High-Level Synthesis of Gracefully Degradable ASICs [*]

Wah Chan
Dept. of Electrical and Computer Engineering
University of California, San Diego
La Jolla, CA 92093-0114

Alex Orailoğlu
Dept. of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

Abstract

We propose a novel graceful degradation scheme, L/U reconfiguration, which can tolerate a single permanent fault in each hardware class of ASIC datapaths. In the proposed scheme, dynamic hardware rebinding and operation rescheduling are performed by a systematic perturbation of the original configuration. A high-level synthesis procedure, which automatically generates such fault-tolerant systems, is also presented. Experiments show that our reconfigurable AISC designs, as compared to optimal non-fault-tolerant designs, achieve optimal pre-reconfiguration and near-optimal post-reconfiguration speed performance.

1 Introduction

Graceful degradation, a time redundancy approach [2], tolerates permanent faults by using dynamic reconfiguration. Unlike standby sparing, graceful degradation enables high levels of hardware utilization both before and after reconfiguration and demands smaller levels of interconnection. It is most suitable for applications where small chip area and superior initial performance are required. However, implementing graceful degradation is not simple because reconfiguration implies dynamic reordering of operations and concomitant rearrangement of resource usage. These two tasks raise the complexity of the controller and interconnect, and as a result, graceful degradation is usually applied to systems with structural or functional regularity such as memory systems [5] and multi-processor arrays [6]. In this paper, we will demonstrate that, by using careful hardware planning and intelligent design methodology, this fault-tolerance technique can be practical for general ASICs that lack structural and functional regularity.

The intelligent exploration methodology required for designing such complex fault-tolerant systems can be delivered by high-level synthesis. The nature of the reconfigurable design problem requires sophisticated planning in both operation scheduling and hardware binding, and these tasks fall squarely within the problem domain of high-level synthesis. In fact, high-level synthesis of fault-tolerant designs has been investigated and implemented in [3], [7] and [8]. Moreover, the procedural approaches in high-level synthesis advocate a systematic definition of the underlying architecture, which in turn assists efficient planning of reconfigurable system designs.

In this paper, we propose a novel reconfiguration scheme and a high-level synthesis methodology to provide graceful degradation capability to general ASIC datapaths. Our objectives are reduction of controller and interconnect overhead, reduction of performance degradation, and increase in hardware utilization. The proposed fault-tolerance scheme, called L/U reconfiguration, targets a single permanent fault in each functional unit class included in the datapaths and provides a single level of reconfiguration. Any fault detection and location technique can be applied independently with this proposed scheme. Fault diagnosis issues are consequently orthogonal to the central focus of this paper and are not herein further covered.

The organization of the rest of the paper is as follows: Section 2 presents the reconfiguration scheme and outlines a set of synthesis constraints imposed by incorporating reconfiguration. Section 3 covers the associated high-level synthesis procedure. Experimental results are given in section 4 and conclusions provided in section 5.

2 L/U Reconfiguration

In this section, we examine the details of the L/U reconfiguration scheme with particular emphasis on its conceptual mechanism; the hardware implementation aspects are given in [1]. We start first with the basic building block of the proposed scheme, namely the canonical L/U block. The canonical L/U block can deliver full pre- and post-reconfiguration hardware utilization. However, it imposes severe restrictions on block size and component types, and additionally worsens the performance degradation after reconfiguration. Subsequently, we extend the canonical form to handle large dataflow graphs (DFG). As partial ordering of operations in any given DFG may be violated by rescheduling, we outline a set of synthesis constraints on scheduling and resource usage to resolve any data and structural hazards. Finally, we extend the proposed scheme to cover heterogeneous hardware classes.

2.1 Canonical L/U Reconfiguration

Graceful degradation tolerates permanent faults by fault detection, fault location, and fault recovery. Upon locating a

[*]This work is supported by the National Science Foundation under grant number MIP-9308535.

faulty hardware unit, fault recovery is triggered to perform hardware reconfiguration. The faulty unit is deactivated and operational status of the system is regained. Without standby sparing, other existing hardware units must replace the faulty one in order to complete all computations. If a replacement unit is available, the system must provide the necessary data and control signals to the replacement so that it can perform the original operations executed by the faulty unit. On the other hand, if all existing hardware units are busy, resource conflicts force some operations to be delayed until a replacement unit becomes available. Accordingly, hardware rebinding and operation rescheduling need to be performed, which complicate the designs of controllers and interconnects. In order to reduce the hardware complexity and cost, we present a novel solution, the **L/U reconfiguration** scheme.

