Behavioral-Level Synthesis of Heterogeneous BISR Reconfigurable ASICs[1]

Lisa Guerra[†], Miodrag Potkonjak[‡], and Jan Rabaey[†]

[†]***University of California - Berkeley, EECS Department, Berkeley, CA 94720***
[‡]***University of California - Los Angeles, CS Department, Los Angeles, CA 90095***

Keywords: Behavioral-Level Synthesis, Built-In-Self Repair, Fault Tolerance, Reconfigurable ASIC, Transformational Synthesis.

Abstract

Behavioral-level synthesis techniques have traditionally focussed on design of a fully-hard-wired application-specific implementation of a given computation. In this paper, new techniques are presented for the synthesis of *reconfigurable* hardware. The technique is applicable for synthesis of several classes of designs, including: 1) design for fault tolerance against permanent faults, 2) design for improved manufacturability, and 3) design of Application Specific Programmable Processors (ASPPs) — processors designed to perform any computation from a specified set on a single implementation platform. This paper focuses on design techniques for efficient Built-In-Self-Repair (BISR), and thus directly addresses the former two applications.

Previous BISR techniques have been based on replacing a failed module with a backup of the same type. We present new heterogeneous BISR methodologies which remove this constraint and enable replacement of a module with a spare of a different type. The approach is based on the flexibility of behavioral-level synthesis to explore the design space. Two behavioral synthesis techniques are developed; the first method is through assignment and scheduling, and the second utilizes transformations. Experimental results verify the effectiveness of the approaches. In particular, significant yield and productivity improvements, calculated using state-of-the art yield modeling techniques, show the power of these techniques for improving manufacturability.

1. Preliminary versions of this work appeared in the *Proc. IEEE/ACM ICCAD*, November 1993 and in the 1*993 Proc. of the IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems,* October 1993.

1.0 Introduction

With the rising cost of semiconductor manufacturing and the increasing complexity of integrated circuits, improvement of manufacturing yields is crucial for achieving economic utilization of semiconductor manufacturing facilities. Fault tolerance techniques such as the Built-In-Self-Repair (BISR) sparing methodology will play an important role in achieving yield improvements. BISR is a hybrid redundancy technique in which a set of spare modules are provided in addition to the core operational modules [Sie92]. If an implementation is found to have defective core modules, these modules can be replaced by functional spare ones before packaging.

BISR methods can be applied not only for yield improvement but also for improvement of design reliability. Designs can be made fault tolerant to failures occurring during operation by automatic replacement of failed modules with spare ones, so that the overall system can continue to function correctly. This is especially important in military systems and space exploration missions [Avi71] where it is critical that there are no system failures, or where manual replacement of failed modules is either impossible or prohibitively expensive.

BISR techniques are regularly used in primary and secondary memories [Moo86, Sie92, Pat88] and sometimes in general purpose bit-sliced execution units [Sie92]. While they have hitherto received little attention in application-specific integrated circuit (ASIC) design, the ever increasing level of integration should soon make them an important methodology for ASIC yield and productivity improvements.

This paper introduces a novel concept of Heterogeneous BISR for ASIC designs. As opposed to traditional BISR, where a failed module is replaced by a spare of the same type, we propose Heterogeneous BISR (HBISR), which enables replacement of a module with a spare of a different type. HBISR leverages on a computation's *flexibility*, or its ability to be implemented in a number of competitive ways, to increase productivity and fault tolerance. In addition, behavioral-level synthesis techniques for exploiting this flexibility in automated design are presented. One method is through assignment and scheduling, and the second utilizes transformations.

The paper is organized in the following way. After a survey of previous works and discussion of several design preliminaries, simple, yet real-life examples are used to introduce the main ideas. Next, new behavioral-level synthesis algorithms which minimize hardware overhead for HBISR during resource allocation, assignment, scheduling and transformations are discussed. The paper concludes by presenting and analyzing experimental results on a variety of real-life DSP examples and briefly outlining extensions and directions for future research.

2.0 Previous Work

The main target for BISR techniques are systems that are bit-, byte-, or digit- sliced. By far the most important use of bit-sliced BISR is in SRAM and DRAM circuits [Moo86, Sch78, Smi81, Man82, Sar84, Aru80]. Almost all current day memory designs use BISR techniques [Kik91, Tan92], as they significantly increase memory production profitability. Programmable logic arrays are another class of bit-sliced devices for which BISR has been well addressed [Gre84, Kor85, Wey87, Has90]. It has also been successfully applied for arithmetic-logic unit byte slices [Sie92, Lev68]. Other areas in which BISR techniques are being used include secondary storage systems [Pat88], massive parallelism [Gri91], wafer scale integration [Sat92], and systolic array designs [Lei85, Neg89]. Note that BISR methodology is not limited to memory and execution units; Lewis, for example, proposed the use of a backup fault tolerant clock [Lew79].

While all previous BISR techniques have been based on replacing a failed module with a backup of the same type, we present new heterogeneous BISR methodologies. Additionally, we present synthesis techniques for their design. Behavioral-level synthesis provides the flexibility of design space exploration so that a variety of design goals can be addressed. An overview and extensive bibliography of behavioral-level synthesis algorithms can be found in [McF90, Wal91, DeM94]. Most of these works, target the optimization of area and speed (throughput). Recently, other important goals, such as power [Cha92], testability [Lee92], and reliability and fault tolerance [Bir89, Bre88, Rag91, Kar92] have been addressed. Little work has been reported on behavioral-level synthesis techniques for reliable and fault tolerant design. Raghavendra and Lursinsap [Rag91] concentrated on designs with self - recovery from transient faults using micro roll-back and checkpoint insertion. Karri and Orailoglu presented scheduling and assignment and transformation-based methods for minimizing hardware overhead while achieving a certain level of fault tolerance in micro roll-back [Kar92, Kar93b] and fault-secure fault-tolerant architectures [Kar93a]. While all previous behavioral-level synthesis methods for enhancing fault tolerance have addressed intermittent and transient faults [Sie92], this work concentrates on permanent faults, where fault tolerance is used for reliability, yield, and productivity enhancement.

3.0 Issues in HBISR Reconfigurable Datapath Design

Given a behavioral description of the algorithm, an underlying hardware model, and a throughput constraint, the goal is to synthesize a minimum area design that can tolerate a number of faulty hardware resources. This section describes our targeted application and computation and

hardware models, as well as several implementation issues in the design of reconfigurable datapaths for HBISR design.

3.1 Computational and Hardware Models

Targeted applications include real-time DSP, video, multi-media, and other numerically intensive algorithms. Applications are represented as hierarchical data-control flowgraphs, with nodes representing the flowgraph operations, and edges the data and control dependencies between them [Rab91]. The underlying model of computation is the homogeneous synchronous data flow model of [Lee87].

The ASIC hardware model being considered is shown in Figure 1. This model, the dedicated

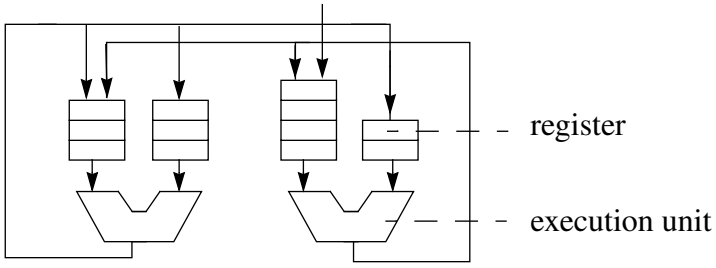

FIG. 1. Dedicated Register File Hardware Model.

register model, clusters all registers in register files, connected only to the inputs of the corresponding execution units [Cla88]. We also assume that there is no bus merging, so there exists a dedicated bus connecting any two units between which there are data transfers. Note that the HBISR methodology itself is not limited to this hardware model; generalizations are discussed in Section 7.0.

3.2 Fault Model, Detection, and Diagnosis

At the gate level, a single stuck-at model [Abr90] is assumed for faults, and at the register transfer level, we assume that a unit is faulty if it has one or more gate level faults. Faults can occur in either an execution unit, register file, or bus. Under the targeted hardware model and these assumptions, 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 effect 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. Note again that the HBISR methodology itself is general, and can be easily extended to other fault models.

