PAPER

Configuration Sharing to Reduce Reconfiguration Overhead Using Static Partial Reconfiguration

Sungjoon JUNG[†a)], *Student Member and* **Tag Gon KIM**[†], *Nonmember*

SUMMARY Reconfigurable architectures are one of the most promising solutions satisfying both performance and flexibility. However, reconfiguration overhead in those architectures makes them inappropriate for repetitive reconfigurations. In this paper, we introduce a configuration sharing technique to reduce reconfiguration overhead between similar applications using static partial reconfiguration. Compared to the traditional resource sharing that configures multiple temporal partitions simultaneously and employs a time-multiplexing technique, the proposed configuration sharing reconfigures a device incrementally as an application changes and requires a backend adaptation to reuse configurations between applications. Adopting a data-flow intermediate representation, our compiler framework extends a min-cut placer and a negotiation-based router to deal with the configuration sharing. The results report that the framework could reduce 20% of configuration time at the expense of 1.9% of computation time on average.

key words: *configuration sharing, reconfiguration overhead management, static partial reconfigurable architectures*

1. Introduction

Reconfigurable architectures (RAs), whose logics are modifiable after fabrication depending on applications, have been widely employed in computing domains. Their hardware reconfigurability allows limited resources to perform multiple functions, and satisfies both flexibility and performance. Although reconfiguration overhead in most RAs is reported serious [1]–[3], configuring a kernel and executing it thousands of times could amortize the overhead. However, in RAs that have relatively small number of resources just like coarse-grained RAs, some loops cannot be implemented at one time due to lack of available resources and need to be divided and to be configured repeatedly [4]. Those frequent reconfigurations may make the reconfiguration overhead to overwhelm the computational speedups, and eventually degrade the overall system performance.

To relieve the overhead problem, there have been resource sharing approaches [5]–[8]. Since the repetitive reconfigurations are mainly caused by inappropriately large number of required operations, the sharing technique reduces the number of operations by sharing the common resources, and makes an application to fit on the given devices. When an application consists of several *temporal partitions* that are units to be configured and executed on RAs at one time, and when many operations are common between those partitions, the resource sharing becomes efficient and drastically reduces the required area. Those research on the resource sharing are usually based on time multiplexing, where similar temporal partitions are synthesized concurrently with multiplexers. It means that controlling bits or replacing inputs of multiplexers is enough to change temporal partitions and that the reconfiguration time is relatively short. However, due to insertion of multiplexers and demultiplexers, datapaths are usually lengthened, which harms the computation time [7]. In addition, the time-multiplexing technique still requires large area because all temporal partitions have to be kept valid on a device throughout the whole execution, which is worsened by multiplexer insertions.

The concept of the *configuration sharing* between temporal partitions is basically different from multiplexing resources. The configuration sharing rather emphasizes a static partial reconfiguration, where a successive partition is loaded upon a current partition just between their executions. By reusing common configurations without configuring them again, the proposed approach directly decreases the amount of configuration bitstreams. Compared to the time-multiplexing manner that configures all partitions at one time, the configuration sharing maintains only one partition at a given time, and requires relatively small area. However, it needs reconfiguration to change a temporal partition, and shows high reconfiguration overhead due to frequent reconfigurations, while changing inputs of multiplexers is enough in the time multiplexing method. The idea of reconfiguring a device every time entering each temporal partition is especially useful when loop bodies are not mappable due to resource constraints, for which Cardoso proposed a repetitive configuration technique called loop dissevering [4]. Loop dissevering is a partitioning approach that divides large loop bodies into smaller parts to satisfy the given resource constraints and reconfigures the parts when needed. Kim et al. also introduced another partitioning methodology to balance configuration time and computation time [9]. However, those techniques only concern how to break down applications to satisfy resource constraints and how to distribute and balance configuration overhead and computation time. The approaches do not provide any effort to decrease configuration time, and our sharing technique will further enhance those partitioning techniques with reduced configuration time. Additionally, the proposed sharing approach is not exclusive with but complementary to the context-switching method. Although it provides very

Manuscript received August 10, 2007.
Manuscript revised July 1, 2008.

[†]The authors are with the Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea.
a) E-mail: sjjung@smslab.kaist.ac.kr

DOI: 10.1093/ietisy/e91–d.11.2675

fast reconfiguration, the context-switching needs additional logic for context controlling whose size is linearly proportional to the number of contexts [10]. Due to such hardware constraint, we only have limited number of hardware contexts, and has to continuously load configurations onto contexts. We believe that the proposed configuration sharing may be applied to such configuring processes also.

For the purpose, this paper introduces a framework which performs backend processes of placement and routing for consecutive temporal partitions to increase configuration sharing. The framework eventually reduces the size of resulting configuration bitstreams, and relieves the overhead problem of frequent reconfigurations. *Xilinx BitGen* also performs a similar task named *difference-based partial reconfiguration* to reduce the size of configuration file [11]. However, it is designed to support small modifications after synthesis, which makes it inappropriate for source-level changes or similarities. Therefore, it inherently differs from our framework that considers multiple temporal partitions from the beginning.

The rest of the paper is organized as follows. In Sect. 2, we introduce the configuration sharing problem between consecutive temporal partitions and its relation to backend processes. Section 3 presents a partial reconfiguration model and defines our experimental architecture. Section 4 explains the compiler framework and some implementation issues in placement and routing. Section 5 reports the results of configuration overhead reduction with the proposed framework. Finally, the paper concludes with Sect. 6.