Figure 1(a) shows a computation schedule with arbitrary functional unit binding. Each column in the diagram represents one functional unit, and each row depicts one clock cycle. In figure 1(a), there are five identical functional units executing eighteen operations in four clock cycles. Hardware utilization is full in the second and the fourth cycle but not in the other two. The staircase line drawn at the middle of the diagram is defined as the **L/U band partition line**. All operations in the upper "triangle" above the partition line belong to the **U band**, and similarly, operations lying in the lower "triangle" belong to the **L band**. The rectangle containing both bands is defined as a **canonical L/U block**.

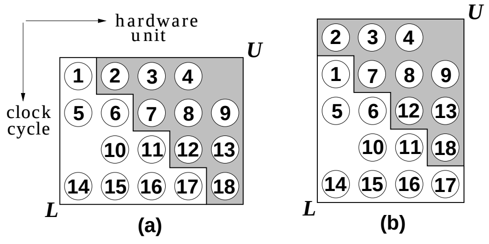

Figure 1: A canonical L/U block (a) before reconfiguration, (b) after reconfiguration.

When a faulty functional unit is detected, both operation rescheduling (software reconfiguration) and functional unit rebinding (hardware reconfiguration) must be executed. The proposed scheme accomplishes both tasks in a systematic fashion as shown in figure 1(b). The U band remains stationary, while the L band can be imagined as being first shifted down by one clock cycle and then shifted right by one unit. The final canonical block presents a new scheduling and binding scheme using one less functional unit but one extra clock cycle.

Figure 1(b) only shows the **logical rebinding** which indicates what operations are executed by each hardware unit. Figure 2 shows the **physical rebinding** of operations. In this figure, all L operations have already been delayed by one clock cycle. The arrows show the required shifting directions of all operations in order to bypass a faulty unit H_i. Incoming data of each functional unit are routed to either its right or left topological neighbor, or passed directly to the unit without shifting. The rebinding rules governing these horizontal movements are given below, where unit H_i is assumed to be faulty.

1. A U operation originally bound to H_j, $j \leq i$, will be shifted left, i.e. rebound to H_{j-1}.
2. An L operation originally bound to H_j, $j \geq i$, will be shifted right, i.e. rebound to H_{j+1}.
3. The remaining operations retain their original binding.

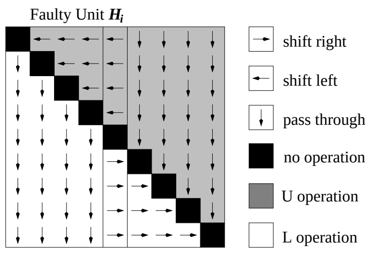

Figure 2: Rebinding directions of operations.

As shown in figure 1, the original $N \times (N-1)$ L/U block reconfigures into the new $(N-1) \times N$ block, where N is the number of intact functional units on chip before reconfiguration. These rigid dimensions restrict the use of the canonical L/U block to small DFGs. A canonical L/U block can be filled up with $N(N-1)$ operations to yield 100% utilization rate under both configurations. In practice, this may not be always possible due to the lack of parallelism in the DFGs. Using this intrinsic underutilization characteristic of most DFGs, we can extend the canonical block format to tackle long DFGs without actual loss of hardware utilization.

2.2 Extension of Canonical L/U Block

An $N \times (N-1)$ canonical L/U block becomes inadequate when a longer DFG is encountered. Figure 3 shows a long computation process that utilizes relatively small number of functional units. In this case, the tall rectangle is partitioned into several L/U blocks. The division line between any two blocks is called **L/U block cut.** The reconfiguration scheme given in figure 1 is applied to individual L/U blocks, and the new schedule after reconfiguration is shown in figure 3.