A number of testing approaches are available to detect that a fault exists, and to diagnose its location. If BISR is used to improve manufacturability, any off-line testing and diagnosis scheme can be used (i.e. partial-scan sequential automatic test pattern generation (ATPG) [Che90, Lee90, Nie91], full-scan and combinational ATPG [Eic78], Built-In-Self-Test (BIST) [Wil73, Bar87], and insertion-point based-schemes [Hay74]). Note that any scheme which does not have strong diagnosis capabilities (e.g. IDDQ-based testing [Gul93]), cannot be used.

If the BISR methodology supports in-field reconfiguration after failure of particular hardware part(s), a BIST scheme is required. In this case, testing capabilities are "built-in" (resulting in test-hardware overhead) to the chip itself. The BIST scheme can be either on-line or off-line [Abr90]. In the off-line BIST scheme, periodic interruption of the functional mode is required.

3.3 Controller Implementation

Upon diagnosis of a fault, the controller is reconfigured. Several alternatives for efficient low overhead controller implementation include:

1. Programmable Controller [Pat89]: While this type of controller often brings a somewhat large implementation area and a small degradation in performance, it provides flexibility not only for HBISR, but also for relatively minor alterations in the chip functionality as is often required in modern day designs.
2. Off-chip controller: This controller can be replaced as necessary since it is located on a separate chip. A number of high performance datapath intensive chips have used this option [e.g. Yeu92]. The same drawbacks and advantage as in the previous case hold.
3. Composed controller: This controller is located on-chip, and is the composition of all possible control configurations that may be used. Its effectiveness depends on how well several different (but often similar) controllers can be merged.

In all cases we assume that the controller itself is fault-free. This assumption can be easily replaced by a fault-tolerance mechanism which provides resiliency of the controller. Since the controller usually occupies a very small fraction of chip area in datapath-intensive ASIC designs [Rab91], simple replication is often an adequate solution. While interconnect from the controller to the datapath usually takes a somewhat larger percentage of the area, in absolute terms it still results in a rather small overhead. Note that while datapath interconnect usually transfers data of significant word lengths, each control interconnect is a single bit.

4.0 Assignment and Scheduling for HBISR Design

4.1 Key Ideas and Motivational Example

Probably the most straightforward approach to BISR is to provide a spare for each hardware instance, resulting in full duplication of the hardware. With the detection of a faulty unit, reconfiguration takes place to initiate use of its spare. This reconfiguration is conceptually a switch that passes control from the failed to the backup unit.

Fortunately, the BISR overhead need not be so high. If the number of faulty units, K, is 1, for example, the behavioral-level synthesis assignment step provides us with the flexibility under which it is clear that only 1 spare for each hardware class is necessary, as opposed to one spare per hardware instance. The operations from the failed unit will be transferred to a spare of the same type.

The flexibility gained through assignment clearly reduces the amount of hardware redundancy needed. Considering the additional flexibility brought by scheduling, however, we can often use even fewer spares. This is possible since assignment and scheduling enable the replacement of a module by a spare of a different type. When a failed unit is detected, instead of reassigning only those operations of the failed unit, we completely reassign and reschedule all operations of the flowgraph. The specific goal addressed can now be restated as follows*: find the minimum area solution which meets the throughput constraint, for which the algorithm can be reassigned and scheduled, even when as many as K units are faulty.*

The main ideas are illustrated with the following example (Figure 2, Table 1), where the goal is a low-overhead HBISR implementation that can tolerate up to one faulty unit. The example includes the imaginary part of a complex number multiplication with a constant and with a

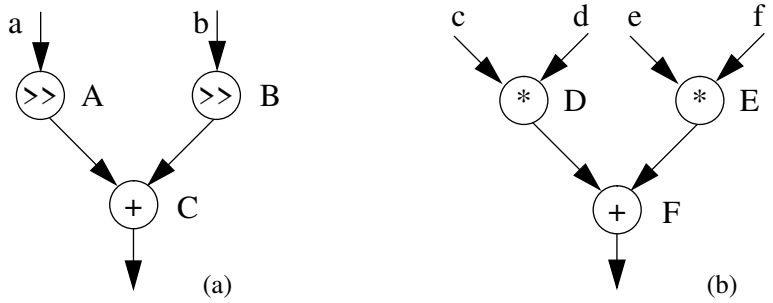

FIG. 2. Scheduling and assignment for HBISR: Imaginary part of complex multiplication: (a) with constant multiplications, (b) with variable multiplications.

variable (multiplication with a constant is such that it can be done using a single shift). Assume that each operation takes one control cycle and the available time is three control cycles. The min-

imum hardware required for non-BISR implementation consists of 2 shifters, 1 multiplier and 1 adder. If scheduling flexibility is not exploited, the minimum BISR hardware will require an additional unit of each type. However, using both assignment and scheduling, only 2 shifters, 2 multipliers, and 2 adders are needed. If the adder fails, operations A and B are done on the 2 shifters in step 1, D and E on the 2 multipliers also in step one, and C and F on the single remaining adder in steps 2 and 3. Table 1 shows the schedules for all three cases of failed units. An important point to note is that no additional shifters are needed. In the event of a shifter failure, the scheduling flexibility brought by the redundant multiplier is exploited.

Control Step	Adder Failure			Shifter Failure			Multiplier Failure		
	>>	*	+	>>	*	+	>>	*	+
1	A, B	D, E		A	D, E		A, B	D	
2			C	B		F		E	C
3			F			C			F

Table 1: Schedules for the example of Fig. 2, to be used when an adder, shifter, or multiplier fails.

4.2 Allocation, Assignment, and Scheduling Algorithm

The global strategy of the Hyper behavioral-level synthesis system [Pot89] is well suited for use as the starting point for developing new algorithms targeting HBISR. In this system, allocation first proposes a hardware solution, then assignment and scheduling are performed to check its feasibility. To take into account BISR requirements, it was necessary to develop a new allocation scheme. A generic assignment and scheduling module can easily be modified for use within this framework. We have used the Hyper scheduler and assigner, with small modifications.

Before explaining the details of the new allocation algorithm, several definitions are presented. An allocation, A, is a proposed set of hardware units for the HBISR 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 at an initial allocation, add hardware until a feasible allocation is found, then remove all unnecessary redundant hardware. Note that for

any proposed allocation solution, it is necessary to assure that scheduling can succeed with **any** combination of K failed units. The basic framework for the allocation algorithm has been set up and implemented for K=1. The pseudo-code for the global flow is presented below:

```
GetInitialAllocation();
while (!Success) {
   SortInDecreasingOrderOfStress(Ordered_HW);
   foreach j  ∈   Ordered_HW {
        Success = Assign and ScheduleWithFailedUnit(j);
        if (!Success)
          break;
   }
   UpdateStress();
   if (!Success) {
         AddHardware();
   }
 }
RedundancyRemoval();
```

The key components of the algorithm are described in the following sections.

4.2.1 Initial Allocation

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 fault-tolerant implementation and K is the number of faults. For each hardware class, *j*, relaxed based scheduling techniques [Rab94] are used to derive an estimate of m_j. The equation for M_j can 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.

4.2.2 Expansion Phase

If the initial allocation fails, the allocation expansion phase is entered, where new hardware units are added (AddHardware routine) one by one until the allocation succeeds. Good selection heuristics have a crucial impact on the speed of the algorithm and the quality of the solution. Firstly, we want to reach the solution as quickly as possible, avoiding the addition of unnecessary units along the way. Secondly, we want to avoid a greedy steepest descent type algorithm which could lead to many suboptimal solutions.

Two modes of addition, stress-based addition and last gasp addition, have been constructed. Stress-based addition uses a measure called the *global stress* of a hardware resource class to

decide which hardware type to add next. This measure is described in Section 4.2.4, and is composed of several heuristic measures of the difficulty of assignment and scheduling of each hardware class. The larger the stress, the more likely it is that type of unit is the cause for the failure of the assignment and scheduling.

For additional robustness, we have also added a last gasp addition phase, similar in concept, but not technique, to the Last Gasp routine of Espresso [Bra84]. This phase is entered if it is found that the stress measure has ceased to give useful feedback. During last gasp addition, units are added one by one in random order till a feasible allocation is reached. In practice, this phase is rarely entered, but assures a solution will be found if one exists.

4.2.3 Redundancy Removal

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, success cannot be achieved. In general, the units with minimum stress are tried for removal 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. The idea of the technique is to remember all allocations and child allocations that failed, and to 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. More formally, let us define F to be the set of failed child allocations. Let G be the set of A' and all children of A'. If $G \cap F \neq \phi$, then A' need not be considered as a possible allocation.