2. Configuration Sharing and Backend Process

2.1 Configuration Sharing Motivation

As explained previously, the partial reconfigurability enables to reconfigure only the differences in configurations between temporal partitions, which can be utilized to reduce the reconfiguration overhead. One simple motivational example is depicted in Fig. 1. Figure 1 (a) shows two temporal partitions, where one is dependent on the other and therefore two are sequentially executed. As shown, two operations of a multiply and an addition (grey boxes) are common in both partitions. In Fig. 1 (b), configurations for two temporal partitions are depicted. Because two common operations fortunately have the same locations on the device, it is sufficient to configure only three operations (white operations in the second temporal partition) in order to construct the second configuration upon the first one. This reusing of the common resources reduces the configuration overhead of the second temporal partition from 5 cells to 3 without any insertion of multiplexers or demultiplexers. This motivational example introduces the configuration sharing that is different from the ordinary time-multiplexed resource sharing. Since the configuration sharing maintains only one partition at a given time as well as no requirement in multiplexers, it is more applicable to relatively harsh area constraints compared to time-multiplexing, but need repetitive reconfigurations to change temporal partitions.

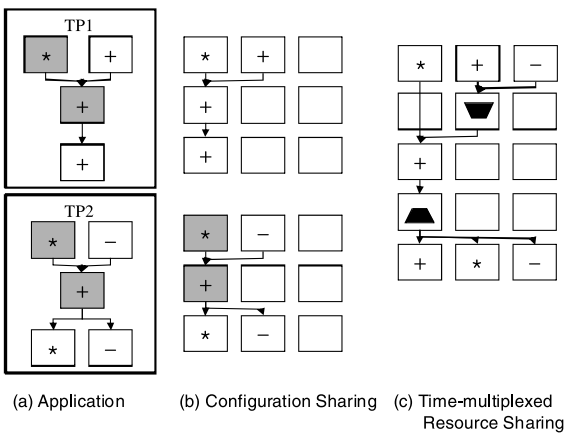


Fig. 1 Configuration sharing motivation.

2.2 Relation to Backend Process

Since resource sharing is a traditional problem in high level synthesis, there have been many works on sharing, placement and routing for time-multiplexing [5]–[8]. However, it is hard to directly employ the approaches to share configurations between temporal partitions and to reduce reconfiguration overhead for several reasons.

First, the main reason for which we cannot apply previous placement techniques is the difference in run-time model. Although both approaches deal with multiple temporal partitions, the time-multiplexing configures a device once at the beginning, while the configuration sharing needs reconfigurations every time changing a temporal partition. Let us look at Fig. 1 again. Figure 1 (b) is an implementation with the configuration sharing, while Fig. 1 (c) is with the time-multiplexing. As shown, both can successfully share common resources of a multiply and an addition. However, the difference occurs in placing non-shared operations. The time-multiplexing needs two separate resources in the first row to place an addition and a subtract although they belong to different temporal partitions, while the configuration sharing allocates them to only one resource. Therefore, the example explains that the placer should manage resource usages separately with temporal partitions as well as it has to assign common operations in the same locations and to keep critical paths short just like the traditional placer.

Second, we cannot employ the existing routing approaches, because they are not designed for configuration sharing. Most RAs have their own smallest addressable or reconfigurable units, *Frames* in Xilinx Virtex series, *Processing Array Elements* in PACT XPP, and so forth. It is not allowed to further divide an addressable unit and to modify just a single bit for switches or wiring points. Even when we want to reconfigure a single connection, we have to update a whole reconfigurable unit containing it. Therefore, in order to keep common resources actually shared between temporal partitions and to reduce the reconfiguration over-

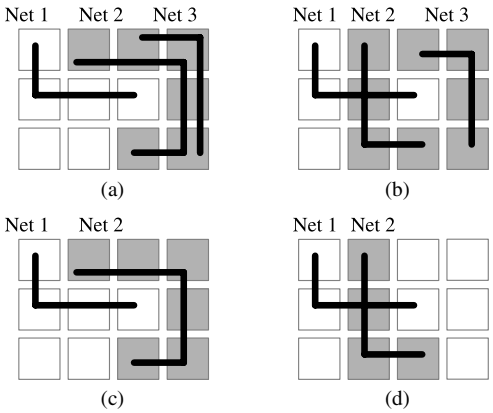


Fig. 2 Configuration sharing example.

head, elements used in successive temporal partitions should be carefully routed not to cross the already placed resources with the constraint of the smallest addressable units.

Figure 2 details the effect of routing in configuration sharing problem. Let us assume that there are two temporal partitions and three nets, and that the former partition requires only Net 1, and the latter needs all three nets. It means that Net 1 is reusable in the latter partition. When a cell has resources enough for routing, Fig. 2 (a) and (b) show two different configurations of the three nets, where gray boxes represent configurations required for the latter partition. Although different configurations may result from different placements, let us assume the locations of all sources and sinks are fixed just to exemplify the effect of routing to the configuration sharing. As depicted, Fig. 2 (a) results a complete reuse of Net 1 at the cost of path length of Net 2, while Fig. 2 (b) keeps Net 2 short but has to reconfigure a cell in Net 1. As a result, Fig. 2 (a) requires 6 cells to be configured, while Fig. 2 (b) needs 8 cells. If more cells in shared nets are reconfigured, the sharing becomes less effective and the size of configuration gets larger. This simple example explains that the router has to be redesigned not to harm the potential configuration sharing by taking rather winding paths instead of shortest paths. However, such longer paths may result inefficient implementation with longer computation time, and the problem to select either Fig. 2 (a) or (b) must depend on criticality or congestions. When Net 2 is along the critical path or the resource constraint of Net 3 is harsh, Fig. 2 (b) becomes more acceptable than Fig. 2 (a).

However, maximizing the reusability does not necessarily mean minimizing the reconfiguration overhead. Figure 2 (c) and (d) show the counter-example that is the same to Fig. 2 (a) and (b) respectively except Net 3. In Fig. 2 (c), shared configurations are not overwritten at the cost of the delay of Net 2, and 6 cells should be configured. On the other hand, Fig. 2 (d) does not sacrifice Net 2, and also achieves small configurations of only 4 cells. This is due to non-existence of Net 3, and explains that traversing free cells while avoiding the shared configurations may not reduce the total configuration overhead. Therefore, the configuration sharing may become ineffective with sparse nets where most cells remain free. Fortunately, many applications suffer from hard resource constraints of RAs, which means that applications with sparse nets are rare.

