HIGH LEVEL SYNTHESIS FOR RECONFIGURABLE DATAPATH STRUCTURES

Lisa Guerra, Miodrag Potkonjak[†], Jan Rabaey

Dept. of EECS, University of California at Berkeley, CA
[†]C&C Research Laboratories, NEC USA, Princeton, NJ

Abstract

High level synthesis techniques for the synthesis of restructurable datapaths are introduced. The techniques can be used in applications such as design for fault tolerance against permanent faults, design for yield improvement, and design of Application Specific Programmable Processors. This paper focuses on design techniques for Built in Self Repair (BISR), which addresses the first two of these applications. The new BISR methodology consists of two approaches which exploit the design space exploration abilities of high level synthesis. The first method uses resource allocation, assignment, and scheduling, and the second uses transformations. The effectiveness of the approaches are verified on a set of benchmark examples.

1 Introduction

Built in Self Repair (BISR) is a fault tolerance technique in which spare modules are provided to supplement the core operational modules. Using BISR techniques, yield can be improved by replacing defective modules with spares before packaging. Alternatively, reliability can be improved by automatic replacement of failed modules with spare ones, so that the overall system can continue to function correctly. This is especially important for military systems or space exploration missions, where it is critical that there are no system failures.

BISR techniques are regularly used in memories and bit sliced designs [1]. So far they have not received much attention in ASIC design, but with higher levels of integration they will soon be important for improving yield and/or reliability in this area as well.

High level synthesis techniques have previously been used to address a variety of design goals [2], but little work has been done on techniques for fault tolerant design. While previous high level synthesis methods have addressed intermittent and transient faults [3, 4], this work concentrates on permanent faults.

The main concepts introduced are based on exploiting the flexibility provided by high level synthesis during design space exploration. Although this paper focuses on BISR design, the techniques introduced have a high potential to facilitate the synthesis of Application Specific Programmable Processor (ASPP) datapaths. Intelligent strategies to use the flexibility of solutions is a crucial component for achieving small overhead designs of reconfigurable datapaths in ASPP design. Most often, hardware overhead is minimized by identifying a set of configurations which are similar in terms of the required hardware resources. Consider, for example, the design of an ASPP to implement the 2 different computations A and B. Let Ai and Bj represent particular implementation solutions for the computations A and B. The ASPP implementation is the union of the hardware, $Ai \cup Bj$, for any i and j. The goal is not to find the $Min(Ai) \cup Min(Bj)$ implementation, but to find the $Min(Ai \cup Bj)$ solution, which often is one for which Ai and Bj have similar hardware requirements.

Before proceeding, a number of assumptions are presented. In this paper the hardware model [5] shown in Figure 1 will be used. To stress the importance of interconnect minimization, the model clusters all registers in register files, connected only to the inputs of the corresponding execution units. We assume that there is no bus merging.

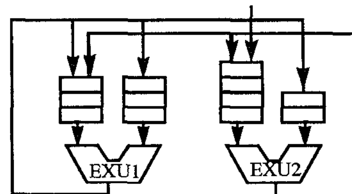

Fig. 1. Hardware Model: Interconnect-Regfile-EXU

Faults can occur in either an execution unit, a register file,

1063-6757/93 $03.00 © 1993 IEEE

or a bus. Under the given hardware model, all faults can be classified as execution unit faults. A faulty register file prevents its corresponding execution unit from receiving data, and thus has the same affect as a fault in the execution unit. Similarly, a faulty bus can be treated as a failure in the execution unit at its receiving end. Other high level synthesis hardware models, can be addressed using the methodology presented here with proper modification of the algorithms.

Within this framework, the high level synthesis BISR process can be defined as follows: **Given an ASIC computation, an underlying hardware model and an execution time bound t_{avail}, synthesize a minimum area design, so that up to K hardware units can be faulty.**

If these techniques are used for fault tolerance against permanent faults, it is assumed that an error checking mechanism exists. If the methods are instead used for yield enhancement, it is assumed that manufacturing testing will detect the faulty units. In either case, the controller is reconfigured upon detection of a fault. The controller is assumed to be either reprogrammable or to lie on a separate chip.