Suppose, for example, that the proposed solution A = {3 adders, 2 subtracters, 2 multipliers} failed because a reassignment and scheduling could not be found for its child allocation A(adder) = {2 adders, 2 subtracters, 2 multipliers}. A subtracter was added, and the new allocation A' = {3 adders, 3 subtracters, 2 multipliers} was successful. At this point, the removal phase is entered. At first glance, knowing that A={3 adders, 2 subtracters, 2 multipliers} failed, and that A'={3 adders, 3 subtracters, 2 multipliers} succeeded, it is not clear what will happen with A''={2 adders, 3 subtracters, 2 multipliers}. With remember-and-look-ahead, however, we can immediately dismiss A'' from consideration since F={A(adder)}, G = {A'', A''(subtracter), A''(adder), A''(multiplier)}, gives $G \cap F \neq \phi$ since A''(subtracter) == A(adder).

4.2.4 Ordering Mechanism and Stress Functions

For a successful allocation, a feasible schedule for each child allocation must be found. We thus order the schedules in decreasing order of difficulty, so that we can exit as fast as possible in the event that there is an insufficient allocation. The ordering is a function of the global stress, so that schedules for the failure of highly stressed units are tried first.

The ordering mechanism as well as several other portions of the allocation algorithm rely heavily on the idea of *stress* of a hardware unit. In the remainder of this section three intuitive and experimentally verified heuristics for the stress function are described.

1. Minimum bounds stress, MB: By experimental observation, operations of type j whose relaxed scheduling minimum hardware bound (R) is close to its absolute minimum hardware bound (X), are difficult to schedule. The absolute minimum bound [Rab94] for hardware type j is: $X = (NumNodes_j \cdot duration_j)/(AvailableTime) + K$, where $NumNodes_j$ is the number of nodes that are executed on hardware of type j, $duration_j$ is the clock cycle duration of the hardware, *AvailableTime* is the sample period in clock cycles, and K is the number of allowable faulty units. The minimum bounds stress for hardware type j is:

$$MB_j = 1 - \frac{R - X}{R} = \frac{X}{R} \quad \text{(EQ 1)}$$

   The absolute minimum bound indicates the number of units needed assuming that the flow-graph structure has enough parallelism to achieve 100% hardware utilization. The relaxed scheduling bounds on the other hand, take the graph structure and some data precedences into account giving a more accurate bound. Neither of the bounds take into account constraints such as conflicts in writing to register files, and neither fully honors data precedences. The closer these two bounds are, the smaller the hardware slack available to satisfy these constraints, and thus the better the particular unit is as a candidate for addition.

2. ε-critical network stress, C: If a high percentage of the nodes of a particular hardware type j lie in the ε-critical network, this type of operation is likely a bottleneck for scheduling, and its hardware is thus a good candidate for addition. The ε-critical network consists of all operations on paths which have lengths within a small ε percentage of the critical path length. The ε-critical network stress for hardware type j is:

$$C_j = 1 - \frac{NumNodes_j - \varepsilon NetNumNodes_j}{NumNodes_j} \quad \text{(EQ 2)}$$

These first two measures take into account various elements of the algorithm specification. Both deal with aspects of the overall structure of the flowgraph, and the minimum bounds stress also accounts for the user-specified available time. Since they capture information about the specification and the initial starting allocation, they are most valuable in the beginning of the allocation addition phase. We therefore heavily weight their effect to be greatest in the beginning and to quickly diminish as hardware is added.

3. Scheduling stress, S: Unlike the previous two, this measure changes dynamically with the allocation. It is calculated during the assignment and scheduling. The scheduling difficulty, SD(k), is calculated for each operation, k, and is inversely proportional to the slack time between the As Late As Possible (ALAP) scheduling time and a relaxed As Soon As Possible (ASAP) scheduling time [Pot89]. This value is summed over all nodes of type j, to give the un-normalized stress:

$$stress_j = \sum_{k \in Nodes(j)} SD(k) \qquad \text{(EQ 3)}$$

Since we are interested in the minimum area solution, we normalize the stress value by the hardware cost of the unit, giving the scheduling stress for hardware type j as a function of the scheduling difficulty and the hardware cost:

$$S_j = f(stress_j, c_j) = \frac{stress_j^{\,2}}{c_j} = \frac{\left(\sum_{k \in Nodes(j)} SD(k)\right)^2}{c_j} \qquad \text{(EQ 4)}$$

where c_j is the hardware cost (area) of an execution units of hardware type j.

Finally, we define the global stress, G, for hardware type j:

$$G_j = f(MB_j, C_j, S_j) = S_j \cdot (MB_j \cdot C_j)^{\frac{1}{x^{\beta}}} \qquad \text{(EQ 5)}$$

where x is the number of additional units added, and β is an empirical parameter determined through testing to be equal to 3. The global stress is a functionally weighted function of the heuristics, and was constructed through the use of testing and statistical validation. As mentioned above, since MB and C capture information about the starting allocation, they have a large impact

on the global stress function in the early stages of the allocation addition phase. The scheduling stress, S, quickly gains dominance as units are added.

5.0 Transformations for HBISR Design

Transformations are alterations in the computational structure such that the behavior (the relationship between output and input data) is maintained. Transformations are used extensively in several computer science, computer engineering, and CAD areas, most often in compilers [Fis88] and behavioral-level synthesis [Sno78, Wal89, Pot91, Tri87]. This section shows how transformations, using specifically tailored optimization techniques, can significantly reduce the area overhead for designs with BISR requirements.

5.1 Key Ideas and Motivational Examples

The basic idea behind using transformations in behavioral-level synthesis for HBISR is to transform the computation according to the needs imposed by the available hardware, for each possible scenario of failed units. The simple example in Figure 3 will be used to illustrate this

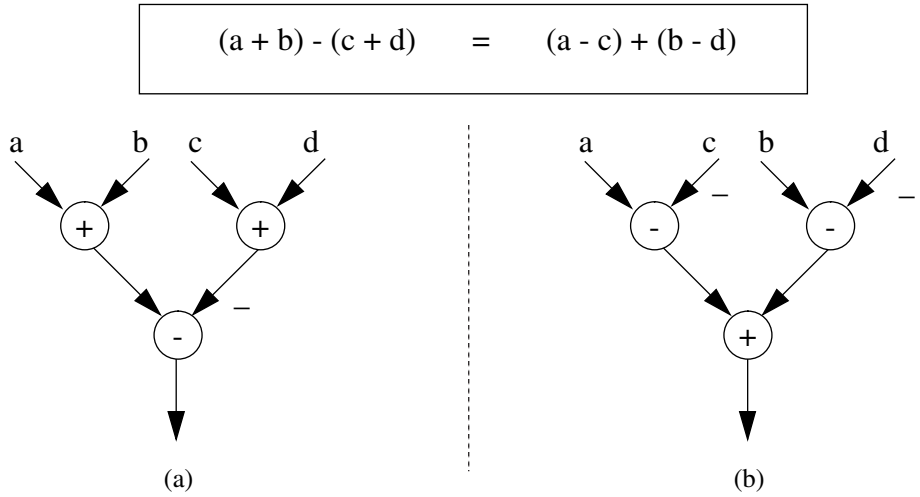

FIG. 3. Transformations for HBISR: using associativity and the inverse element law, the HBISR hardware overhead is reduced.

idea. In all the examples in this section, assume that each operation takes one control cycle, and that transformations are done in such a way that important numerical properties (e.g. numerical stability and overflow control) are maintained in the transformed designs. The validity of the assumptions about the numerical properties of the transformed designs can be verified using fixed-point simulation tools [e.g. Rab91]. The assumed available time for the first example is two control cycles. The identity shown in Figure 3 is used to transform 3a into 3b. It is easy to verify

that both algorithms calculate the same output for the same set of inputs. All operations lie on the critical path, so it is not possible to reduce BISR overhead using the techniques of Section 4.0. In this case, however, transformations can help to reduce the overhead. If we consider only implementation 3a, and assume that any unit can fail, then 3 adders and 2 subtracters are needed, since 2 adders and 1 subtracter were needed for the non-BISR implementation. However, if we consider both implementations, only 2 of each type of unit are needed. If the subtracter fails, we can use implementation 3a which needs 2 adders and 1 subtracter, and when the adder fails we can use implementation 3b which needs 2 subtracters and 1 adder.