As explained with the previous examples, resources that a graph algorithm discovers sharable does not necessarily represent that the corresponding configurations are reusable in consecutive temporal partitions. Although configurations of those resources still remain potentially sharable, the configuration sharing rather depends on how to place and route resources, which requires enhancements in the backend processes.

3. Architecture Model

This section explains the partial reconfiguration model in which we are especially interested, and details our experimental architecture.

3.1 Partial Reconfiguration Model

Since the reconfiguration overhead seriously degrades a system performance, some state-of-the-art RAs including Virtex [11] and PACT XPP [12] support partial reconfiguration. Partial reconfiguration is to change only a portion of configuration including functionality, routing, or constants without completely reconfiguring the entire device. There are two distinguishable classes in partial reconfiguration: *dynamic* and *static* partial reconfiguration [11]. In dynamic partial reconfiguration, a device can be reconfigured while the remainder still remains active. However, dynamic partial reconfiguration may support limited reconfiguration since there would be live data on a device and any careless reconfiguration may produce data hazard like loss or collision, called *internal contention*. On the other hand, static partial reconfiguration requires a device to be completely reconfigured before active, but it may provide more flexibility in reconfiguration since there is no risk of internal contention.

In this paper, an application is a list of temporal partitions. At the compile time, temporal partitions are synthesized to manage which and how many resources are required for the partitions. Such resources are configured onto a device when the program reaches the entry point of each temporal partition, and released when the partition completes the given tasks. Once the partition terminates, its resources are no longer required, and can be simply flushed out, or reused for the subsequent partition. This execution model guarantees there is no live data on the device, and makes static partial reconfiguration proper to the inter-temporal-partition configuration sharing, which provides more simplicity and flexibility in reconfiguration.

3.2 Experimental Architecture

To make experiments practical, we designed an experimental system as shown in Fig. 3. The system includes a general purpose processor, a global shared memory, and a configuration controller as well as a regular array of configurable

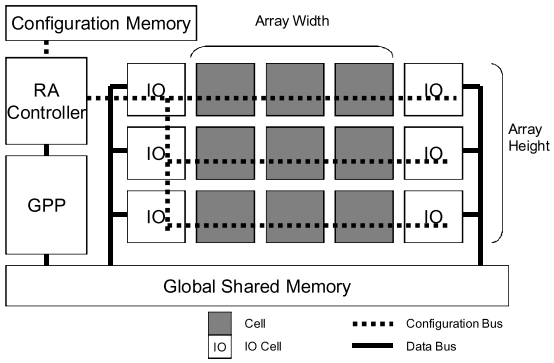


Fig. 3 System architecture.

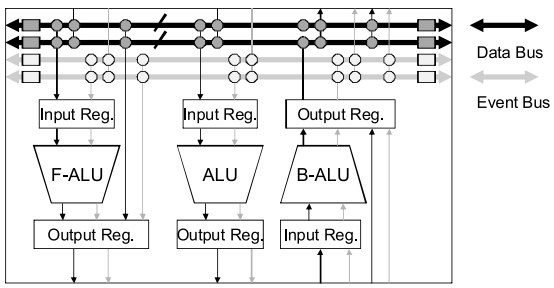


Fig. 4 Cell architecture.

computing and routing elements, called *cells*. We adopted the Harvard architecture to separate the data memory and the configuration memory. In the system, the computing array with the RA controller serves as a coprocessor[†], for which the controller provides interfaces such as loading configurations and writing run-time variables. When loading configurations, the RA controller brings up the configuration bitstreams from the configuration memory, and passes them to the leftmost-topmost cell. The bitstream subsequently flows through the array in a top-to-bottom, left-to-right pipelined manner, and configures the target cell. The computing array is able to access the global shared memory via IO ports along both sides of the array. We also constrained the parallel memory access up to 4 words per cycle.

The cell architecture, which is very similar to PACT XPP [12], a commercial coarse grained architecture, is shown in Fig. 4. As illustrated, a cell consists of one main ALU and two vertical routing elements, one for forward and the other for backward routing. Each vertical routing element also has a basic computing unit to support simple arithmetic and logical operations during routing. To keep the synthesis rules simple, we designed that the main ALU is able to perform most low-level SUIF constructs, and that the side ALUs are dedicated to IR-specific operators like ones to resolve control-dependencies.

Figure 4 also explains two independent routing networks. One is for data, and the other is for events including predicates and tokens that usually manage control and memory dependencies. Each network consists of horizontal and vertical buses. During vertical routing, data or event may be buffered at output registers, or simply bypassed. However, a long routing path cannot satisfy timing constraint without buffers, and needs buffers to be inserted along the path. The inserted buffers also become reusable resources in consecutive temporal partitions. Horizontal bus is also segmented by programmable wiring points at both sides. These specific definitions on switches, wiring points, buffers, and operations provide us a complete encoding. The configuration bitstream required to configure a single cell is of 13 words long excluding run-time constants. As we assumed a configuration bus bandwidth 1 word wide, configuring a cell consumes at least 13 cycles which can be further extended with constant initializations. Moreover, as mentioned previously, the configuration bitstream flows to the target cell in a pipelined manner, which causes propagation delay. Therefore, run-time constants and cell locations may greatly affect reconfiguration time, and we need a simulation to accurately analyze the effect of the configuration sharing.

About data passing protocol of a cell, we adopted the same synchronization mechanism that PACT XPP proposes. An ALU computes when all required data and event inputs are available on input registers. After a computation, the contents of the input registers are consumed and ready for another data, while the output registers hold output values until all successors receive them. The data passing protocol of the cell architecture is the perfect match to our IR that is based on a data-flow machine.

