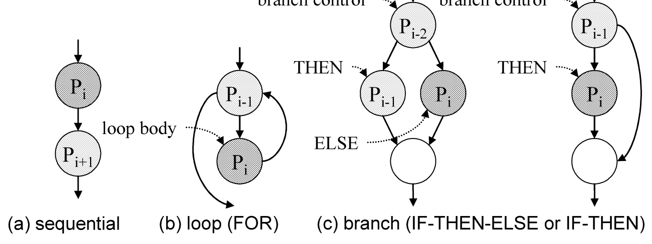

Fig. 5 Merging patterns.

1. Configuration memory constraint

$$\sum_{P_i \in MP} M_i \leq MEM$$

2. Area constraint

$$\sum_{P_i \in MP} RC_i \leq RC$$

If satisfying above constraints, each pattern is merged into a new candidate partition. Fetch, configuration, and computation time of each new one are estimated by the following equations. In these equations, if the number of iterations (N) is unknown at compile-time, an arbitrary value is given by users.

$$T_{C_{new}} = \sum_{P_i \in MP} T_{C_i}$$

$$T_{F_{new}} = \sum_{P_i \in MP} T_{F_i}$$

$$T_{co_{new}} = \begin{cases} \sum_{P_i \in MP} T_{co_i} \text{ ; SEQUENTIAL} \\ N \times (T_{co_{FOR}} + T_{co_{BODY}}) + T_{co_{FOR}}\text{; LOOP} \\ \sum_{P_i \in MP} T_{co_i} \text{ ; IF-THEN} \\ T_{co_{IF}} + max(T_{co_{THEN}}, T_{co_{ELSE}})\text{; IF-THEN-ELSE} \end{cases}$$

4.3.2 Estimations of Performance Gain

After merging each pattern into a new candidate partition as addressed earlier, the total execution time due to a new one will be calculated in order to evaluate the performance gain. However, it is very expensive operation to precisely calculate the total execution time at every merging iterations. Thus, in our framework, performance gain is evaluated by estimating the difference of execution time before/after merging.

Before estimating the difference of execution time before/after merging, the *exposed execution time* should be estimated. Figure 6 shows all possible execution cases of two consequent partitions. The exposed execution time can be estimated by the following equations.

- Case 1: $T_{C_{i-1}} \geq T_{F_i}$ and $T_{co_{i-1}} \leq T_{C_i}$

$$T_i = T_{co_i} + T_{C_i} - T_{co_{i-1}}$$

- Case 2: $T_{C_{i-1}} \leq T_{F_i}$ and $T_{C_{i-1}} + T_{co_{i-1}} \leq T_{F_i} + T_{C_i}$

$$T_i = T_{co_i} + (T_{C_i} + T_{F_i}) - (T_{co_{i-1}} + T_{C_{i-1}})$$

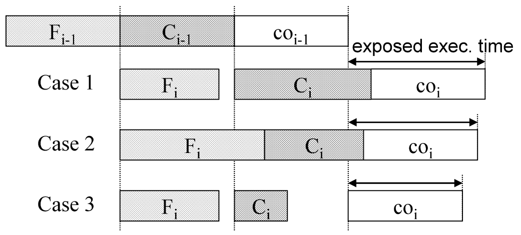

Fig. 6 Estimation of the exposed execution time.

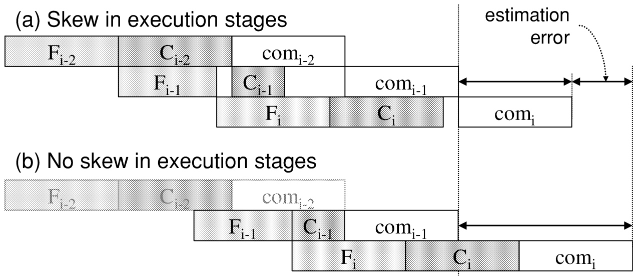

Fig. 7 Estimation error of the exposed execution time.

- Case 3: $T_{C_{i-1}} + T_{co_{i-1}} \geq T_{F_i} + T_{C_i}$

$$T_i = T_{co_i}$$

When the exposed execution time is estimated by the above equations, the estimation error can be occurred. This estimation error in the exposed execution time estimation is caused by the skew in execution stages as illustrated in Fig. 7. However, the estimation error in exposed execution time estimation of sequential candidate partitions is not significant for the overall execution time because the estimated execution time is always longer than the real one. So this error of sequential candidate partitions is ignored in this paper.