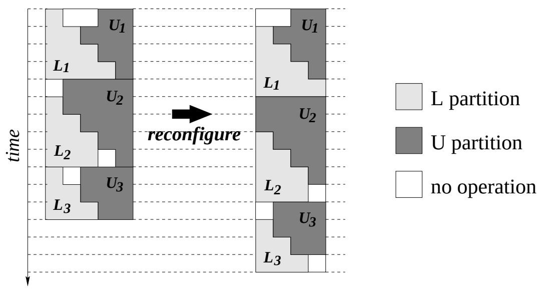

Figure 3: The L/U block cutting scheme.

As shown in figure 3, each L/U block contributes a performance penalty of one extra clock cycle after reconfiguration. To minimize this performance degradation, the whole computational process should be divided into as few L/U blocks as possible. In other words, taller blocks are desirable. To obtain taller blocks, space can be appended above or under the canonical blocks. The original canonical L/U block is now embedded into a general L/U block as the **body**. However, unlike a canonical L/U block, the vertical dimension of the body part in a general L/U block can be less than $(N-1)$. The appended space above the body is defined as **head**, and the space below as **tail**. Figure 4 shows the format of a legitimate L/U block with maximal hardware utilization.

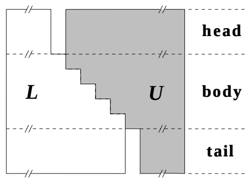

Figure 4: A legitimate L/U block with maximum utilization.

To avoid resource conflicts after reconfiguration, one functional unit per clock cycle within the head and tail sections must be idle. This subsequent under-utilization can be masked by matching it to the inherent parallelism deficit in the DFG. By mapping the fully parallel portions of the DFG into bodies and the under-utilized portions into heads and tails, the effective hardware under-utilization, necessitated by graceful degradation, is minimized.

2.3 Design Constraints

The L/U reconfiguration scheme places two types of design constraints to the synthesis procedure. These constraints only govern individual L/U blocks, and no constraints are needed for regulating the inter-relationships among different blocks. The first type of constraints is the **band-delay constraints**. Since data dependence exists among the operations of a DFG, direct application of L/U reconfiguration may result in data hazards. Delaying an L operation by one clock cycle after reconfiguration, may result in its concurrent execution with its data-dependent operations in the U block. Incorrect data may consequently be provided to the U operations. Such data-dependency hazards are called **band-delay hazards**, and the constraints added to eliminate these hazards are thus named band-delay constraints.

The possible type of constraints, called **geometrical constraints**, is added to resolve resource conflicts after reconfiguration. These constraints regulate the maximum hardware utilizations throughout an L/U block. In particular, the geometrical constraints ensure that the number of operations scheduled at each clock cycle does not exceed the number of available functional units, either before or after reconfiguration.

The following five rules outline all synthesis constraints introduced by L/U reconfiguration. The first rule inserts band-delay constraints, while the last four rules correspond to geometrical constraints. The notation $|\beta_i|$ denotes the total number of operations scheduled in band $\beta, \beta \in \{L, U\}$, at clock cycle i.

1. If a U operation depends on the output of an L operation which is executed at clock cycle *i*, the U operation must be scheduled no earlier than clock cycle *i+2*.
2. If $\exists\, i \in \{i : |L_i| + |U_i| = N\}$, a *staircase* L/U band partition line must be included and span vertically at least from clock cycle $j = min(i)$ to $k = max(i), \forall\, i$. The section of an L/U block with a staircase partition line is defined as the $body$. The space above the $body$ is defined as the $head$, and the one below as the $tail$.
3. (a) $|L_{i+1}| = |L_i| + 1,\ \forall\, i \in body$
   (b) $|U_{i+1}| = |U_i| - 1,\ \forall\, i \in body$
4. (a) $|L_i| < |L_{beginning\,of\,body}|,\ \forall\, i \in head.$
   (b) $|U_i| \leq |U_{beginning\,of\,body}|,\ \forall\, i \in head.$
5. (a) $|U_i| < |U_{end\,of\,body}|,\ \forall\, i \in tail.$
   (b) $|L_i| \leq |L_{end\,of\,body}|,\ \forall\, i \in tail.$

2.4 Reconfiguration for Heterogeneous Hardware Classes

When multiple classes of functional units are present, the L/U reconfiguration scheme can be directly applied to each homogeneous hardware class. In the current implementation, a **homogeneous L/U block cutting** approach is accompanied with a **synchronous rescheduling** approach. All hardware classes have their L/U block cuts at the same set of clock cycles, and rescheduling is simultaneously performed in all classes even though only one class contains a faulty unit. Figure 5 shows the resulting scheme when these two approaches are used. Although figure 5 shows rebinding on both hardware classes, rebinding needs to be performed only in the class with the faulty unit. When a second fault hits another hardware class, only rebinding is executed in that class. No further rescheduling is required. We can thus conclude that the L/U reconfiguration scheme can tolerate one fault per hardware class.