Finally, the main differences between our experimental architecture and PACT XPP are instruction set architectures, bus bandwidths, array sizes, memory controller, bus delays and especially an encoding scheme to generate configuration binary streams that is an unavailable commercial property. However, the other characteristics including three ALUs per cell, two kinds of buses, horizontal bus structures, and even data passing protocols, are kept as similar as possible to PACT XPP according to available papers [12]–[14]. Regarding configuration sharing, such differences between two architectures especially in available resources may result different configurations, but they do not hurt the main theme of this paper to reduce the reconfiguration overhead by sharing consecutive configurations.

4. Backend Process for Configuration Sharing

In this section, we introduce our compiler framework, and describe extensions for traditional placement and routing techniques to solve the configuration sharing problem.

4.1 Overall Framework

This section describes our compiler framework, which is very similar to a conventional hardware-software co-design framework. As we addressed in Sect. 2.2, the configuration sharing is highly affected by placement and routing. To our

[†]The coprocessor instruction set is implemented using SimIt-ARM interfaces for external devices.

best knowledge, compiler framework being aware of the configuration sharing is not available, which is the reason we designed our own framework. Figure 5 shows the entire toolflow, where a bold box represents our main implementation scope. First, C program is partitioned into hardware and software parts. While the software part is compiled by a regular C compiler, the hardware part is transformed into SUIF [15] by the SUIF frontend. To alleviate the complexity in synthesis rules, the SUIF transformer, *porky*, dismantles high level constructs into low level ones. Arrays, loops and complex operations are transformed into pointers, conditional branches, and simple operations. Low SUIF, in turns, is translated into our IR.

For the compiler framework, we also designed an IR similar to Pegasus that has been proven synthesizable [16]. Our IR assumes a data-flow machine and converts control dependencies and memory side-effects into explicit data dependencies using predicates and tokens. As mentioned in Sect. 3, the assumption on a data-flow machine is a good match for our cell architecture whose computation begins with all required inputs ready.

After IR generation, resource sharing on DFGs follows. Based on a graph model and a resource sharing algorithm introduced in [17], resources are annotated with sharing information, and passed to placer and router for further precise and efficient configuration sharing. We perform placement again with loosening the area constraint when routing fails due to the resource constraint. After routing all dataflows, both configuration streams and stub codes are generated. Stub codes include configuration loading instructions and run-time variable passing instructions, which are integrated into and simultaneously compiled with the application's software parts.

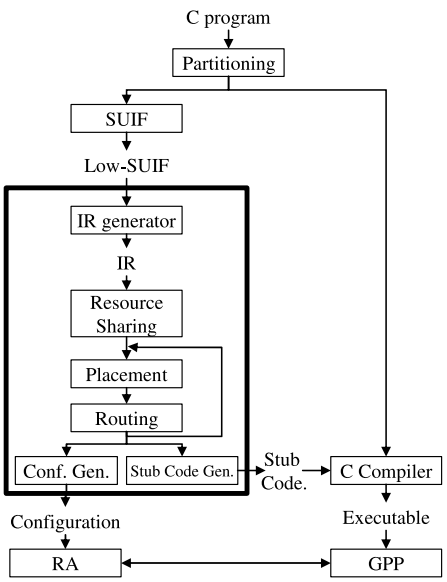


Fig. 5 Compiler framework.

4.2 Placement and Routing

Considering the configuration sharing problem, we carefully designed a backend for placement and routing. Thanks to the cell architecture executing most low SUIF constructs, technology mapping in gate level becomes less important. Instead, we could concentrate our effort to placement and routing. As mentioned in Sect. 2.2, the configuration sharing problem is tightly related to placement and routing, and needs some enhancements in the processes.

Among many available placement algorithms, we adopted a min-cut based placement algorithm that is reported having reasonable computation time and yet generating effective placements in [18]. Min-cut placement algorithm repeatedly divides the given resources into several *spatial partitions* and distributes operations while minimizing the number of nets cut generated by the partitions. The net-cut minimization is usually performed with *moving* operations to the other spatial partition or with *swapping* operations between partitions. Our min-cut placement algorithm is based on a well-known linear-time heuristic by Fiduccia and Mattheyses [19], and further extended with the terminal propagation [20].

As mentioned in Sect. 2.2, placement need to handle multiple temporal partitions simultaneously to place the common operations on the same locations while separately managing resource usages for each temporal partition. Since the original min-cut placement algorithm only deals with operations in a single temporal partition, we enhanced it for multiple temporal partitions, and modified moving and swapping methods. Figure 6 details our placement technique.

Figure 6 (a) shows two temporal partitions, TP_A and TP_B, and their DFGs. Nodes named A_{number} or B_{number} stand for operations that are not sharable and only dedicated to their own temporal partitions. A node named S_1 is the only sharable operation between two partitions. Let us assume that there are 4 computational resources on the device which are vertically partitioned into two, as annotated with SP_1 and SP_2 in Fig. 6 (b). Therefore, there arises a resource constraint of 2 resources per each spatial partition which we have to satisfy at any stage of placement. In other words, *a set of nodes in a spatial partition cannot exceed the re-*

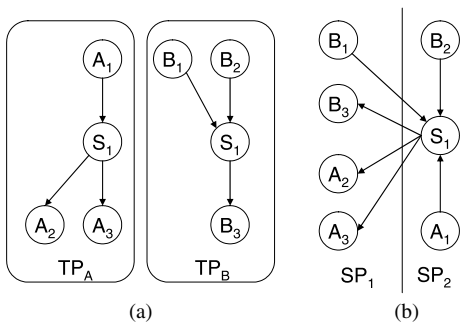


Fig. 6 Placement example.

source constraint at the viewpoint of any temporal partition. To make it formal, let us define SP_i and TP_j to be sets of operations in a spatial partition i and a temporal partition j respectively. Also, $R(n_i)$ is a set of resources required for a node n_i, while $R(SP_j)$ is a set of all resources in SP_j. When TP and SP represent complete sets of temporal partitions and spatial partitions, the resource constraint is defined as follows. For all $TP_i \in TP$ and $SP_j \in SP$,

$$\sum_{n_k \in TP_i \cap SP_j} R(n_k) \leq R(SP_j) \tag{1}$$

One of the valid placements is depicted in Fig. 6 (b). At the viewpoint of TP_A, SP_1 and SP_2 have $\{A_2, A_3\}$ and $\{S_1, A_1\}$, which satisfies Eq. (1) with 2 resources per each spatial partition. It is the same with TP_B.