Performance gain after merging sequential candidate partitions can be easily evaluated with the above exposed execution time estimation equations and time estimation equations for a new candidate as stated in Sect. 4.3.1. When two sequential candidate partitions P_i and P_{i+1} are merged into a new candidate partition P_{new}, the difference in the total exposed execution time between before and after merging is affected by a subsequent partition P_{i+2} as well as P_{new}. Thus, the exposed execution time of P_{i+2} after merging has to be estimated again. Performance gain in the total exposed execution time can be estimated by the following equation.

$$T_{before\ merging} = T_i + T_{i+1} + T_{i+2}$$
$$T_{after\ merging} = T_{new} + T'_{i+2}$$
$$gain = T_{before\ merging} - T_{after\ merging}$$

As stated earlier, in order to merge a loop into a candidate partition, each of loop control and body parts have to be represented by a single candidate partition. Figure 8 shows how a loop is executed before and after merging. In the figure, P_i represents loop control part and P_{i+1} represents body part. Each exposed execution time of loop control part and body part can be easily estimated by the above exposed execution time estimation equations in the same manner as merging of sequential candidate partitions. However, in every iteration except the first iteration, the exposed execution time of body part can be different, as T_{i+1} and T'_{i+1} shown in Fig. 8. After the first iteration, the configuration stage and computation stage of loop control part can be delayed. The delay causes the exposed execution time of body part to be changed. This estimation error is significant because it is accumulated by loop iterations. Thus, this error should be corrected. From the figure, changed exposed execution time T'_{i+1} is reduced by delay of the configuration time and computation time of loop control part. Those delays can be expressed by $(T_{C_{i+1}} - T_{F_i})$ and $(T_{co_{i+1}} - T_{C_i})$. Therefore, the changed exposed execution time can be estimated by the following equation.

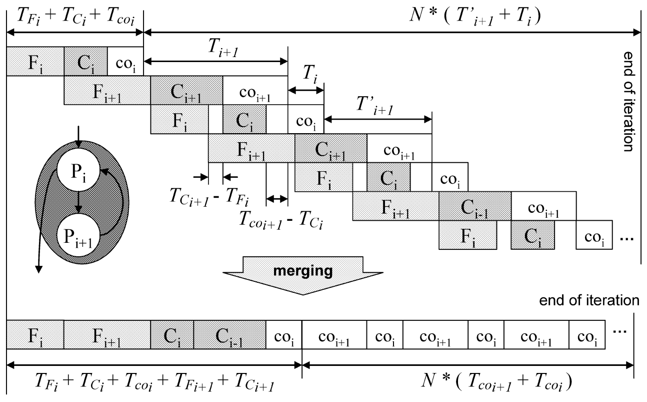

Fig. 8 Merging a loop into a single candidate partition.

$$T'_{i+1} = \begin{cases} T_{i+1} - (T_{C_{i+1}} - T_{F_i}) - (T_{co_{i+1}} - T_{C_i}) \\ \qquad \text{if } T_{F_i} < T_{C_{i+1}} \text{ and } T_{C_i} < T_{co_{i+1}} \\ T_{i+1} - (T_{C_{i+1}} - T_{F_i}) \\ \qquad \text{if } T_{F_i} < T_{C_{i+1}} \text{ and } T_{C_i} > T_{co_{i+1}} \\ T_{i+1} \quad \text{otherwise} \end{cases}$$

Using the previous formulae, we are able to derive the following equation for the estimation of the overall execution time of a loop.

$$T_{loop} = T_{F_i} + T_{C_i} + T_{co_i} + N \times (T_i + T'_{i+1})$$

After merging, because the fetching and configuration of loop is executed only once, the overall execution time can be easily estimated by the following equation without the exposed execution time estimation.

$$T_{new} = T_{F_i} + T_{C_i} + T_{co_i} + T_{F_{i+1}} + T_{C_{i+1}} + N \times (T_{co_{i+1}} + T_{co_i})$$

Therefore, the difference of the total execution time is estimated by the following equation.

$$gain = T_{loop} - T_{new}$$

When the fetching and configuration time cannot be fully amortized in loop iterations, the execution time of loop before merging is longer than one after merging. However, when the fetching and configuration time can be fully amortized in loop iterations, the execution before merging is faster. In order to amortize the fetching and configuration time in loop iterations, the computation time should be sufficient or the fetching and configuration time should be short enough to hide. The increase of computation time can be achieved by merging the innermost loop in nested loops.