In general, there exists a large variety of transformations, each of which reduces a computation in different ways. The transformations to reduce HBISR overhead can be grouped into two classes: (1) transformations to increase the resource utilization (and therefore need) of the units of the same type as the failed execution unit, and (2) transformations to reduce the number of operations that use the type of unit that failed.

While the former strategy is the same as that used during scheduling, the latter is specific to transformations. Transformations in the former group include retiming, functional pipelining, associativity, and loop permutation, while those in the latter group include strength reduction (substitution of multiplication with a constant by shifts and additions), constant propagation, dead code elimination and common sub-expression elimination. Some transformations can even be used for both strategies simultaneously (e.g. inverse element law, distributivity, loop fusion, and loop blocking).

The remainder of this section illustrates how four important and powerful transformations, associativity, the inverse element law, retiming, and pipelining, can be used for transformation-based HBISR. Although it is not explicitly stated, it is implied that transformations in the explanatory examples and in the software application are supported by the commutativity transformation.

Figure 4 shows the application of associativity for HBISR. For this example, the available time is three and assignment and scheduling flexibility does not help to reduce overhead. Notice that shift A on Figure 4a and multiplication B on Figure 4b are the only operations which are not on the critical path. It can be determined that the minimum non-BISR hardware configuration, for

the computation of Figure 4a requires 2 adders, 2 multipliers and 1 shifter. Using associativity,

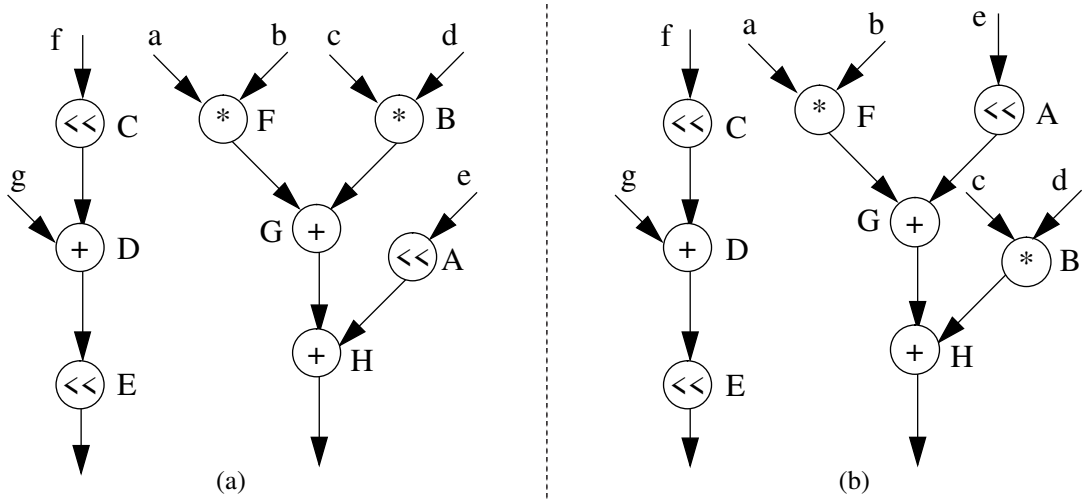

FIG. 4. Associativity for HBISR: (a) before and (b) after application of associativity.

only one additional adder and one additional shifter are needed. Table 2 shows the feasible schedules when 3 adders, 2 multipliers and 2 shifters are available for various scenarios of unit failures. When a shifter fails, the implementation from Figure 4a is used, when a multiplier or adder fail the implementation of Figure 4b is used (actually, either 4a or 4b can be used when an adder fails).

Control Step	Shifter Failure			Multiplier Failure			Adder Failure		
	>>	*	+	>>	*	+	>>	*	+
1	C	F, B		C, A	F		C, A	F	
2	A		D, G		B	D, G		B	D, G
3	E		H	E		H			H

Table 2: Feasible schedules for the example of Fig. 4. Two shifters, two multipliers, and two adders are sufficient for HBISR implementation.

The inverse element law transformation is used in the example shown in Figure 5. For an available time of two, the non-BISR design can be implemented with 1 adder and 1 subtracter. As in the previous example, scheduling and assignment cannot enable a low overhead HBISR implementation (of only one additional execution unit). However, use of the inverse element law (combined with the enabling transformations of distributivity and associativity), gives two identities (Figure 5) which can be used to transform 5a into 5b.

By using this transformation, just two subtracters and one adder are sufficient to enable the implementation of the required functionality, regardless of which unit is detected as faulty. If one of subtracters fails, the structure of Figure 5a is used; if the adder fails then 5b used.

Note that it is sometimes possible to totally eliminate the need for a particular type of unit. The computation of Figure 5, for example, could be implemented for HBISR using only three subtracters. This HBISR scheme is not preferred however, since a subtracter is slightly more expensive than an adder. Also notice that this particular application of the inverse element law can similarly be applied to pairs of multiplications and divisions. In such a case, however, it would be significantly more efficient to use the HBISR scheme which uses two dividers and one multiplier instead of the solution where three dividers are used.

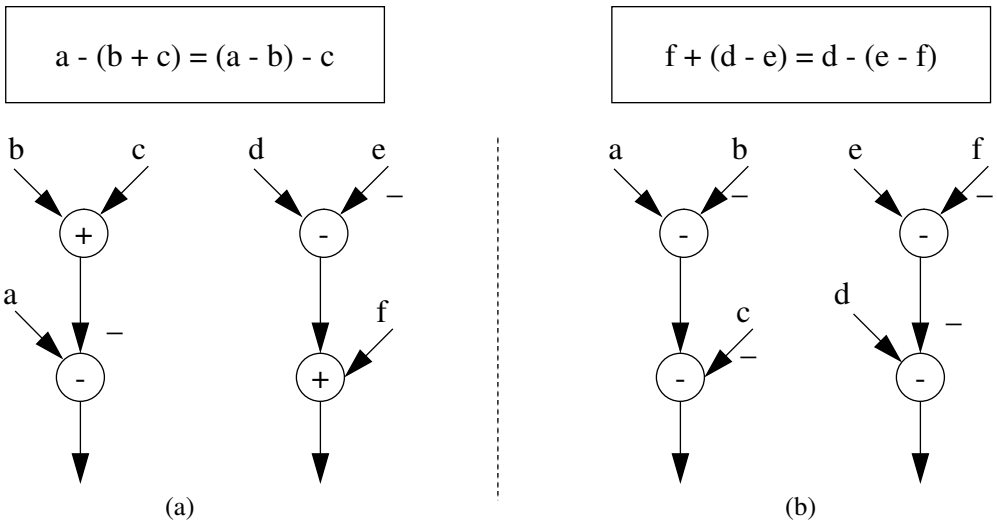

FIG. 5. Inverse element law for HBISR.

Using a larger set of transformations (to include algebraic and redundancy manipulations) brings more options for the trading of operations. For example, x^2 - y^2 can be implement in two ways, either as $(x \times x) - (y \times y)$ or as $(x - y) \times (x + y)$. In this identity one multiplication can be traded for an adder.

Next, Figure 6 shows how retiming can be used for HBISR. The available time in this example is two control cycles. The rectangles in Figures 6 (and 7) denote pipeline delays (states). The operation following the delay uses data produced in the previous iteration by the operation preceding the delay. Therefore, operations which have direct dependencies only with respect to the states, can be scheduled in the first control step. Notice, once again that all the operations reside on the critical path, so there is no flexibility during scheduling.

