Computer Aided Design of Fault-Tolerant Application Specific Programmable Processors

Ramesh Karri, Kyosun Kim, and Miodrag Potkonjak

Abstract—Application Specific Programmable Processors (ASPP) provide efficient implementation for any of m specified functionalities. Due to their flexibility and convenient performance-cost trade-offs, ASPPs are being developed by DSP, video, multimedia, and embedded IC manufacturers. In this paper, we present two low-cost approaches to graceful degradation-based permanent fault tolerance of ASPPs. ASPP fault tolerance constraints are incorporated during scheduling, allocation, and assignment phases of behavioral synthesis. Graceful degradation is supported by implementing multiple schedules of the ASPP applications, each with a different throughput constraint. In this paper, we do not consider concurrent error detection. The first ASPP fault tolerance technique minimizes the hardware resources while guaranteeing that the ASPP remains operational in the presence of all *k*-unit faults. On the other hand, the second fault tolerance technique maximizes the ASPP fault tolerance subject to constraints on the hardware resources. These ASPP fault tolerance techniques impose several unique tasks, such as fault-tolerant scheduling, hardware allocation, and application-to-faulty-unit assignment. We address each of them and demonstrate the effectiveness of the overall approach, the synthesis algorithms, and software implementations on a number of industrial-strength designs.

Index Terms—Application specific programmable processors, fault tolerance, graceful degradation, behavioral synthesis.

1 Introduction

Modern applications require high performance, low power, and inexpensive multiple functionalities. Furthermore, multiple standards (e.g., CDMA and TDMA in wireless communications and NTSC, PAL, and SECAM for television) may exist for the same application. Whereas multiplicity of standards, diverse quality-of-service offerings, and the rapidly changing deployment scenarios mandate a need for flexibility, portability, and mobility necessitate low power operation. However, neither general-purpose processors nor dedicated special-purpose processors by themselves can offer these diverse implementation properties. Consequently, several major processor manufacturers, including Fujitsu [10] (with their 86k line of programmable processors) and Motorola [30] (with their application-specific programmable DSP processors), are offering a comprehensive line of application-specific programmable processors (ASPPs) that preserve all the advantages of special-purpose processors while retaining the cost and flexibility provided by general-purpose processors. These market trends confirm the rapidly growing need for efficient synthesis techniques for ASPP designs. We focus on the reconfigurability of datapath-intensive application-specific computation since we are targeting DSP, video, control, and communication applications. Further, wireless communication (TDMA, CDMA), television (NTSC, PAL, SECAM, HDTV), video compression (JPEG, MPEG), audio compression (AC-3, AC-97), etc. are standardized. Therefore, the functionality of commercial electronic products is limited to a minor variation of these standards.

An application specific programmable datapath, shown in Fig. 1, implements a seventh order IIR filter (IIR7) and two schedules of the volterra filter (VOLTERRA).[1] The flexibility and redundancy inherent in such ASPP designs are an excellent source for providing permanent fault tolerance with low overhead. For example, in Fig. 1, if all units are operational, either IIR7 or VOLTERRA can be executed. In the presence of a single faulty unit, one of the schedules of VOLTERRA is still operational, as shown in Table 1.

An ASPP can be configured to execute only those applications that do not use the faulty units. Alternatively, when only limited repair is economically feasible, one can invoke an application on the ASPP requiring the smallest number of repair steps [31]. Statistically, we still have an IC which realizes all of the required functionalities, but with reduced costs for yield enhancement repair [25].

From a design tools perspective, the focus has been on synthesizing a specific implementation for a given computation which meets a combination of design constraints, such as throughput, latency, and power, while optimizing a primary design goal, area. We developed a behavioral synthesis tool that synthesizes ASPP designs starting with the specifications of any m functionalities. We use the architectural flexibility of an ASPP to develop two approaches for graceful degradation-based permanent fault tolerance. We show how these ASPP fault tolerance constraints can be incorporated during the scheduling,

- *R. Karri is with the Department of Electrical Engineering, Polytechnic University, Brooklyn, NY 11201. E-mail: ramesh@india.poly.edu.*
- *K. Kim is with Samsung Electronics, Yonin-City Kyungki-Do, 449-900 Korea. E-mail: kkim99@samsung.co.kr.*
- *M. Potkonjak is with the Computer Science Department, 3532 Boelter Hall, University of California at Los Angeles, Los Angeles, CA 90095-1596. E-mail: miodrag@cs.ucla.edu.*

Manuscript received 19 Apr. 1999; accepted 25 July 2000.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 112582.