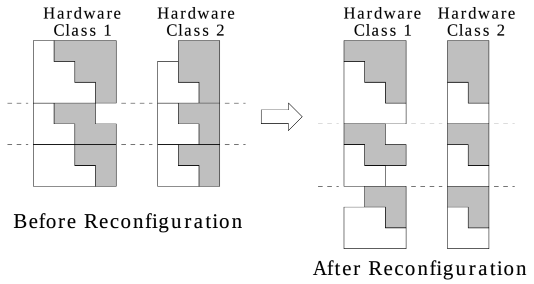

Figure 5: Homogeneous block cutting and synchronous rescheduling.

The homogeneous block cutting and the synchronous rescheduling approach reduce *both* the hardware and performance overheads. First, the proposed approaches reduce controller overhead by sharing the same core finite state machine [1] for various classes. Second, except for the band-delay constraints within each homogeneous hardware class, the proposed approaches do not impose any additional design constraints across L/U blocks and hardware classes.

3 Synthesis Procedure

High-level synthesis (HLS) procedures typically consist of three major steps: hardware allocation, operation scheduling and hardware binding. This traditional three-step procedure is, however, insufficient for handling the complex algorithmic information required to produce reconfigurable designs. In our HLS procedure for generating L/U reconfigurable ASICs, functional unit allocation is specified by the user as an input constraint. Then, the algorithm performs L/U partitioning, scheduling and block cutting one block at a time. After successful L/U block scheduling, hardware binding can be performed to complete the synthesis procedure.

3.1 L/U Block Scheduling

The L/U block scheduling algorithm performs three tasks: (1) divide all operations into L and U bands, (2) schedule all operations while conforming to band-delay constraints, and (3) determine homogeneous block cuts for all hardware classes. Figure 6 shows the iterative L/U scheduling algorithm, which cuts, partitions and schedules L/U blocks for all hardware classes simultaneously in a top-down fashion.

At first, a tentative block height is estimated, and all operations that can be potentially scheduled within this block are identified. The next step partitions these operations into L and U by using the Kernighan-Lin algorithm [4] with two partitioning criteria: (i) the L-to-U dataflow is minimized so that the number of band-delay constraints can be reduced; (ii) the L and U bands are forced into their respective triangular shapes and sizes by using scheduling probability distributions [9].

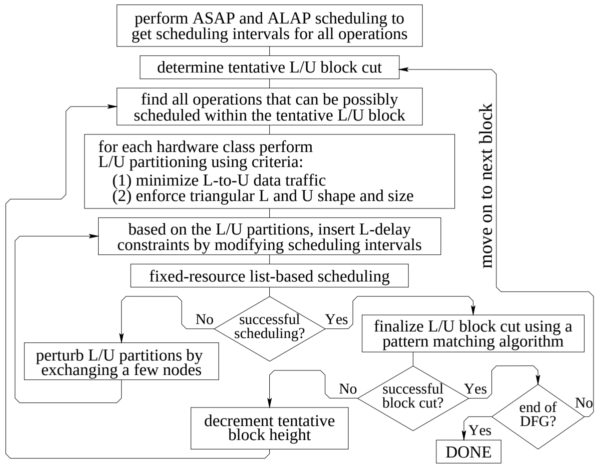
perform ASAP and ALAP scheduling to get scheduling intervals for all operations
determine tentative L/U block cut
find all operations that can be possibly scheduled within the tentative L/U block
for each hardware class perform L/U partitioning using criteria:
(1) minimize L-to-U data traffic
(2) enforce triangular L and U shape and size
move on to next block
based on the L/U partitions, insert L-delay constraints by modifying scheduling intervals
fixed-resource list-based scheduling
No
successful scheduling?
Yes
finalize L/U block cut using a pattern matching algorithm
perturb L/U partitions by exchanging a few nodes
No
successful block cut?
Yes
end of DFG?
No
Yes
decrement tentative block height
DONE

Figure 6: The L/U block scheduling algorithm.