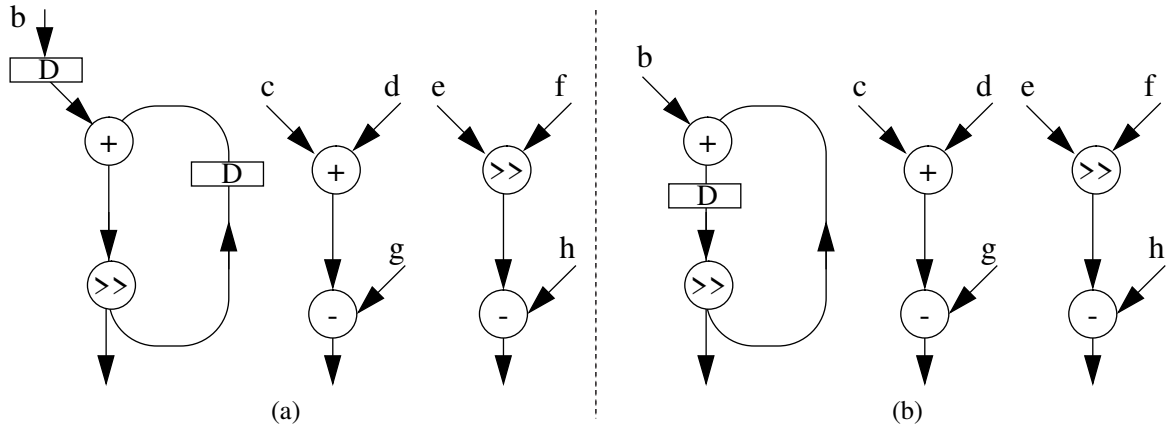

FIG. 6. Retiming for HBISR: rearranging ASAP and ALAP times [Wal91] so that operations which require the faulty units are more uniformly distributed over the available time.

Although retiming cannot, in this case, change the slacks [Wal91] on various operations, it can reshuffle the operation overlaps. This redistribution is done such that operations competing for a faulty unit are no longer bound to happen in the same control step. Analysis of the various schedules shows that 3 subtracters, 2 adders, and 2 shifters are sufficient for HBISR implementation. This results once again in a lower overhead than that achievable using only assignment and scheduling.

It is important to recognize that the effectiveness of retiming as a behavioral-level synthesis optimizing transformation is often strongly constrained by the underlying structure of computation. Furthermore many widely used applications have no algorithmic delays in their functional description, and therefore retiming can not be applied at all. In such instances, pipelining can be used to reduce the HBISR overhead cost.

Functional pipelining [Par95] with n stages is functionally equivalent to the procedure of first adding n algorithmic delays on each primary input or each primary output, then applying retiming [Pot91]. We assume that the number of pipeline stages is fixed and indicated by the amount of additional latency which can be introduced.

Figure 7 illustrates how pipelining can be used as a HBISR transformation. It is assumed that, due to a latency constraint, at most one pipeline stage can be used. The available time is two control steps. The non-BISR implementation requires either 2 shifters, 1 adder and 1 multiplier, or 1 shifter, 2 adders, and 1 multiplier. Using the structures of Figure 7b and Figure 7c, just 2 of each unit are necessary for HBISR implementation. The corresponding schedules are shown in Table 3.

The algorithm version that was used in each scenario is also indicated on the table. Pipelining

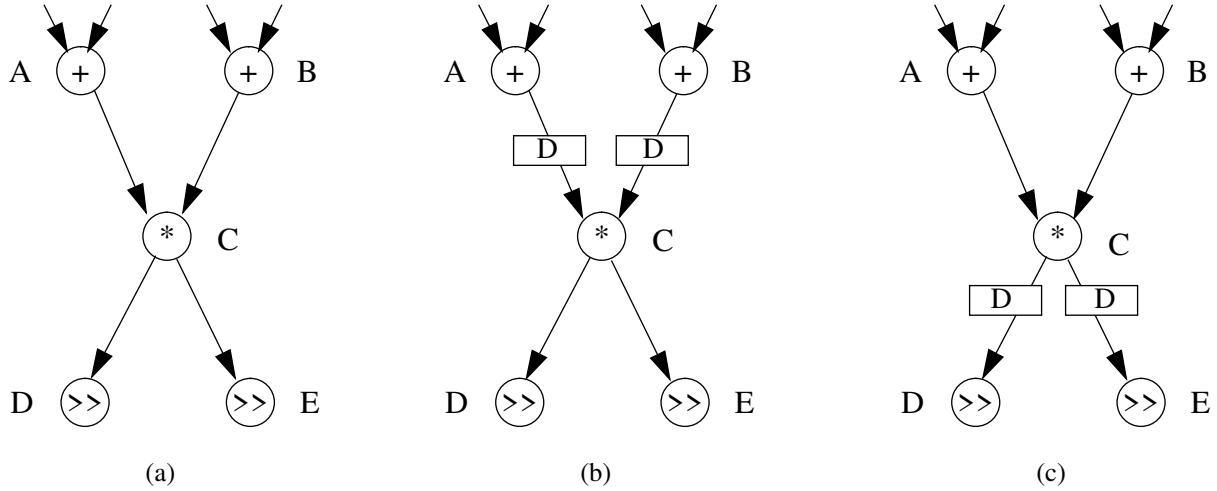

FIG. 7. Pipelining for HBISR.

obviously does not change the number of operations and thus belongs to the group of transformations which reduce HBISR overhead by redistributing scheduling freedom of the various types of operations.

Control Step	Adder Failure (7b)			Shifter Failure (7c)			Multiplier Failure (7b)		
	>>	*	+	>>	*	+	>>	*	+
1		C	A	D		A, B		C	A, B
2	D, E		B	E	C		D, E		

Table 3: Schedules for the example of Fig. 7 after the application of appropriate pipelining.

5.2 Optimization Algorithm

The transformation-based HBISR optimization algorithm is given by the following pseudo-code:

```
while (!Done){
   for (i = 1.. hw_types) { resolved(i) = NO;}
   for (count = 1.. hw_types){
      k = SelectMostCriticalResource();
      OptimizeUsingProbalisticSamplingWithFailedUnit(k);
      if (estimate() indicates feasibility) {resolved(k) = YES;
      }else {OptimizeUsingPipeliningWithFailedUnit(k);
            if (estimate() indicates feasibility) {resolved(k) = YES;
            } else {break;}
   }
   if (there is an unresolved resource) {AddHardware();
   }elseif (scheduling() != feasible) {AddHardware();
   } else { Done = TRUE;}
}
```

The initial hardware allocation is given by the HBISR allocation algorithm. New resources are added until a transformed and feasibly scheduled version of the computation is found for each fault unit scenario. In the initial phases of the exploration process, estimation routines based on relaxed scheduling [Rab94] are used in place of scheduling. Only when estimates indicate that a complete solution is potentially found, is the scheduling run. The AddHardware and SelectMost-CriticalResource routines add resources using the stress functions of Section 4.2.4, in determining which hardware to add and the order in which to try the fault scenarios.

For the core optimization, two approaches are employed: a probabilistic sampling algorithm [Pot91] and a pipelining-based algorithm [Pot91]. This probabilistic sampling algorithm applies two types of basic moves: retiming and generalized associativity, where the later is a transformation that combines associativity, inverse element law, and commutativity moves. The algorithm has two phases. The first phase is a global search using probabilistic sampling, where the design space is probabilistically evaluated to detect the k most promising starting points (k is a small integer number which is a function of the number of nodes in the computation). Note that starting points with varying numbers of operations of various types (e.g. subtraction vs. addition) are generated using generalized associativity moves, which include the inverse element law. The second, local optimization phase, uses the basic steepest descent approach to locally maximize these starting points. After each move, the objective function is evaluated, to get an estimate of the final area (execution units, interconnect, and registers) expected from the system. This objective function is composed of 3 key parts, all of which are strongly correlated to the final area: the critical path, the

number of delays, and a measure of the expected resource utilization of each hardware type (the overlap component). During the local phase, the overlap components of the objective function are normalized by the available number of resources of each hardware type. When a unit is in short supply due to failure, the overlap component for the resource is large, and thus the algorithm will transform the graph in such a way that the need for this unit is alleviated.

When the probabilistic sampling does not succeed in transforming the graph for successful implementation under the given fault scenario, a pipelining-based optimization [Pot91] is used, in which varying number of pipeline states are tried.

Both the probabilistic sampling and pipelining algorithms run in $O(n^2)$ time. Since the number of required hardware resources is bounded by the number of nodes in the computation, the worst-case running time is $O(n^3)$. Experimental studies indicate that the actual run-time is quadratic and was less than 10 minutes for all examples on a Sun SPARCstation-5.

Notice that both classes of transformations for HBISR are utilized: (1) transformations to increase the chance for high utilization (and therefore reduced need) of the units of the same type as the failed execution unit, and (2) transformations to reduce the number of operations of a failed type by trading operations of that type for other operations.