With the constraint, we also define moving and swapping of nodes to decrease nets cut. Moving a node n is relatively simple as long as the target spatial partition have free resources $R(n)$ throughout all temporal partitions containing n. On the other hand, nodes to swap may have overlapping or non-overlapping temporal partitions, and managing resource usages throughout the temporal partitions is rather complicated. Sometimes, it is not sufficient to swap two nodes but two sets of nodes to conserve the resource constraint. Since swapping a set of nodes and testing the validity requires a sequence of pairwise comparisons, we define next rules to simplify the process.

For NS_1 and NS_2 that are subsets of SP_1 and SP_2 respectively, the swapping of two sets is possible if

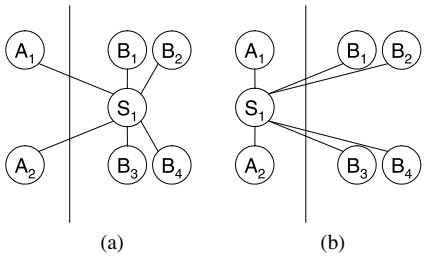
$$\bigcap_{n_i \in NS_1} TP(n_i) = \phi, \text{ if } |NS_1| > 1 \tag{2}$$

$$\bigcap_{n_j \in NS_2} TP(n_j) = \phi, \text{ if } |NS_2| > 1 \tag{3}$$

$$\bigcup_{n_i \in NS_1} TP(n_i) = \bigcup_{n_j \in NS_2} TP(n_j) \tag{4}$$

where $TP(n)$ is a set of temporal partitions containing node n. For example, $\{A_1\} \Leftrightarrow \{A_3\}$ or $\{S_1\} \Leftrightarrow \{A_2, B_3\}$ is a valid swapping, while $\{B_1\} \Leftrightarrow \{A_1\}$ is not. Although $\{S_1, A_1\} \Leftrightarrow \{A_2, A_3, B_1\}$ seems fair and fulfills the resource constraint, it is prohibited due to Eq. (2) and Eq. (3). Instead, the swapping could be divided into two independent ones of $\{S_1\} \Leftrightarrow \{A_2, B_1\}$ and $\{A_1\} \Leftrightarrow \{A_3\}$, which makes the process simple. Equation (4) also forbids $\{S_1\} \Leftrightarrow \{A_2\}$ even if there is a free resource in $TP_B \cap SP_1$. Instead, B_2 first moves to SP_1, then $\{S_1\} \Leftrightarrow \{A_2, B_2\}$ follows, which results the exactly same cell distribution to a swapping $\{S_1\} \Leftrightarrow \{A_2\}$. Although we intend pairwise comparisons for the process to be as small as possible with Eq. (2)–(4), swapping still requires a heavy computation. Therefore, it is only performed when the moving process to reduce nets cut is not available during the linear-time heuristic by Fiduccia and Mattheyses [19].

Unlike the original placement only considering a single temporal partition, the proposed one deals with all nets simultaneously. It means that nets cut in one temporal partition may affect nets cut in another partition. Figure 7 shows



(a) (b)

Fig. 7 Temporal partition and its effect on criticality.

the case. In the figure, A_i and B_j stand for operations in TP_A and TP_B respectively, while S_1 belongs to both temporal partitions. When A_i and B_j require specific locations in SP_1 and SP_2, a usual min-cut placer tends to choose Fig. 7 (a) rather than Fig. 7 (b), since the former placement generates less nets cut than the latter. It explains that routing paths in TP_A may increase to reduce those in TP_B. However, Fig. 7 (b) becomes more acceptable when the net A_1-S_1-A_2 is along the critical path in TP_A. Therefore, we also weight nets to reduce critical path length instead of minimizing nets cut. Referring to [21], the slack ratio A_{ij} of a net ij could be defined as a ratio of the longest path length containing the net to the maximum path length. For a critical net whose slack ratio is near to 1, we imposed high weight so that the placer would not cut the net.

The IR annotated with the placement information is, in turns, passed to a router that is based on *PathFinder*, a negotiation-based routing algorithm [21]. The PathFinder algorithm models two important routing factors of delay and congestion in a single cost function:

$$C_n = A_{ij} d_n + (1 - A_{ij})(d_n + h_n) p_n$$

where d_n is a delay of the node n, h_n is a congestion history, and p_n is a current congestion. Since A_{ij} is a slack ratio compared to the critical path, a net along the critical path, $A_{ij} = 1$, tends to minimize the delay term, while a net in a less critical net will take extra delay to reduce congestion.

The main requirement in designing a router for the configuration sharing is that it has to separate shared resource from non-shared ones and should take rather longer paths instead of shortest paths to maximize the reused configurations as explained in Sect. 2.2. Therefore, we modify the cost function like below.

$$C'_n = C_n + R_n$$

In the cost function, R_n denotes a configuration sharing based term, whose value is either 0 (means non-shared and free-to-use) or an empirical constant r_n (stands for shared cell). When the routing algorithm begins, R_n is set to r_n for all shared cells and the routing algorithm tries to avoid traversing those cells at the cost of delay or congestion. However, when delay or congestion become so high that R_n is ignorable to C_n, then the router will no longer avoid but finally use those shared cells. Once a shared cell is reconfigured for routing of non-shared nets, there is no necessity for the router to keep avoiding it. R_n for the cell is set to 0, a

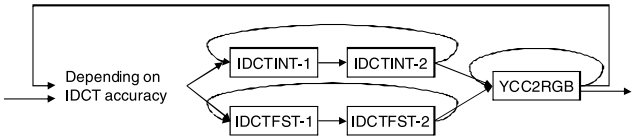


Fig. 8 JPEG decompression process.

congestion history term h_n is cleared, and the router will try to use the cell without any restriction from the beginning.