After successful L/U partitioning, band-delay constraints are inserted between operations with L-to-U data dependence by modifying their scheduling intervals. The initial scheduling intervals are obtained from ASAP and ALAP scheduling. Band-delay constraints are enforced by setting the scheduling interval of the U operation to start and to terminate two clock cycles behind the scheduling interval of the L operation.

The next step schedules all operations within the tentative block height based on the modified scheduling intervals. The list-based algorithm is chosen [9] to perform a resource-constrained scheduling. One modification to the original list-based algorithm is to check on any band-delay constraint violation while preparing the ready operation list. If scheduling fails, the algorithm backtracks by perturbing the existing L/U partitioning (exchanging a few nodes), inserting the subsequent band-delay constraints, and performing another scheduling.

A successful schedule may exceed the tentative L/U block height. L/U block scheduling for different hardware classes may terminate at disparate clock cycles. A final homogeneous block cut must be determined. This task is performed by a pattern matching algorithm which searches for a legitimate L/U block format with maximal block height for the successful schedule. Operations excluded from the current L/U blocks (one per hardware class) will be reconsidered by the subsequent blocks. If no successful block cut is secured, the algorithm backtracks to L/U partitioning.

4 Experimental Results

We have randomly generated several DFGs to demonstrate the performance and hardware efficiency of the L/U reconfiguration scheme. Two algorithmic DFGs are also included: the elliptic filter (EF) with homogeneous hardware classes (ALUs) and a 3×3 Fast Fourier Transform (FFT). All experimental results (performance comparisons only) are given in table 1, and the final L/U partitioning and schedule of the FFT example are given in figure 7.

To measure the performance overheads of the L/U reconfiguration scheme, two types of comparisons are made. First, the speed performance of reconfigurable designs is compared against optimal non-fault-tolerant (NFT) designs. As compared to the **optimal** NFT designs (manually constructed), pre-reconfiguration performance overhead of the reconfigurable designs is 0.0%. Post-reconfiguration performance of the reconfigurable designs (also compared to the optimal NFT designs) has on average an 8.2% overhead. Pre-reconfiguration comparisons are made between the reconfigurable designs (column 10) and the optimal NFT designs with comparable hardware resources (column 6). Similary, post-reconfiguration comparisons are made between the reconfigurable designs (column 11) and the optimal NFT designs with a single hardware unit from various classes removed (column 7 to 9).

In the second type of comparisons, the pre- (column 10) and post-reconfiguration (column 11) performances of the reconfigurable designs are compared to yield the degradation percentages (column 12). On average, this performance degradation is 30.2%. This degradation is, however, mainly contributed by the reduction in hardware resources.

Figure 7(a) shows the DFG performing the 2-D 3×3 FFT. The L/U band partition and the pre-reconfiguration schedule are shown in Figure 7(b). To complete all operations in five clock cycles (critical-path length), eight adders, six

DFG	No.	resource			Number of clock cycles							Performance overhead %			
		alloc.			Non-fault-tolerant				L/U Reconfig.			w.r.t. non-fault-tol.			
	nodes	+	×	-	full	rm+	rm×	rm-	full	rec.	%deg.	full	rm+	rm×	rm-
DFG1	15	3	–	–	6	8	–	–	6	8	33.3	0.0	0.0	–	–
DFG2	21	5	–	–	6	7	–	–	6	7	16.7	0.0	0.0	–	–
DFG3	27	3	–	–	10	14	–	–	10	14	40.0	0.0	0.0	–	–
DFG4	33	3	3	–	7	8	10	–	7	10	42.9	0.0	25.0	0.0	–
DFG5	40	4	3	–	10	11	12	–	10	13	30.0	0.0	18.2	8.3	–
DFG6	39	3	3	3	5	7	7	6	5	8	60.0	0.0	14.3	14.3	33.3
DFG7	47	3	3	3	9	10	10	9	9	11	22.2	0.0	10.0	10.0	22.2
EF	34	4 ALUs			14	15 (with 3 ALUs)			14	15	7.1	0.0	0.0	0.0	0.0
FFT	49	8	6	5	5	6	6	6	5	6	20.0	0.0	0.0	0.0	0.0
										Average	30.2	0.0	8.2		

Table 1: Experimental results.

multipliers and five subtracters are allocated. The whole computation process is included in one level of L/U blocks that results consequently in a post-reconfiguration performance overhead of one clock cycle only.