4.3.3 Algorithm

An algorithm to perform temporal partitioning of configuration contexts is shown in Fig. 9. An input of algorithm is CDFG of configuration units and a sequence of configuration partitions are generated finally. The algorithm begins with the configuration unit including the first BODY-part of the maximal loop-nesting level (innermost loop) and traverse each configuration unit from the innermost block to the outermost in structural order until all configuration units are traversed. Each iteration of the algorithm performs merging and time estimations of a new candidate partition. These steps are already illustrated in Sect. 4.3.1. The `update_CDFG()` function alters the candidate partitions to be merged into a new hyper-one in CDFG. Then, the performance gain due to merging is estimated as the same manner described in Sect. 4.3.2. If the performance gain is achieved, a new candidate is added into `CP_set`. Even if the performance gain is not achieved, a new candidate is kept as a bad candidate partition in `BCP_set` to be merged with the next candidate partition in structural order. Therefore, the `CP_set` set preserves a good candidate partitions during the merging process. Moreover, at the end of the algorithm, while the `BCP_set` cannot be mapped to RAs, the `CP_set` represents the final set of configuration partitions.

```
$CP\_set$ = {all nodes of input CDFG}
    // set of good candidate partitions
$BCP\_set = \phi$ // set of bad candidate partitions

$P_i$ = first loop-body node of maximal loop-nested level

do {
    $P_{i+1}$ = next_candidate_in_structural_order($P_i$)

    // Section 4.3.1
    // $MP\_set$: set of all candidate partitions to be merged
    // $P_{new}$: a new candidate partition generated from merging
    $MP\_set$ = all_candiates_in_pattern_including($P_i$)
    if (satisfy_resource_constraints($MP\_set$) {
        time_estimations($P_{new}$, $MP\_set$)
        update_CDFG($P_{new}$, $MP\_set$)

        // Section 4.3.2
        gain = gain_estimation($P_{new}$, $MP\_set$)

        if (gain >= 0) { // performance gain
            $BCP\_set -= MP\_set$
            $CP\_set -= MP\_set$
            $CP\_set -= \mathcal{P}(MP\_set)$ // power set of $MP\_set$
            $CP\_set += \{MP\_set\}$
        } else { // no performance gain
            $BCP\_set += MP\_set$
        }

        $P_i = P_{new}$
    } else $P_i = P_{i+1}$
} while (!(All configuration units are traversed))
```

Fig. 9 Algorithm for merging.

5. Experiments

5.1 Experimental Setup

In order to evaluate the performance of the proposed techniques, we use some benchmark kernel selected in two sets of benchmark suites; MediaBench [2] and DSPStone [1]. These benchmark suites have been used widely for evaluating various architectures and software optimizations such as partitioning and compilers. In these benchmark suites, we selected some interesting kernels; *Reference_IDCT* in MPEG2 decoder (MediaBench), *LARp_to_rp* in GSM (MediaBench), and matrix multiplier (DSPStone). These patterns are composed of the nested loops having mostly parallel computations. For example, the *Reference_IDCT* mainly consists of two 3-nested-loops and each nested loop iterates 512 times (matrix multiplication with an 8×8 block). Also, 30.20% of the time is spent in the *Reference_IDCT* routine from the MPEG2 decoder application profiles [11]. Thus, these are suitable for evaluating the partitioning result for dynamically reconfigurable architectures.

Partitioning framework in Sect. 4 has been implemented. It takes C code as the source and, as the target, produces the configuration context for the reconfiguration architecture. In order to minimize the effect of our P&R algorithm [17] and to guarantee that the computation time of each partition is always minimized, our P&R tool maps directly DAGs of each partition onto a reconfigurable array. In other words, each critical path of DAGs is routed linearly from an input port to an output so that the computation time cannot be increased artificially. However, this P&R method requires larger reconfigurable array size than the ordinary array size to implement applications. Therefore, we will show the number of RCs required to implement. The performance of the generated configuration context has been measured by means of our research purpose simulator, which is cycle-accurate and displays cycle counts as the result. The simulator can be tuned with the various hardware parameters including a ratio of configuration time to computation time and the size of reconfigurable array.

5.2 Results

Table 1 and Figs. 10, 11 show the obtained results when applying the proposed partitioning method to *Reference_IDCT* with different reconfigurable architecture parameters and an 8×8 input block. In the table, “# of partitions” shows the number of partitions and “# of RCs (Max/Sum)” represents the number of RCs required by the largest partition

Table 1 Results on Reference_IDCT with various hardware parameters.