1. This has been synthesized automatically using the reported synthesis system.

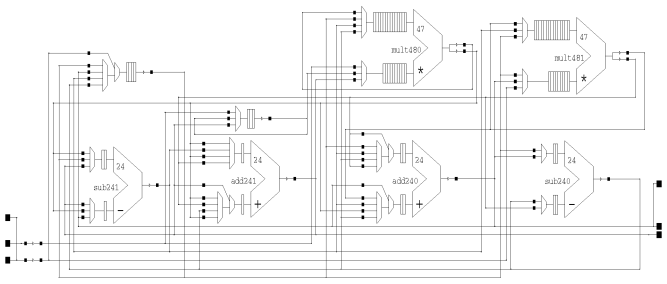


Fig. 1. ASPP implementing seventh order IIR and VOLTERRA filters.

allocation, and assignment phases of behavioral synthesis. We assume that a concurrent fault detection mechanism has already been implemented and, hence, focus only on incorporating graceful degradation. Graceful degradation is supported by implementing multiple schedules of the ASPP applications, each with a different throughput constraint. In the first technique, the hardware resources are minimized while guaranteeing that the ASPP remains operational in the presence of all k-unit faults. On the other hand, in the second technique, the fault tolerance of an ASPP is maximized subject to constraints on the hardware resources. Fault-tolerant ASPP synthesis imposes several unique tasks such as fault-tolerant scheduling, hardware allocation, and application-to-faulty-unit assignment. We address each of them and demonstrate the effectiveness of the overall approach, the synthesis algorithms, and software implementations on a number of industrial-strength designs.

The rest of the paper is organized in the following way: We first briefly survey the related work along several dimensions. Next, we will discuss the hardware and fault models. In Sections 5 and 6, we formulate the two fault-tolerant ASPP synthesis problems, describe the synthesis algorithms, and present experimental results. In Section 7, we conclude by summarizing the results, outlining the future directions in the design of fault-tolerant ASPPs.

2 Related Research

The most relevant related work can be traced along three lines of research and development: reconfigurable hardware, behavioral synthesis, and fault tolerance techniques.

Field programmable gate arrays (FPGA) are being extensively used for dynamic reconfiguration [7], [8], [16], [43]. FPGA-based reconfigurable architectures, algorithms suitable for run-time reconfiguration and implementation of proof-of-concept designs, have been investigated [7], [16]. This has resulted in a dynamic instruction set computer and FPGA-based implementation for the traveling salesman problem, mean filtering, and edge detection. FPGA techniques for on-the-fly adaptation of a video signal processor have been investigated by the dynamic computing project [43]. Since it takes a few milliseconds to download a hardware netlist onto an FPGA, it entails significant performance overhead and, hence, is limited to applications with infrequent context switching. Although dynamically programmable gate arrays (DPGAs) store multiple personality vectors to reduce the reconfiguration time, their memory requirement increases to 33 percent (compared to 10 percent in FPGAs) [8].

TABLE 1
Functional Units Used by IIR7 and VOLTERRA Applications

		IIR7	VOLTERRA	
type	instance		1	2
+	add240	√		√
	add241	√	√	
-	sub240	√		
	sub241	√		
*	mult240	√	√	
	mult241	√		√

There are a number of papers which describe fault tolerance schemes for FPGAs. For example, Doumar et al. [11] proposed a scheme where fault tolerance is achieved by shifting the configuration data inside the FPGA. Dutt et al. [12] achieve fault tolerance by using incremental rerouting in the FPGA. Meyer et al. [24] developed a greedy algorithm for enhancing fault tolerance of one-time programmable FPGAs. Finally, Lach et al. [19] presented both methodology for design of tiling-based fault-tolerant FPGA systems as well as a survey of earlier efforts on designing fault-tolerant FPGA-based systems.

Recently, application specific instruction sets processors (ASIP) received a great deal of attention. An ASIP has a programmable architecture tuned to an application class. Identifying an optimal instruction set to improve the applications in the selected class, subject to area and power dissipation constraints, has been addressed by several researchers [41], [21], [6], [22], [29]. Analysis tools to profile the applications in a class to select an optimal instruction set [41], [21] and synthesis tools to design an architecture to efficiently execute the instructions in this instruction set [6], [22], [29] have been developed. Although ASIPs provide greater flexibility than ASICs, this comes at the expense of low performance, high cost and power, and a need for compilation support. Although synthesis of application-specific programmable processors [5], [15] and application-specific instruction sets processors (ASIP) [14], [22], [29] have been receiving a great deal of attention, none of these ASIP and ASPP design methodologies address fault-tolerant design.

Behavioral synthesis has been an active area of research for more than two decades [9], [13], [23] and numerous outstanding systems have been built targeting both data path-oriented and control-oriented applications [23], [33]. Behavioral synthesis traditionally has addressed synthesis and optimization of a single CDFG for sampling rate, area, and, more recently, power and test hardware overhead minimization [9], [23].