5. Experiment

We conducted experiments to examine the performance of the proposed framework to reduce configuration overhead for repetitive reconfigurations. Out of MediaBench [22], we used several functions that have interesting patterns to leverage the configuration sharing. Those functions are mainly composed of regular and repetitive computations of data inputs or streams, and such regularity guarantees many of resources to be shared. The selected benchmarks include JPEG image compressor/decompressor, G.721 speech encoder/decoder, and MPEG motion compressor/decompressor.

To explain benchmark characteristics, let us exemplify the JPEG decoder, a standardized decompression method for images. Among many routines in JPEG, we selected two IDCT routines, IDCTFST that is fast but rather inaccurate and IDCTINT that is slow but accurate, and YCC2RGB routine that is used for general colorspace conversion. Figure 8 shows the JPEG decompression flows. As IDCT is composed of two sequential loops that processing columns and rows respectively, we divided the kernels into two independent ones (tagged with -1 and -2) to save areas. In run-time, these kernels become loaded on the architecture depending on accuracy options. For instance, in an accurate integer IDCT case (the upper path in Fig. 8), IDCTINT-1 is loaded first, and followed by IDCTINT-2 after computation. Since IDCT requires tens of iterations, these two configurations become repeatedly loaded. After finishing IDCT computations, the colorspace conversion follows, and IDCT on next image frames begins again. In the decoder example, without the configuration sharing, we have to configure all required cells every time before IDCTINT-1 and IDCTINT-2. However, adopting the proposed configuration sharing approach, we can only configure the differences between two kernels to construct them. Fortunately, two separated IDCT loops have very similar data access and manipulation patterns that are proper to the sharing technique, while the colorspace conversion routine is absolutely different from the IDCT loops, and requires full reconfiguration. For the remaining benchmarks, we also similarly selected routines for the experiments.

Before presenting results on the configuration sharing, Table 1 gives resource characteristics that are obtained with our resource sharing algorithm as explained in Fig. 5. In the table, first two columns named *P*1 and *P*2 in each benchmark represent resource characteristics of two temporal partitions like operations and interconnections. Simply speaking, the number explains how many vertices and edges are in data flow graphs. Although a vertex is usually implemented with a cell, actual hardware resources required for an edge highly depend on placement and routing, and this table only provides potential possibility to share such resources like buses or buffers.

Table 1 Resource sharing.

Exp.	P1	P2	Shared	% Shared
IDCTINT	651	620	188	30.3 %
IDCTFST	592	585	179	30.6%
FDCTINT	336	343	245	72.9%
FDCTFST	295	298	240	81.4%
G721.encode	691	586	428	73.0%
G721.decode	691	586	428	73.0%
MPEG.encode	455	579	267	58.7%
MPEG.decode	466	590	274	58.8%
Avg.				51.6%

Table 2 Configuration size (bytes).

Exp.	Non shared		Shared		% Tot.
	P1	P2	P1	P2	
IDCTINT	11,685	12,413	10,901	11,525	93.1%
IDCTFST	11,961	12,309	10,193	9,869	82.7%
FDCTINT	5,989	5,993	3,917	4,185	67.6%
FDCTFST	5,765	5,821	2,441	3,065	47.5%
G721.enc	5,501	5,493	4,961	4,949	90.1%
G721.dec	5,501	5,493	4,961	4,949	90.1%
MPEG.enc	8,877	11,425	8,707	9,537	89.9%
MPEG.dec	9,037	12,161	8,997	9,421	86.8%
Avg.					81.0%

The next column named *Shared* explains the resulting shareable resources between two partitions by our sharing algorithm. The last column shows the same value that is normalized against the number of smaller resources among two partitions. Therefore, 100% of resource sharing represents one partition is a subset of the other. According to the table, the IDCT routines usually seem less efficient in resource sharing than the FDCT and G.721 routines. This is partly because the operations and dataflows in IDCT show different patterns, and partly because heavy graph sizes multiply the search space for the resource sharing algorithm. For G.721, the same result for encoding and decoding is because two benchmarks share the same kernels. Depending on the data dependency of the benchmarks and the size of search space, the algorithm could share 30.3–81.4% of resources, 51.6% on average.

Although we provided the graph level sharing, it is not necessary to conclude the same ratio of configuration bitstreams could be reused. Instead, the exact reconfiguration bitstreams are only available after placing and routing those resources, and Table 2 summarizes the configuration binary file sizes. In the table, Part. 1 and 2 represent two temporal partitions in each benchmark. While the column named *Non shared* explains the configuration sizes without the configuration sharing, the *Shared* column denotes the configuration sizes required to construct each partition based on the

other partition. For instance, in IDCTINT, 10,901 bytes are required to configure Part. 1 when Part. 2 exists on the device, while 11,525 bytes required to load Part. 2 upon Part. 1. The last column *% Tot.* represents the ratio of the sum of two configuration sizes. According to the table, the sum of configuration sizes reduces from 6.9% in IDCTINT to 52.5% in FDCTFST. On average, the configuration file reduces 19.0% in size. We can notice that the configuration sharing becomes more effective in the FDCT routines than in the IDCT routines just like Table 1 where more resources in FDCTs are shared than in IDCTs. It seems that the configuration sharing is related to the resource sharing, and concludes that the efficient resource sharing algorithm may help the configuration sharing.

Table 3 shows the RA area that is required to execute each benchmark. Since two partitions are executed sequentially, the results do not mean the sum of the areas of two partitions, but represent the larger area among them. According to the table, the shared cases seem to need slightly more areas than the non-shared cases. It is because that the shared cases require more spaces for routing unshared resources so as not to cross the already placed shared cells.