Reconfigurable array size		$9\times9 \sim 10\times10$	11×11	12×12	13×13	static
# of partitions		14	7	5	3	1
# of RCs (Max/Sum)		17/46	21/41	23/40	24/38	38
$T_{configuration}$ = 2 cycles/RC	Execution time [cycles] (conf. time / comp. time)	4405	4354	**2729**$\leq^{\dagger}$	4148	**4190** (76/4114)
	Speedup	0.951	0.962	**1.535**	1.01	1
$T_{configuration}$ = 4 cycles/RC	Execution time [cycles] (conf. time / comp. time)	8519	6574	**2874**$\leq^{\dagger}$	4183	**4266** (152/4114)
	Speedup	0.5	0.65	**1.484**	1.02	1
$T_{configuration}$ = 20 cycles/RC	Execution time [cycles] (conf. time / comp. time)	42591	24642	5182	**4455**$\leq^{\dagger}$	**4874** (760/4114)
	Speedup	0.11	0.2	0.9	**1.09**	1

†It means that whenever the reconfigurable array size is larger than this case, the result is same.

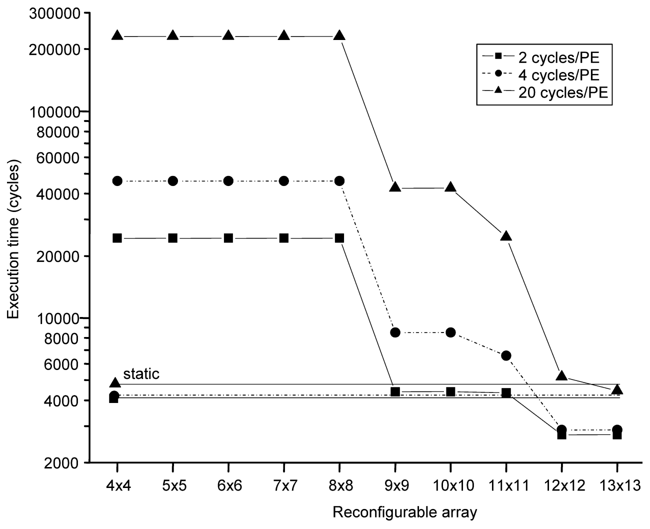

Fig. 10 Total execution time of Reference_IDCT.

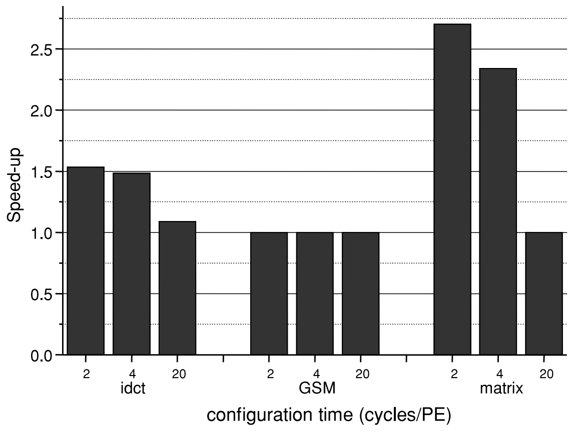

Fig. 12 Speedup on several benchmarks.

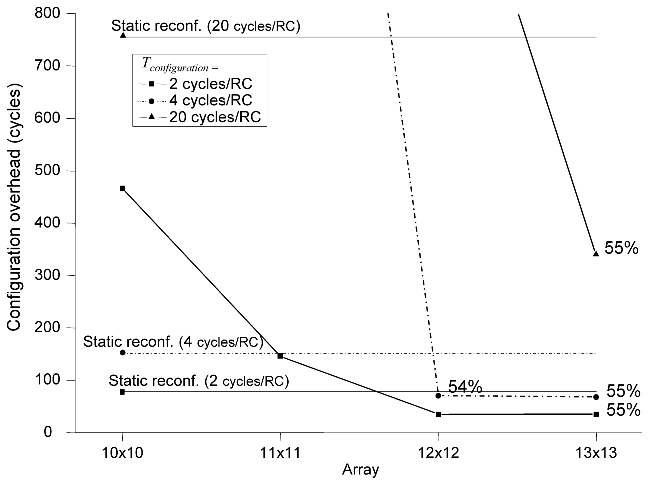

Fig. 11 Reconfiguration overhead of Reference_IDCT.

and the sum of RCs required by all partitions. The last column “static” means that the whole benchmark is mapped on reconfigurable array at once.