6.0 Experimental Results

The HBISR methodology, techniques and proposed algorithms were validated on the set of DSP, video, control, and communication examples shown in Table 4. Supporting tools from the Hyper behavioral-level synthesis system [Rab91], were used for other synthesis tasks. The table shows all relevant data for the standard and the HBISR synthesis procedures. During the selection of benchmark examples, special attention was devoted to include examples with a variety of computational structures. For example, note that although the different forms of the 8th order low pass IIR Avenhaus filters provide the same functionality, they have drastically different structures and sizes. The average and median HBISR design area overheads were 18.8% and 19.4%, respectively. Note also that although the initial implementations of all examples had on average 3.7 dif-

ferent types of hardware units, an average of only 2.4 additional units were needed for the HBISR designs.

Example[a]	Non-BISR #Units	HBISR #Units	# Hardware Classes	Non-BISR Area (mm^2)	HBISR Area (mm^2)	Percentage Area Overhead
Jaumann	5	8	4	4.39	7.07	61.0
WFFT8	6	8	3	1.79	2.49	39.0
5 WDF	6	9	4	1.43	1.73	21.0
8 IIR DFa	7	10	4	8.06	10.86	34.7
8 IIR GMa	8	9	4	4.84	4.95	2.3
WConv 8	8	9	3	1.91	2.38	24.6
7 IIRa	9	11	4	18.18	23.76	30.7
8 IIR GMb	9	12	4	6.66	6.88	3.3
8 IIR P	9	12	w4	2.23	2.55	14.4
8 IIR C	9	12	4	4.24	4.69	10.6
DCT	10	12	3	1.40	1.77	26.4
5 IIR	11	14	4	4.55	5.56	22.2
7 IIRb	17	19	4	4.47	4.92	3.1
8 IIR DFb	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
Nonlin 6	10	12	3	1.90	2.06	8.4
Sort 9	11	13	3	3.21	4.05	26.2
Sort 6	6	8	3	1.32	1.51	15.3

Table 4: Results for assignment and scheduling-based HBISR.

a. Examples: Jaumann - Jaumann LDI filter; WFFT8 - Winograd 8 point FFT; 5 WDF - 5th order elliptical wave digital filter; 8 IIR DFa & 8 IIR DFb - 8th order bandpass IIR direct form filter for two different set of timing constraints; 8IIR GMa & 8IIR GMb - 8th order bandpass IIR filter Gray Markel form for two sets of timing constraints; WConv8 - 8 point Winograd Cyclic Convolution; 7IIRa & 7IIRb - 7th order low pass IIR filter for two sets of timing constraints; 8 IIR P & 8 IIR C - 8th order IIR filter parallel and cascade form; DCT - 1 dimensional DCT; 3StateLinCn - 3 state linear controller; Wavelet - Wavelet quadrature mirror filter; Nonlin 6 - 6 order nonlinear filter, and Sort 9 and Sort 6 - fast sorting algorithms for 6 and 9 numbers respectively

Table 5 shows results for several examples designed using the transformation-based methods of HBISR design. The average and median area increases are only 8.8% and 7.2%, respectively. The examples had an average of 4 different types of execution units, but an average of only 1.8 additional hardware units were needed.

Example[a]	Non-BISR #Units	HBISR #Units	# Hardware Classes	Non-BISR Area (mm^2)	HBISR Area (mm^2)	Percentage Area Overhead
11 FIR	8	9	4	5.45	6.5	19.3
7 IIR	7	9	4	9.27	9.92	7.0
35 FIR	7	8	4	12.31	13.34	8.4
55 FIR	14	16	4	20.77	23.44	12.9
8 IIR	16	18	4	24.85	27.04	8.8
Lin 3	18	19	3	33.06	34.52	4.4
Lin 4	21	23	3	36.00	38.49	6.9
Echo 8	17	19	5	12.27	12.81	4.4
Adapt 6	23	26	5	14.78	15.84	7.2

Table 5: Results for transformation-based HBISR.

a. Examples: 11 FIR - 11th order high pass FIR filter; 7 IIR - 7th order low pass IIR filter; 35 FIR - 35th order Butterworth Flat Low Pass FIR filter; 55 FIR - 55th order multi-band FIR filer; 8 IIR - 8th order IIR Avenhaus direct form II filter; Lin 3 and Lin 4 - 2 different 5-state linear controllers; Echo8 - 8 tap echo canceller, and Adapt 6 - 6th order adaptive filter

While the HBISR techniques increase the yield per wafer (percentage of functional die), the area overhead reduces the number of wafer per die. Productivity (number of functional dies per wafer) takes both of these effects into account. Productivity improvement can thus be used to measure the effectiveness of the HBISR techniques.

For these calculations, we used Stapper's yield formula [Sta92], which calculates the probability that exactly m out of n modules operate correctly for a given value of the variability parameter μ and single module yield, Y1:

$$\bar{Y}_{mn} = \binom{n}{m} \bar{Y}_1^m \left(\prod_{i=0}^{m-1} \frac{\mu + i}{\mu + i\bar{Y}_1} \right) \times (1 - \bar{Y}_1)^{n-m} \left(\prod_{j=0}^{n-m-1} \frac{\mu + j\bar{Y}_1/(1 - \bar{Y}_1)}{\mu + m\bar{Y}_1 + j\bar{Y}_1} \right)$$

A slight modification is made to the formula to take into account units of largely different areas.

This model is based on the combination of the binomial and beta distributions for faults. The high accuracy of this model has been demonstrated on a variety of real-life production designs. Although Stapper's procedures primarily target BISR memory design, they have been regularly and successfully used in both BISR for custom [Kor84] and programmable [Pat89] datapath analysis.

Table 6 shows the yield and productivity data for the examples designed using transformation-based HBISR. A 10% initial yield is assumed for designs without redundant HBISR circuitry. This or similar values were also assumed in [Sta92, Kor84, Pat89]. The relative productivity is the relative yield increase divided by the relative area increase. The data are calculated for various values of the variability parameter μ. This parameter gives an indication of the assumed probability of clustered defects, the most common sources of chip malfunctions. Large values of μ correspond to smaller levels of clustering, and therefore lower processing variability. For all examples, including those of Table 4 whose yield and productivity data are not shown here, a significant improvement in productivity is apparent for all values of μ.

Example	Percentage Yield					Relative Productivity				
	μ=0.5	μ=1	μ=2	μ=5	μ=∞	μ=0.5	μ=1	μ=2	μ=5	μ=∞
11 FIR	16.62	18.42	20.50	23.52	30.02	1.393	1.544	1.718	1.971	2.531
7 IIR	15.30	16.60	18.00	19.89	23.34	1.430	1.551	1.682	1.859	2.181
35 FIR	16.82	18.69	20.80	23.78	29.63	1.552	1.719	1.919	2.199	2.733
55 FIR	15.18	16.48	18.02	20.46	27.82	1.345	1.460	1.596	1.812	2.464
8 IIR	15.11	16.38	17.88	20.32	28.42	1.389	1.506	1.643	1.868	2.612
LIN 3	15.49	16.89	18.58	21.37	31.62	1.484	1.618	1.780	2.047	3.029
LIN 4	14.92	16.13	17.56	19.93	29.49	1.396	1.509	1.643	1.864	2.759
Echo 8	15.07	16.32	17.82	20.24	28.65	1.443	1.563	1.707	1.939	2.744
Adapt 6	14.54	15.62	16.89	18.98	27.69	1.356	1.457	1.568	1.771	2.583

Table 6: Yield and relative productivity for transformation-based HBISR designs, for various values of the variability parameter μ. The initial yield is assumed to be 10%.

In general, additional multiplexers add extra delays which can affect clock cycle time and throughput (if they are on the critical path). Another important consequence of the low hardware overhead of HBISR is fewer multiplexers as compared to alternate fault-tolerance techniques.

7.0 Generalizing the HBISR Methodology

7.1 Arbitrary Hardware Models

In this section we describe how the HBISR methodology can be used when an arbitrary hardware model is adopted. All that is necessary, actually, is to first propose the HBISR implementation, and then check whether the computation can be correctly implemented on all subsets of the implementation where hardware primitives (execution units, register and interconnects) of the model are removed (assumed faulty and not used) one by one.

Note that even in relatively small designs, the total number of execution units, interconnect, and registers is usually high. Therefore, while an arbitrary model offers potential for further hardware reduction and an attractive conceptual generalization, the number of required schedules and assignments may combinatorially explode.

Clearly there is a trade-off between potential overhead reduction and tractability of the synthesis algorithms. A more practical HBISR scheme can be achieved by grouping several hardware components together and assuming that all of them are either simultaneously faulty or simultaneously functional. This is actually a generalization of the initial proposed dedicated register file hardware model (Figure 1), where an execution unit, its register files and corresponding interconnects are considered as simultaneously susceptible to a fault.