Even though Table 2 provides configuration bitstream sizes, the accurate configuration cycles in our pipelined model only be available with run-time evaluation. Moreover, the configuration sharing is usually done at the cost of path length as discussed in Sect. 2.2, which may increase the computation time and decrease the overall sharing effectiveness. Therefore, we also conducted cycle-accurate run-time experiments with our RA simulator in conjunction with the SimIt-ARM simulator. As depicted in Sect. 3, we assumed that the configuration propagates throughout the architecture in a pipelined fashion, and considered delays due to buffers that were inserted to match timing constraints.

Table 3 Allocated area (column × row cells).

Exp.	Non shared	Shared
IDCTINT	12 × 18	12 × 19
IDCTFST	12 × 18	12 × 18
FDCTINT	12 × 8	12 × 8
FDCTFST	12 × 8	12 × 9
G721.enc	12 × 8	12 × 9
G721.dec	12 × 8	12 × 9
MPEG.enc	12 × 18	12 × 19
MPEG.dec	12 × 19	12 × 19

Table 4 shows computation cycles and configuration cycles for the benchmarks. To show how much reconfiguration overhead is reduced, we carried out two independent experiments with and without the configuration sharing respectively. The other parameters were kept the same for both experiments. In the tables, *Computation* and *Configuration* are cycles for computation and configuration, while *Total Execution* is the sum of the other two. The percentage in the *Shared* column represents the normalization against the corresponding value in the *Non shared* case. As shown, the repetitive reconfigurations makes the configuration overhead in all benchmarks comparable to the computation cycles. In MPEG benchmarks, relatively short computation time in kernels permits the configuration overhead to be more than half the total execution time. For those benchmarks having high reconfiguration overhead, the proposed framework reduces 20.0% of configuration cycles on average. The reduction is mainly due to small configuration sizes as shown in Table 2, and seems directly proportional to the decease in bitstream sizes. While the configuration overhead is successfully reduced, the reduction is achieved at the cost of 1.9% longer computation cycles due to winding nets not to touch the shared resources. As a result, the total execution time decreases by 7.1% on average. The results explain that the proposed technique relieves the reconfiguration overhead with the small increases in computation time and is especially helpful for repetitive reconfigurations.

Table 5 shows comparisons of the proposed configuration sharing to other time-multiplexed resource sharing. The time-multiplexing usually aims to reduce the required area by sharing common operations, which is the same with our goal because the reduced area means that the configuration time is also decreased in partial reconfigurable architectures. Table 5 summarizes reductions in area or LUT usages. Since the reductions are highly dependent on benchmarks, partitioning units, or architecture models, the table

Table 5 Comparisons to other time-multiplexed resource sharing.

Approaches	% Area (Shared/Non-shared)
Lin et al. [5]	85.8% LUTs
Fischer et al. [6]	52.6 – 77.7%
Mondal and Memik [8]	50.8 – 91.1
Memik et al. [7]	56%
Proposed approach	47.5 – 93.1%

Table 4 Configuration and computation cycles.

	Non shared			Shared		
Exp.	Total Execution	Computation	Configuration	Total Execution	Computation	Configuration
IDCTINT	1,571,615	924,434	647,181	1,528,459 (97.3%)	944,432 (102.2%)	584,027 (90.2%)
IDCTFST	1,525,661	892,521	633,140	1,416,042 (92.8%)	894,649 (100.2%)	521,393 (82.4%)
FDCTINT	965,383	659,632	305,751	883,956 (91.2%)	680,907 (103.2%)	203,049 (66.4%)
FDCTFST	952,192	657,079	295,113	799,642 (84.0%)	665,164 (101.2%)	134,478 (45.6%)
G721.enc	3,773,414	2,053,514	1,719,900	3,597,241 (95.3%)	2,054,550 (100.1%)	1,542,691 (89.7%)
G721.dec	3,779,264	2,059,364	1,719,900	3,603,390 (95.3%)	2,060,699 (100.2%)	1,542,691 (89.7%)
MPEG.enc	741,888	268,800	473,088	712,786 (96.1%)	288,768 (107.4%)	424,018 (89.6%)
MPEG.dec	2,298,240	814,464	1,483,776	2,103,720 (91.5%)	820,224 (100.7%)	1,283,496 (86.5%)
Average				92.9%	101.9%	80.0%

does not mean that one approach is better or worse than another, but only provides just simple comparisons. As depicted in the table, the resource sharing could reduce the area down to 50.8% at the best case. On the other hand, the configuration overhead is decreased down to 47.5% in the case of FDCTFST with the proposed configuration sharing, which is better than or competitive with the other time-multiplexed resource sharing techniques. Moreover, two approaches of Mondal's and Memik's adopt JPEG Jdmerge in MediaBench as their benchmarks, which enables additional comparisons of the proposed one. In a case of Jdmerge, Mondal's could reduce the area down to 53%, and Memik's to 52%, while the proposed method decreased the configuration size down to 54.7%. Although the results seem to be very similar, the meanings are absolutely different. For two approaches using the time-multiplexing, the resulting area of 52% or 53% is the size that should be maintained throughout the entire execution time as two temporal partitions have to remain valid as shown in Fig. 1 (c). However, in the configuration sharing, 54.7% only means the sum of two temporal partitions that do not need to be configured at one time. Therefore, the required area for the proposed case is much smaller than the remaining time-multiplexing ones, which explains that it is possible to implement the given tasks on smaller devices. In Memik's work, there is also mentioned the increase in computation time. As explained in Sect. 2.1, time-multiplexing requires multiplexer/demultiplexer insertions, which results the increase in critical paths and eventually in computation time. Memik denotes that computation time is lengthened by 6% on average. However, the proposed configuration sharing is free from multiplexers/demultiplexers, and harms computation time only by 1.9%, which is relatively smaller than Memik's.