When the reconfigurable array size is to 11×11, the overall execution is slow, as shown in Table 1 and Fig. 10. In this range, because each partition in loops have insufficient computation to hide the reconfiguration overhead, the reconfiguration stage of each partition is performed every iteration. Namely, because the innermost loop of *Reference_IDCT* is reconfigured 512 times repeatedly, the reconfiguration overhead is significantly increased. When the reconfigurable array size is 12×12 or 13×13, two inner loops are merged into a single partition. And this merged partition has sufficient computation time so that the reconfiguration time of this partition can be amortized as shown in Fig. 11. Therefore, the reconfiguration overhead is reduced by about 55% compared to the reconfiguration overhead of static reconfiguration and the significant speedups are achieved. One more thing we can notice from Table 1 is that when $T_{configuration}$ is 2 and 4 cycles/RC, the overall execution of the best partitioning results including reconfiguration stages runs faster than the computation stage of static cases. This means that reconfiguration overhead can be fully eliminated when reconfigurable architectures have relatively small configuration time.

Figure 12 shows several speedups obtained with the proposed temporal partitioning and partition placement method compared to the static reconfiguration. This speedup factor is from 1.1 to 2.7 over static reconfiguration. However, in the case of *LARp_to_rp* in GSM, there is no improvement because the whole routine consists of a single loop and the body of that loop has insufficient computation to hide the reconfiguration overhead. In this case, the best performance is given when the whole application is mapped. One thing we can notice from the above results is that the performance using the proposed method is better than mapping the whole application into a single configuration when an application contains a sequence of loops or several nested loops.

6. Conclusions and Future Works

This paper has presented a novel temporal partitioning method to amortize reconfiguration overhead for the reconfigurable architecture with the dynamic nature which can allow the general-purpose applications and the arbitrary number of input blocks. The proposed temporal partitioning method splits a configuration context into temporal partitions in order to amortize the fetch and reconfiguration overhead. When the generated temporal partitions are executed, by overlapping the execution stages (fetch, configuration, and computation stages) on different configurations, the better performance is achieved. The results with *Reference_IDCT* reports that the proposed partitioning method reduces about 55% of reconfiguration overhead. Also, the results show that speedups of 1.1 to 2.7 over static reconfiguration can be achieved. The results strongly confirm that when applications include a sequence of loops or nested loops, the proposed partitioning method may be more effective.

This framework does not currently consider loop transformation techniques such as loop unrolling, loop fission, and loop distribution. The loop transformation techniques may lead to a substantial performance improvement in combination with partitioning of configuration contexts. Therefore, the on-going work focuses on the combination with loop transformation techniques.

References

[1] V. Zivojnovic, J.M. Velarde, C. Schlager, and H. Mey, "DSPstone: A DSP-oriented benchmarking methodology," Proc. Signal Processing Applications & Technology, pp.715–720, Dallas, TX, 1994.

[2] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems," Proc. International Symposium on Microarchitecture, pp.330–335, 1997.

[3] M.B. Gokhale and J.M. Stone, "NAPA C: Compiling for a hybrid RISC/FPGA architecture," Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'98), pp.126–135, 1998.

[4] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri, "An integrated partitioning and synthesis system for dynamically reconfigurable multi-FPGA architectures," 10th Reconfigurable Architecture Workshop (RAW'98), pp.31–36, Orlando, FL, March 1998.

[5] S. Hauck, Z. Li, and E. Schwabe, "Configuration compression for the Xilinx XC6200 FPGA," Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99), pp.138–146, Los Alamitos, CA, 1999.

[6] S. Ganesan and R. Vemuri, "An integrated temporal partitioning and partial reconfiguration technique for design latency improvement," Proc. Conference on Design, Automation and Test in Europe (DATE), Paris, France, March 2000.

[7] Z.A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit," Proc. International Symposium on Computer Architecture (ISCA), pp.225–235, June 2000.

[8] Z. Li, K. Compton, and S. Hauck, "Configuration caching management techniques for reconfigurable computing," Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00), pp.87–96, 2000.

[9] Z. Li and S. Hauck, "Configuration compression for Virtex FPGAs," Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01), pp.147–159, 2001.