7.2 Varying Levels of Fault Tolerance

Until now we have restricted our attention to the case when the number of failed modules, K, is equal to one. A natural question to ask is how well the proposed methodology is suited to address higher levels of fault tolerance. In general as K is increased, the improvement in the productivity can give diminishing returns, and can even produce lower productivity [Sta92]. An interesting issue for HBISR design is the actual selection of the value of K which gives the optimal effective yield, as a trade-off between resilience to failure and hardware overhead.

Analysis of the number of different schedules and assignments needed for high levels of fault tolerance indicates that direct application of the methodology is most likely impractical, assuming current manufacturing technologies. However, conceptually simple modifications pro-

vide a convenient way to achieve our goal. In the first part of this section, the trends in the implementation complexity of the generic methodology will be analyzed. Then, several modifications are introduced for extending the behavioral-level synthesis techniques to efficiently handle large values of K.

Suppose that we have targeted an ASIC design with n different types of operations and that we want an implementation that can tolerate up to K nonfunctional units. The number of different schedules and assignments needed is equal to the number of combinations of K elements from n [Hal86]: $\binom{n}{K}$. It is clear that the required number of schedules and assignments grows quickly with K and n. For example, a fault tolerance to 4 failed units out of a total of 9 units would require 2160 different schedules and assignments.

It is immediately clear that for larger values of K it is even more important to efficiently use the look ahead and remember technique, and to terminate unfeasible allocations as early as possible by trying them early in the search process. Another important point is that design space exploration can be conducted efficiently by employing an iterative deepening technique, by testing the proposed allocations for iteratively increasing values of K.

However, even if all necessary assignments and schedules are produced in a reasonable amount of time, important implementation details (e.g. the exponentially growing size of the controller or the number of different programs which have to be generated and stored) become the limiting factor. In this situation, techniques which combine the scheme when only assignment flexibility is used with the one when both scheduling and assignment flexibility are explored provides a good trade-off between complexity and efficiency. Several such schemes can be envisioned. For example, after generating the solution for K = 2, if we add two instances of each type of execution unit with the corresponding register files and interconnect as required for the assignment only-based BISR, we have a design which is fault tolerant against as many as four faulty units.

Another possibility is that we divide all execution types in two subsets. The first subset of execution types is treated using the assignment-only BISR scheme, while the second subset is addressed using the full-fledged behavioral-level synthesis approach. In this scenario, it is apparently advantageous to select more expensive units for the second subset.

7.3 Application-Specific Programmable Processor (ASPP) designs

The HBISR approach, as demonstrated, can be used for ASIC yield improvement or low-hardware overhead fault-tolerance against permanent faults. The technique is directly built on the flexibility provided by behavioral-level synthesis during design space exploration. The identification and the techniques for exploiting this flexibility, however, are in themselves important. Intelligent strategies to use the flexibility of solutions can also be used in the reconfigurable datapath design of Application Specific Programmable Processors (ASPPs). An ASSP design provides an efficient implementation for a set of different applications. This is in contrast to traditional ASIC designs which implement a single application. Minimal hardware ASPP implementation is achieved most often by identifying a set of implementations for individual applications which are not necessarily minimum hardware implementations, but which are similar in terms of the required hardware resources.

Consider, for example, the design of an ASPP to implement the two different computations A and B. Let A_i and B_j represent particular implementation solutions for the computations A and B, where $|i|$ and $|j|$ are the total number of possible implementations of A and B respectively. As the ASPP implementation must be able to implement both computations, its hardware is the union of the hardware, $A_i \cup B_j$, for any i and j. The goal is not to find the $Min(A_i) \cup Min(B_j)$ implementation, but to find the $Min(A_i \cup B_j)$ solution, which in many instances is one for which A_i and B_j have similar hardware implementations. The techniques introduced in this paper have a high potential to facilitate the synthesis of ASPP datapaths, due to their ability to produce a great variety of competitive solutions. This increases the likelihood of finding the solution pair (A_i, B_j), for which both A_i and B_j have a small implementation cost and require mainly identical resources.

8.0 Conclusions

Behavioral-level synthesis of datapaths has traditionally concentrated on synthesizing a specific implementation for a given computational problem. New techniques have been presented to compose a reconfigurable BISR implementation with a minimum amount of area overhead. BISR is an efficient yield, productivity, and reliability fault tolerance improvement technique, which will continue to gain importance especially with the increase in commercial significance of massive parallelism. We have presented novel synthesis techniques based on assignment, scheduling, and transformations, which support a new Heterogeneous BISR methodology for ASIC designs. These methods are based on the flexibility of the design solution space and the exploration poten-

tial of behavioral-level synthesis processes to find designs where resources of several different types can be backed up with the same unit.

Future issues include evaluating the effectiveness of alternate hardware models and varying levels of fault tolerance, and investigating the role of other system-level and behavioral-level synthesis tasks (e.g. algorithm selection, module selection, and partitioning) in fault-tolerant design. A key future research direction involves the extension of techniques to exploit algorithm flexibility, for new applications such as ASPP design. Behavioral-level synthesis for the design of reconfigurable datapath structures is a new field; with many venues of research still remaining to be explored.

9.0 References

[Abr90] M. Abramovici, M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Designs*, Computer Science Press, New York, NY, 1990.

[Aru80] J.A. Arulpragasm and R.S. Swartz, “A Design for Process State Preservation on Storage Unit Failure,” *International Symposium on Fault-Tolerant Computing*, pp. 47-52, 1980.

[Avi71] A. Avizienis, et al., “The STAR (Self-Testing and Repairing) Computer: An Investigation on the Theory and practice of Fault-Tolerant Computer Design,” *IEEE Transactions on Computers,* Vol. 20, No. 11, pp. 1312-1321, 1971.

[Bar87] P.H. Bardell, W.H. McAney, and J. Savir, *Built-In Test for VLSI: Pseudorandom Techniques*, Wiley, Somerset, NJ 1987.

[Bir89] W.P. Birmingham, A.P. Gupta, and D.P. Siewiorek, "The MICON System for Computer Design," *ACM/IEEE Design Automation Conference*, pp. 135-140, 1989.

[Bra84] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

[Bre88] A.A. Brennan, "Automatic Synthesis for Reliability," *Technical Report CMUCAD-88-2*, Carnegie-Mellon University, January 1988.

[Cha92] A. Chandrakasan, et al., “HYPER-LP: A System for Power Minimization Using Architectural Transformations,” *IEEE International Conference on CAD*, pp. 300-303, 1992.

[Che90] K.T. Cheng and V.D. Agarwal, "A Partial Scan Method for Sequential Circuits with Feedback," *IEEE Transactions on Computers*, Vol. 39, No. 4, pp. 544-548, 1990.

[Cla88] L. Classen, et al., “Automatic Synthesis of Signal Processing Benchmarks using the CATHEDRAL Silicon Compilers,” *IEEE CICC*, pp. 14.7.1-14.7.4, 1988.

[DeM94] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, Mc Graw Hill, New York, NY, 1994.

[Eic78] E.B. Eichelberger and T.W. Williams, " A Logic Design Structure for LSI Testability," *Journal of Design Automation and Fault-Tolerant Computing*, Vol. 2, No. 2, pp. 165-178, 1978.

[Fis88] C.N. Fischer and R.J. LeBlanc, Jr., *Crafting a Compiler,* The Benjamin/Cummings Publishing Co. Inc., Menlo Park, CA, 1988.

[Gre84] J.W. Greene and A.E. Gamal, "Configuration of VLSI Arrays in the Presence of Defects," *Journal of the ACM*, Vol. 31, No. 4, pp. 694-717, 1984.

[Gri91] M. Griffin, et al., “An 11-Million Transistor Neural Network Execution Engine,” *ISSCC*, pp. 180-181, San Francisco, CA, 1991.

[Gul93] R.K. Gulati and C.F. Hawkins, *I_{DDQ} Testing of VLSI Circuits*, Kluwer Academic Publishers, Norwell, MA, 1993.

[Hal86] M.Hall, *Combinatorial Theory*, John-Wiley, New York, NY, 1986.

[Has90] N. Hassan and C.L. Liu, "Fault Covers in Reconfigurable PLAs," *Proceedings of the International Conference on Fault-Tolerant Computing*, pp. 166-173, 1990.