2 BISR Algorithms for ASIC Design

The most straightforward approach to BISR is to provide a spare for each hardware instance, resulting in full duplication of the hardware. Fortunately, the BISR overhead need not be so high. If the number of faulty units, K, is 1, for example, the assignment step provides the flexibility under which it is clear that only 1 spare for each hardware class is necessary. The operations from the failed unit will be assigned to the spare of the same type.

Considering the additional flexibility brought by scheduling, however, even fewer spares can often be used. When a failed unit is detected, instead of reassigning only those operations implemented by the failed unit, a complete reassignment and rescheduling of all operations of the computation is performed.

The example of Figure 2 illustrates how BISR overhead can be greatly reduced by exploiting scheduling flexibility brought by one type of spare unit to alleviate the need for another type of redundant unit. For this and all subsequent examples, assume that each operation takes 1 control cycle, and K=1. The minimum required hardware consists of 1 adder and 2 multipliers, or 2 adders and 1 multiplier. If scheduling flexibility is not exploited, the minimum BISR hardware will be 2 adders and 3 multipliers, or 3 adders and 2 multipliers. However, if we allocate only 2 adders and 2 multipliers, a complete BISR implementation

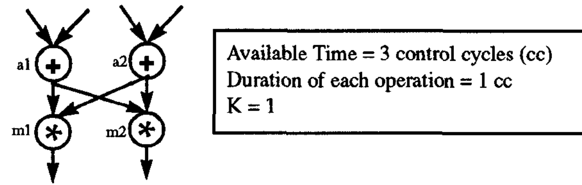

Fig. 2. Motivational Example

can still be achieved by altering the schedule. This can be verified by the schedules given in Table 1.

Control Step	Failed Unit			
	Adder (A)		Multiplier (M)	
	A	M	A	M
1	a1	-	a1,a2	-
2	a2	-	-	m1
3	-	m1,m2	-	m2

Table 1: Schedules when Different Units Fail

2.1 Allocation, Assignment, Scheduling for BISR

As a starting point for the BISR algorithms, we used tools from the HYPER high level synthesis system [5] for allocation, scheduling and assignment. For BISR it was necessary to develop a new allocation scheme.

Before the explanation of the new allocation algorithm, several definitions are presented. An allocation, A, is a proposed set of hardware units for the BISR realization of an application algorithm. For a given K, there are many possible combinations of K units which can fail. Let i represent one such failure event. The child allocation A(i), $A(i) \subset A$ is the effective allocation of hardware (allocation of good units) for the failure event i. Note that the number of elements $|A - A(i)|$ is equal to K. A feasible allocation, A, is thus one for which successful reassignment and scheduling can be accomplished for all of its child allocations A(i).

The basic idea of the allocation mechanism is to start with an initial allocation, add hardware until a feasible allocation is found, then remove all unnecessary redundant hardware. The pseudo-code for the global flow of the algorithm is shown below.

A sharp minimum bound, M_j, on the necessary amount of hardware of each class j is used as the initial allocation. M_j is defined as $M_j = m_j + K$ where m_j is a minimum bound on the amount of hardware *j* necessary for any non-BISR implementation and K is the number of faults. For each hardware class, *j*, relaxation based scheduling techniques [6] are used to derive m_j. The equation for M_j can

```
GetInitialAllocation();
While (!Success) {
   SortInDecreasingOrderOfStress();
   Success = Assign& ScheduleWithFailedUnit();
   UpdateStress();
    if (!Success)
        GetNewHWUnit();
}
RedundancyRemovalWithLookAheadPruning();
```

be understood by observing that *any* implementation requires at least m_j units, and since up to K units of type j can be bad, at least (m_j + K) units are needed.