5 Conclusion

A novel graceful degradation scheme and an associated high-level synthesis methodology have been presented. Our approach can be applied to general ASICs with multiple hardware classes and irregular datapath structures. Small performance degradations are achieved, in conjunction with (1) the synthesis algorithm and (2) the novel reconfiguration scheme, due to aggressive exploitation of under-utilizations embedded in DFGs. The advantages of the proposed graceful degradation technique include superior pre- and post-reconfiguration hardware utilization, instant reconfigurability, and small hardware and performance overheads.

References

[1] W. Chan and A. Orailoğlu. High-Level Synthesis of Gracefully Degradable ASICs. Technical Report CS95-447, Univ. of California, San Diego, Dept. of CSE, September 1995.

[2] M. Chean and J. A.B. Fortes. A Taxonomy of Reconfiguration Techniques for Fault-Tolerant Processor Arrays. *IEEE Computer*, 23(1):55–66, January 1990.

[3] B. Iyer, R. Karri, and I. Koren. Phantom Redundancy: A High-Level Synthesis Approach for Manufacturability. In *Proceedings of ICCAD*, pages 658–661, November 1995.

[4] K. H. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graph. *Bell System Technical Journal*, 49(2):291–307, February 1970.

[5] R. Naidu and S. Mahapatra. Fault Tolerance in N-MOS Random Access Memories with Dynamic Redundancy Methods. *Microelectronics and Reliability*, 28(2):193–200, 1988.

[6] R. Negrini, M. G. Sami, and R. Stefanelli. *Fault Tolerance Through Reconfiguration in VLSI and WSI Arrays*. MIT Press, 1989.

[7] A. Orailoğlu and R. Karri. Coactive Scheduling and Checkpoint Determination during High-Level Synthesis of Self-Recovering Microarchitectures. *IEEE Transactions on VLSI Systems*, 2(3):304–311, September 1994.

[8] A. Orailoğlu and R. Karri. Automatic Synthesis of Self-Recovering VLSI Systems. *IEEE Transactions on Computers*, February 1996.

[9] P. G. Paulin and J. P. Knight. Algorithms for high-level synthesis. *IEEE Design and Test of Computers*, 6(6):18–31, December 1989.

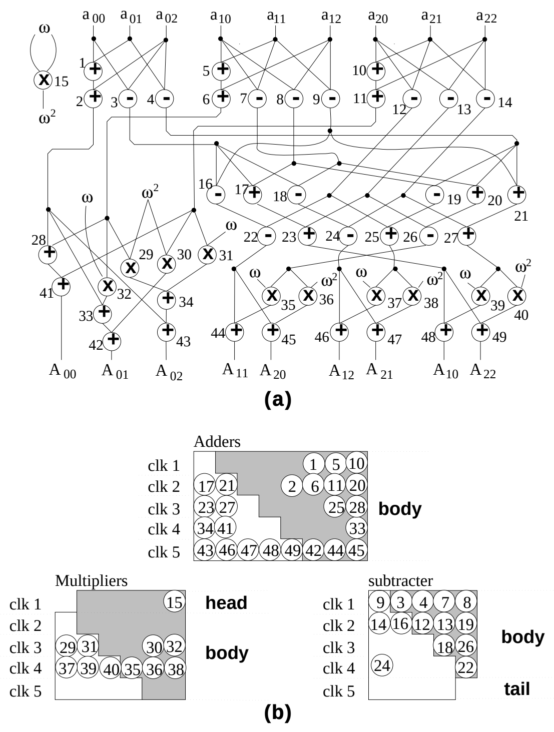

Figure 7: The 3×3 FFT example: (a) The dataflow graph. (b) The L/U partition and schedule before reconfiguration.