[Hay74] J.P. Hayes, "On Modifying Logic Networks to Improve their Diagnosability," *IEEE Transactions on Computers*, Vol. 23, No. 1, pp. 56-62, 1974.

[Kar92] R. Karri and A. Orailoglu, "Transformation-Based High-Level Synthesis of Fault-Tolerant ASICs," *ACM/ IEEE Design Automation Conference*, pp. 662-665, 1992.

[Kar93a] R. Karri and A. Orailoglu, "High-Level Synthesis of Fault-Secure Microarchitectures," *ACM/IEEE Design Automation Conference*, pp. 429-433, 1993.

[Kar93b] R. Karri and A. Orailoglu, "Optimal Self-Recovering Microarchitectural Synthesis", *International Symposium on Fault-Tolerant Computing,* pp. 512-521, Touluse, France, 1993.

[Kik91] S. Kikuda, "Optimized Redundancy Selection Based on Failure-Related Yield Model for 64-Mb DRAM and Beyond," *IEEE Journal of Solid-State Circuits,* Vol. 26, No. 11, pp. 1550-1555, 1991.

[Kor84] I. Koren and M. Breuer, "On Area and Yield Considerations for Fault-Tolerant VLSI Processor Arrays," *IEEE Transactions on Computers*, Vol. C-33, No. 1, pp. 21-27, Jan. 1984.

[Kor85] I. Koren and D.K. Pradhan, "Introducing Redundancy into VLSI Designs for Yield and Performance Enhancement," *International Conference on Fault-Tolerant Computing*, pp. 330-335, 1985.

[Lee87] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, Vol. 36, No. 1, pp. 24-36, 1987.

[Lee90] D.H. Lee and S.M. Reddy, "On Determining Scan Flip-Flops in Partial-Scan Designs," I*EEE International Conference on CAD, pp. 322-325, 1990.*

[Lee92] T. Lee, W. H. Wolf, and N.J. Jha, "Behavioral Synthesis for Easy Testability in Data Path Scheduling," *IEEE International Conference on CAD*, pp. 616-619, 1992.

[Lei85] T. Leighton and C.E. Leiserson, "Wafer-Scale Integration of Systolic Arrays," *IEEE Transactions on Computers*, Vol. 34, No. 5, pp. 448-461, 1985.

[Lev68] K.N. Levitt, M.W. Green, and J. Goldberg, "A Study of the Data Communication Problems in Self-Repairable Multiprocessors," *Conference Proceedings of AFIPS*, Vol. 32, pp. 515-527, Thompson Book, Washington, DC, 1968.

[Lew79] D.W. Lewis, "A Fault-Tolerant Clock Using Standby Sparing," *International Conference on Fault-Tolerant Computing, IEEE Computer Society*, Madison, WI, pp. 33-40, 1979.

[Man82] T. Mano, et al., "Redundancy Circuits for a Fault-Tolerant 256K MOS RAM," *IEEE Journal of Solid-State Circuits*, Vol. 17, No. 8, pp. 726-731, 1982.

[McF90] M.C. McFarland, A.C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems," *Proceedings of the IEEE*, Vol. 78, No. 2, pp. 301-317, 1990.

[Moo86] W.R. Moore, "A Review of Fault-Tolerant Techniques for the Enhancement of Integrated Circuit Yield," *Proceedings of the IEEE,* Vol. 74, No. 5, pp. 684-698, 1986.

[Neg89] R. Negrini, M.G. Sami, and R. Stefanelli, *Fault Tolerance Through Reconfiguration in VLSI and WSI Arrays*, MIT Press, Cambridge, MA, 1989.

[Nie91] T.M. Niermann and J.H. Patel, "HITEC: A Test Generation Package for Sequential Circuits," *European Design Automation Conference,* pp. 214-218, 1991.

[Par95] K.K. Parhi, "High-Level Algorithm and Architecture Transformations for DSP Synthesis", *Journal of VLSI Signal Processing*, Vol. 9, No. 1-2, pp.121-143, 1995.

[Pat88] D.A. Patterson, G. Gibson, and R.H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings SIGMOD*, pp. 109-116, 1988.

[Pat89] D.A. Patterson and J.L. Henessy, *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann Publishers, San Mateo, CA, 1989.

[Pot89] M. Potkonjak and J. Rabaey, “A Scheduling and Resource Allocation Algorithm for Hierarchical Signal Flow Graphs,” *ACM/IEEE Design Automation Conference,* pp. 7-12, 1989.

[Pot91] M. Potkonjak and J. Rabaey, “Optimizing Resource Utilization Using Transformations,” *IEEE International Conference on CAD*, pp. 88-91, 1991.

[Rab91] J. Rabaey, et al., “Fast Prototyping of Data Path Intensive Architectures,” *IEEE Design and Test*, Vol. 8, No. 1, pp 40-51, June 1991.

[Rab94] J. Rabaey and M. Potkonjak, "Estimating Implementation Bounds for Real-Time Application Specific Circuits," *IEEE Transactions on CAD*, Vol. 13, No. 6, pp. 669-683, 1994.

[Rag91] V. Raghavendra and C. Lursinsap, “Automated Micro-Roll-Back Self Recovery Synthesis,” *ACM/IEEE Design Automation Conference*, pp. 385-390, 1991.

[Sar84] D.B. Sarrazin and M. Malek, "Fault-Tolerant Semiconductor Memories," *IEEE Computer,* Vol. 17, No. 8, pp. 49-56, 1984.

[Sat92] K. Sato, et al., “A System-Integrated ULSI Chip Containing Eleven 4 Mb RAMs, Six 64kb SRAMs and an 18k Gate Array,” *ISSCC*, pp. 52-53, San Francisco, CA, 1992.

[Sch78] S.E. Schuster, "Multiple Word/Bit Line Redundancy for Semiconductor Memories," *IEEE Journal of Solid-State Circuits,* Vol. 13, No. 10, pp. 698-703, 1978.

[Sie92] D.P. Siewiorek and R.S. Swartz, *Reliable Computer Systems: Design and Evaluation,* 2nd edition, Digital Press, Burlington, MA, 1992.

[Smi81] R.T. Smith, et al., "Laser Programmable Redundancy and Yield Improvement in a 64K DRAM," *IEEE Journal of Solid-State Circuits,* Vol. 16, No. 10, pp. 506-514, 1981.

[Sno78] E.A. Snow, D.P. Siewiorek, and D.E. Thomas, “A Technology-Relative Computer-Aided Design System: Abstract Definition, Transformations and Design Tradeoffs,” *ACM/IEEE Design Automation Conference*, pp. 220-226, 1978.

[Sta92] C. H. Stapper, "A New Statistical Approach for Fault-Tolerant VLSI Systems," *International Symposium on Fault-Tolerant Computing,* pp. 356-365, Boston, MA, 1992.

[Tan92] A. Tanabe, et al., "A 30-ns 64-Mb DRAM with Built-in-Self-Test and Self-Repair Functions," *IEEE Journal of Solid-State Circuits,* Vol. 27, No. 11, pp. 1525-1533, 1992.

[Tri87] H. Trickey, “Flamel: A High-Level Hardware Compiler,” *IEEE Transactions on CAD*, Vol. 6, No. 2, pp. 259-269, 1987.

[Wal89] R.A. Walker and D.E. Thomas, “Behavioral Transformations for Algorithmic Level IC Design,” *IEEE Transactions on CAD*, Vol 8. No.10, pp. 1115-1127, 1989.

[Wal91] R.A. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems*, Kluwer Academic Publishers, Boston, MA, 1991.

[Wey87] C.L. Wey, M.K. Vai, and F. Lombardi, "On the Design of a Redundant Programmable Logic Array (RPLA)," *IEEE Journal of Solid-State Circuits*, Vol. 22, No. 1, pp. 114-117, 1987.

[Wil73] M.J.Y. Williams and J.B. Angell, "Enhancing Testability of Large Scale Integrated Circuits Via Test Point and Additional Logic," *IEEE Transactions on Computers*, Vol. 22, No. 1, pp. 46-60, 1973.

[Yeu92] A. Yeung and J. Rabaey, “A Data-Driven Architecture for Rapid Prototyping of High Throughput DSP Algorithms,” *IEEE VLSI Signal Processing Workshop*, pp. 225-234, 1992.