Currently, we do not have plentiful optimizations in the compiler frameworks. The lack of pointer analysis results that many false dependencies including memory dependencies and loop carried dependencies remain unresolved. The false dependencies make the benchmarks run rather sequentially, and our system executes the same kernels approximately twice faster than ARM when we consider configuration time, computation time, and run-time variable initialization time, altogether. We believe that further implementation on pointer optimizations may boost the system performance, although the technique do not directly affect the effectiveness of the configuration sharing. Instead, we could figure out architecture specific placement issues that are likely to enhance the configuration sharing. Since a cell has three independent ALU components, cramming as many operations as possible into a cell will help to decrease the configuration size. However, placing shared resources and non-shared ones together will obviously hazard the configuration sharing. Moreover, the lack of horizontal bus at the bottom of a cell makes any use of ALUs require buses of the next cell. Therefore, we need a careful placement of non-shared operations not to interfere shared operations with any use of horizontal buses. We believe that those additional architecture-specific placement techniques remain further implementation issues.

6. Conclusion

In this paper, we addressed the configuration sharing problem using static partial reconfiguration. The configuration sharing that directly decreases the amount of configuration bitstream sizes between similar temporal partitions is different from the traditional time-multiplexed resource sharing. In order to ensure that configurations are reused through consecutive temporal partitions, enhancements in placement and routing are required to avoid rewriting the already configured resources. We proposed a min-cut placement algorithm to cope with multiple temporal partitions, and introduced a cost function for a negotiation-based router. The results report that the proposed configuration sharing reduces 20% of configuration time at the expense of 1.9% of computation time on average, and explain that the approach becomes reasonable for repetitive reconfigurations.

References

[1] T. Callahan, J. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," Computer, vol.33, pp.62–69, 2000.
[2] 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. Comput., vol.49, no.5, pp.465–481, 2000.
[3] Z. Li, Configuration Management Techiniques for Reconfigurable Computing, Ph.D. Thesis, Northwestern University, 2002.
[4] J.M.P. Cardoso, "Loop dissevering: A technique for temporally partitioning loops in dynamically reconfigurable computing platforms," IPDPS '03: Proc. 17th International Symposium on Parallel and Distributed Processing, p.181.2, IEEE Computer Society, 2003.
[5] C.-C. Lin, D. Chang, Y.L. Wu, and M. Marek-Sadowska, "Time-multiplexed routing resources for FPGA design," IEEE Custom Integrated Circuits Conference, pp.152–155, 1996.
[6] V. Fischer, M. Drutarovsky, P. Chodowiec, and F. Gramain, "InvMixColumn decomposition and multilevel resource sharing in AES implementations," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.13, no.8, pp.989–992, 2005.
[7] S.O. Memik, G. Memik, R. Jafari, and E. Kursun, "Global resource sharing for synthesis of control data flow graphs on FPGAs," Proc. 40th Design Automation Conference, pp.604–609, ACM, 2003.
[8] S. Mondal and S. Öğrenci Memik, "Resource sharing in pipelined CDFG synthesis," ASP-DAC '05: Proc. 2005 Conference on Asia South Pacific Design Automation, pp.795–798, New York, NY, USA, ACM, 2005.
[9] J. Kim, J. Cho, and T.G. Kim, "Temporal partitioning to amortize reconfiguration overhead for dynamically reconfigurable architectures," IEICE Trans. Inf. & Syst., vol.E90-D, no.12, pp.1977–1985, Dec. 2007.
[10] K. Puttegowda, D.I. Lehn, J.H. Park, P. Athanas, and M. Jones, "Context switching in a run-time reconfigurable system," J. Supercomput., vol.26, no.3, pp.239–257, 2003.
[11] Xilinx, Virtex-II Pro Platform FPGAs: Complete Data Sheet, 2004.
[12] V. Baumgrarte, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP—A self-reconfigurable data processing architecture," International Conference on Engineering of Reconfigurable Systems and Algorithms, 2001.
[13] PACT XPP Technologies, XPP Core Turorial for XDS 3.2, 2002.
[14] PACT XPP Technologies, PACT XPP64-A Reconfigurable Proces-

sor.
[15] R.P. Wilson, R.S. French, C.S. Wilson, S.P. Amarasinghe, J.A.M. Anderson, S.W.K. Tjiang, S.W. Liao, C.W. Tseng, M.W. Hall, M.S. Lam, and J.L. Hennessy, "SUIF: An infrastructure for research on parallelizing and optimizing compilers," SIGPLAN Notices, vol.29, no.12, pp.31–37, 1994.
[16] M. Budiu, Spartial Computation, Ph.D. Thesis, Carnegie Mellon University, 2003.
[17] S. Jung and T.G. Kim, "An operation and interconnection sharing algorithm for partially reconfigurable architectures," ERSA, pp.163–174, 2005.
[18] K. Shahookar and P. Mazumder, "VLSI cell placement techniques," ACM Comput. Surv., vol.23, no.2, pp.143–220, 1991.
[19] C.M. Fiduccia and R.M. Mattheyses, "A linear-time heuristic for improving network partitions," DAC '82: Proc. 19th Conference on Design Automation, pp.175–181, Piscataway, NJ, USA, IEEE Press, 1982.
[20] A.E. Dunlop and B.W. Kernighan, "A procedure for placement of standard-cell VLSI circuits," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.4, no.1, pp.92–98, 1985.
[21] L. McMurchie and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for FPGAs," FPGA, pp.111–117, 1995.
[22] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems," MICRO, pp.330–335, 1997.

Sungjoon Jung received his B.S., M.S., and Ph.D. degrees in electrical engineering from Korea Advanced Institute of Science and Technology (KAIST) in 2000, 2002, and 2008, respectively. He is currently working at KAIST as a post-doctoral researcher. His research interests include a conventional software compiler and a hardware compiler as well as synthesis algorithms for reconfigurable architectures.

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.