[10] R. Hartenstein, "A decade of reconfigurable computing: A visionary retrospective," Proc. Conference on Design, Automation and Test in Europe (DATE'01), pp.642–649, Munich, Germany, 2001.

[11] J.E. Carrillo and P. Chow, "The effect of reconfigurable units in superscalar processors," International Symposium on Field Programmable Gate Arrays, pp.141–150, Monterey, CA, 2001.

[12] Z. Li and S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation," Proc. 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays (FPGA), pp.187–195, 2002.

[13] J.M.P. Cardoso, "Loop dissevering: A technique for temporally partitioning loops in dynamically reconfigurable computing platforms," Proc. 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), p.181b, 2003.

[14] J.M.P. Cardoso and M. Weinhardt, "From C programs to the configure-execute model," Proc. Conference on Design, Automation and Test in Europe (DATE'03), pp.576–581, Munich, Germany, March 2003.

[15] J. Resano, D. Mozos, and F. Catthoor, "A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware," Proc. Conference on Design, Automation and Test in Europe (DATE), pp.106–111, 2005.

[16] P.M. Heysters, G.J.M. Smit, and E. Molenkamp, "Montium — Balancing between energy-efficiency, flexibility and performance," Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'03), pp.235–241, 2003.

[17] S. Jung and T.G. Kim, "An operation and interconnection sharing algorithm for partially reconfigurable architectures," Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'05), pp.163–174, 2005.

[18] S.S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann, 1998.

[19] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP — A self-reconfigurable data processing architecture," J. Supercomput., vol.26, no.2, pp.167–184, 2003.

[20] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," ACM Comput. Surv., vol.34, no.2, pp.171–210, 2002.

[21] H. Singh, M.-H. Lee, G. Lu, N. Bagherzadeh, F.J. Kurdahi, and E.M.C. Filho, "MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications," IEEE Trans. on Computer, vol.49, no.5, pp.465–481, 2000.

[22] R. Barua, W. Lee, S. Arnarasinghe, and A. Agarwal, "Compiler support for scalable and efficient memory systems," IEEE Trans. on Computer, vol.50, no.11, pp.1234–1247, 2001.

[23] S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "PipeRench: A reconfigurable architecture and compiler," Computer, vol.33, no.4, pp.70–77, 2000.

[24] T. Callahan, J. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," Computer, vol.33, no.4, pp.62–69, 2000.

[25] SUIF Compiler System, The SUIF Group. http://suif.stanford.edu

Jinhwan Kim received the B.S. degree in Electronic Engineering with Computer Science minor from Korea Advanced Institute of Science and Technology (KAIST) in 2000, M.S. degree in EECS from KAIST in 2002. He is currently a Ph.D. student in KAIST. His research interests include discrete event systems modeling/simulation, processor design, co-design methodology and compiler optimization techniques for various architectures.

Jeonghun Cho received the B.S. degree in Electrical Engineering from Korea Advanced Institute of Science and Technology (KAIST) in 1996, M.S. degree and Ph.D. degree in EECS from KAIST in 1998 and 2003. He was with the MCU Application Team of Hynix Semiconductors, Korea, where he worked on C Compiler development for 8-bit microprocessors. He is currently as an assistant professor in the School of EECS at the Kyungpook National University in Korea. His research interest includes optimized compiler, operating system, and design automation for embedded systems and reconfigurable computing. He is a member of ACM and IEEE.

Tag Gon Kim received his Ph.D. in computer engineering with specialization in systems modeling/simulation from University of Arizona, Tucson, AZ, 1988. He was a Full-time Instructor at Communication Engineering Department of Bookyung National University, Pusan, Korea between 1980 and 1983, and an Assistant Professor at Electrical and Computer Engineering at University of Kansas, Lawrence, Kansas, U.S.A. from 1989 to 1991. He joined in Electrical Engineering Department of KAIST, Daejeon, Korea in Fall, 1991 as an Assistant Professor and has been a Full Professor at EECS Department since Fall, 1998. His research interests include methodological aspects of systems modeling simulation, analysis of computer/communication networks, and development of simulation environments. He has published more than 150 papers on systems modeling, simulation and analysis in international journals/conference proceedings. He is a co-author (with B.P. Zeigler and H. Praehofer) of Theory of Modeling and Simulation (2nd ed.), Academic Press, 2000. He was the Editor-in-Chief of SIMULATION: Trans. of SCS published by Society for Computer Simulation International (SCS). He is a senior member of IEEE and SCS and a member of Eta Kappa Nu.