If the initial allocation fails, the expansion phase is entered. In this phase, new hardware units are added one by one until the allocation succeeds. Good selection methods have a crucial impact on the speed of the algorithm and the quality of the solution. The primary criteria for deciding which hardware type to add next involves using a measure called the *global stress* of a hardware resource class. The global stress is composed of three measures: Minimum Bounds Stress, ε-Critical network Stress, and Scheduling Stress [7].

At the completion of the expansion phase, there is no guarantee that the feasible allocation is minimal. It is possible that a subset of the allocation, $A' \subset A$ is also a solution. To assure that a local minimum has been reached, it is necessary to assure that if any units are removed from the current solution, the solution is no longer valid. In general, the units with minimum stress are tried first. It is also imperative, however, to incorporate a remember-and-look-ahead technique, so that time is not wasted attempting allocations that will definitely fail. We remember all allocations and child allocations that failed, and use this information whenever considering an allocation A'. Before attempting A', a look-ahead to its child allocations will determine if there is any overlap between the children of A' and any known allocations that have failed.

For a successful allocation, a feasible schedule for each child allocation must be found. The schedules are ordered in decreasing order of difficulty, so that we can exit as fast as possible in the event that there is an insufficient allocation. This ordering is based on the global stress function described earlier.

2.2 Transformation-Based BISR

Sometimes, it is not possible to reduce BISR overhead using the allocation-based techniques of Section 2. In that case, transformations can be used to reduce the overhead. Transformations are alterations in the computational structure such that the behavior is maintained [2]. The key new idea is to transform the computation in different ways according to the needs imposed by the available hardware, for each possible scenario of failed units. The example in Figure 3 illustrates this idea. The following identity is used to transform 3a into 3b:

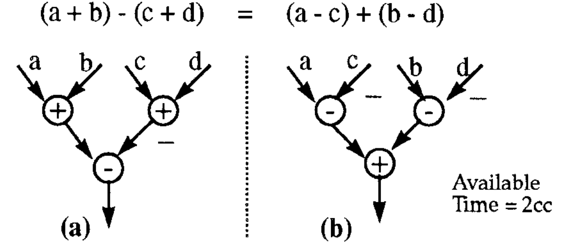

Fig. 3. Transformations for BISR: Motivational Example

All operations lie on the critical path, so it is not possible to reduce BISR overhead using the techniques of Section 2. If only computation 3a is considered, then 3 adders and 2 subtractors are needed for the BISR implementation. However, if both implementations are considered, only 2 subtractors and 2 adders are needed. If the subtractor fails, we can use computation 3a which needs 2 adders and 1 subtractor, and when the adder fails we can use computation 3b which needs 2 subtractors and 1 adder.

Figure 4 shows how retiming can be used for high level synthesis BISR. Retiming reshuffles the operation overlaps in such a way that operations which require units that

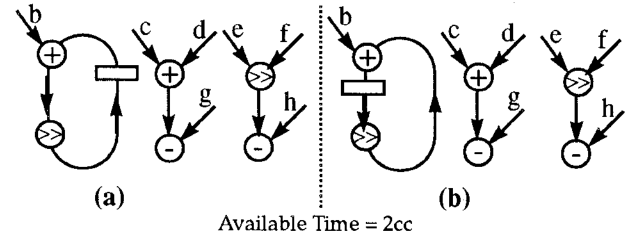

Fig. 4. Retiming for BISR

are in short supply (due to failure) are no longer bound to happen in the same control step. It is easy to determine by analyzing the various schedules, that 3 subtractors, 2 adders and 2 shifters are sufficient for the final BISR implementation.

In general, transformations to reduce BISR overhead can be classified into two groups:

1. Transformations to increase the resource utilization (and therefore reduce the need) of the units of the same type as the failed EXU,
2. Transformations to reduce the number of operations of the same type as the failed resources.

A detailed explanation of how various transformations are used for BISR design is presented in [7].

Associativity, inverse element law, commutativity, and retiming are used in the transformation-based BISR design. Note that HYPER simulation tools can be used to verify that relevant numerical properties (e.g. numerical stability and overflow control) are maintained in all transformed designs. A probabilistic sampling algorithm [5] is used as the core transformation engine. Weighting in the cost function is affected by the number of resources available. The algorithm is thus driven to apply transformations that alleviate the need for resources that are in short supply. A key novelty is that transformations are tried for the various scenarios in decreasing order of estimated difficulty, as determined by the stress function of the previous section.

3 Experimental Results

The BISR techniques were tested on the set of examples shown in Table 2. The results presented here assume the number of faulty units, K, is equal to 1. The average and median BISR design area overhead over all examples was 17% and 16.6% respectively. The initial implementations had an average of 3.75 different types of hardware units, an average of only 2.3 additional units were needed for the BISR designs. Table 3 shows several examples designed using the transformation-based methods of BISR design. The average area increase is only 10.4%, and an average of 1.5 out of 3.75 additional hardware units were needed.

Example	Non-BISR #Units	BISR #Units	# Hw Classes	Non-BISR Area	BISR Area	Area Overhead (%)
WFFT8	6	8	3	1.79	2.49	39.0
5EllipWDF	6	9	4	1.43	1.73	21.0
8IIR GM	8	9	4	4.84	4.95	2.3
WConv8	8	9	3	1.91	2.38	24.6
8IIRParallel	9	12	4	2.23	2.55	14.4
8IIRCascade	9	12	4	4.24	4.69	10.6
DCT	10	12	3	1.40	1.77	26.4
5IIR	11	14	4	4.55	5.56	22.2
7IIR	17	19	4	4.47	4.92	3.1
8IIR DF	23	26	4	19.81	21.20	7.0
3StateLinCn	29	32	4	8.22	9.39	14.0
Wavelet	30	32	4	22.05	26.19	18.8

Table 2: Results for Allocation-based designs

The consequences of the BISR techniques on yield and chip productivity are calculated, using the procedure presented by Stapper [8]. An initial yield of 10% was assumed. The BISR designs had average yield and productivity improvements of 87.4% and 61.0% respectively [7].

Example	Non-BISR #Units	BISR #Units	# Hw Classes	Non-BISR Area	BISR Area	Area Overhead (%)
11FIR	8	9	4	5.45	6.5	19.3
7IIR	7	9	4	9.27	9.92	7.0
35FIR	7	8	4	12.31	13.34	8.4
5StateLinCn	21	23	3	36.00	38.49	6.9

Table 3: Results for Transformation-based designs

4 Conclusions

New techniques to compose a reconfigurable heterogeneous BISR implementation have been presented. The approach is based on the flexibility of high level synthesis during design space exploration. The experimental results and high yield and productivity improvements demonstrate the potential of this approach.

References

[1] D.P. Siewiorek, R.S. Swartz, *Reliable Computer Systems: Design and Evaluation*, 2nd edition, Digital Press, Burlington, MA.

[2] M.C. McFarland, A.C. Parker, R. Camposano, "The High-Level Synthesis of Digital Systems," *Proceedings of the IEEE*, Vol. 78, No. 2, pp. 301-317, 1990.

[3] V. Raghavendra, C. Lursinsap, "Automated Micro-Roll-Back Self Recovery Synthesis," *DAC* , pp. 385-390, 1991.

[4] R. Karri and A. Orailoglu, "Transformation-Based High-Level Synthesis of Fault-Tolerant ASICs," *DAC*, pp. 662-665, 1992.

[5] J. Rabaey et al., "Fast Prototyping of Data Path Intensive Architectures," *IEEE Design & Test Magazine*, June 1991.

[6] J. Rabaey, M. Potkonjak, "Complexity Estimation for Real Time Application Specific Circuits," *ESSCIRC*, pp. 201-204, 1991.

[7] L. Guerra, M. Potkonjak, J. Rabaey, "High Level Synthesis for Reconfigurable Datapath Structures," *Technical Report 93-C107-4-5510-9*, NEC USA, Princeton, NJ, 1993.

[8] C. H. Stapper, "A New Statistical Approach for Fault-Tolerant VLSI Systems," *FTCS*, pp. 356-365, Boston, MA, 1992.