Recently, a few efforts have been reported on behavioral synthesis techniques for fault-tolerant designs. Orailoglu and Karri [27] presented scheduling, assignment, and transformation-based methods for fault tolerance against transient faults. Guerra et al. [15] presented the first work which concentrates on permanent faults. They showed how fault tolerance achieved using a set of spare units can be used for yield and productivity enhancement. Recently, Iyer

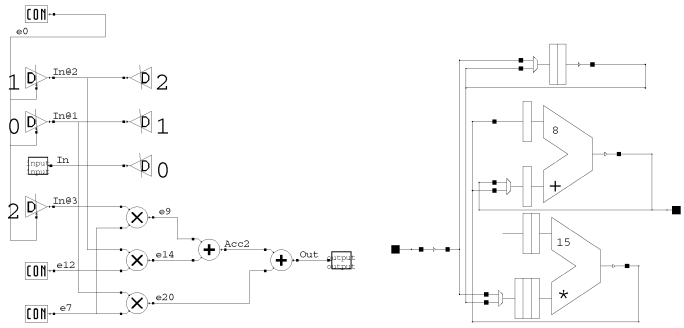


Fig. 2. (a) A hierarchical control data flow graph and (b) microarchitecture of a second-order FIR filter.

et al. [17] introduced a method which explores trade-offs between performance and yield. Antola et al. [2], [3] developed high level synthesis techniques for concurrent and semiconcurrent error detection in data paths. Automatic synthesis of self-recovering microarchitectures has been previously addressed. An algorithm that intertwines checkpoint insertion and scheduling (of operations in the input algorithm to clock cycles) to synthesize self-recovering microarchitectures for supporting fault-recovery in hardware was first presented in [18], [26]. More recently, [4], [36] presented algorithms for recovery point insertion in recoverable microarchitectures that minimizes the number of rollback points given constraints on the number of registers and maximum number of time steps between any two rollback points (the stride). For efficient concurrent detection, [37] showed how fault-security constraints can be incorporated during high level synthesis.

3 Preliminaries

3.1 Computational Model

Our computational model for a single application is homogeneous synchronous data flow [20], a special case of the data flow process network family of computational semantics. The model assumes a periodic computation done on an incoming semifinite stream of data along the time axis. Within this model, a task is represented as a hierarchical Control Data Flow Graph $G(N, E, T)$ (or CDFG), with nodes N representing the flow graph operations and the edges E and T, respectively, the data and timing dependences between the operations. Note that the control dependences are subsumed by the timing control dependences. A CDFG of a 2nd-order FIR filter is shown in Fig. 2a. The homogeneous synchronous data flow model provides semantics for numerous behavioral synthesis systems targeting numerically intensive applications [42]. Many of most popular DSP, video, continuous media, communication, control, and graphics applications follow the selected computational model.

3.2 Hardware Model

In modern designs, a variety of register file models have been used [9]. From among them, we have selected the dedicated register file hardware model for modeling at the structural register-transfer (RT) level. This model clusters all registers in register files and each file is then connected only to the inputs of the corresponding execution units. Fig. 2b shows a microarchitecture which is synthesized from the the second-order FIR filter CDFG shown in Fig. 2a. An important benefit of the chosen hardware model is that it reduces the interconnect at the expense of additional registers. This trade-off is particularly important for modern and future submicron technologies. A more practical reason for using the dedicated register file model is that we are using the behavioral synthesis utility tools from the Hyper behavioral synthesis system [42]. Hyper provides tools for translations from a high-level applicative language to the internal CDFG format, estimation subroutines, simulations, and hardware mapping facilities, as well as access to a large number of real-life design specifications. Other hardware models can be directly addressed using the proposed methodology and synthesis algorithms, although appropriate minor modifications to the software are required to address different interconnect schemes.

There are three types of controllers that are suitable for ASPP designs. *Programmable Controllers* [28] often bring a somewhat large implementation area overhead and a limited degradation in performance. However, it provides flexibility not only for ASPP, but also for the additional introduction of new functional specifications for a given datapath. *Off-chip controllers* are flexible in that they can be replaced as necessary since they are located on a separate integrated circuit. A number of high performance datapath intensive chips have been designed using this option [34]. The same drawbacks and advantage as in the case of programmable controller option hold. *Composed controller* is a third alternative wherein the controller is located on-chip and is the composition of all possible control configurations that are required. Its effectiveness depends on how well several different (but often very similar) controllers can be merged using logic synthesis tools. Of these, we use the composed controller for our ASPP implementations.

3.3 Fault Model

We assume a widely used single stuck-at fault model [1]. Before the graceful degradation step in an ASPP can be invoked, the fault in the ASPP should be detected. Any off-line testing and diagnosis scheme, such as full-scan, combinational ATPG, and BIST, can be used to detect faults. The ASPP fault tolerance techniques can tolerate faults that occur either in an execution unit or in a register

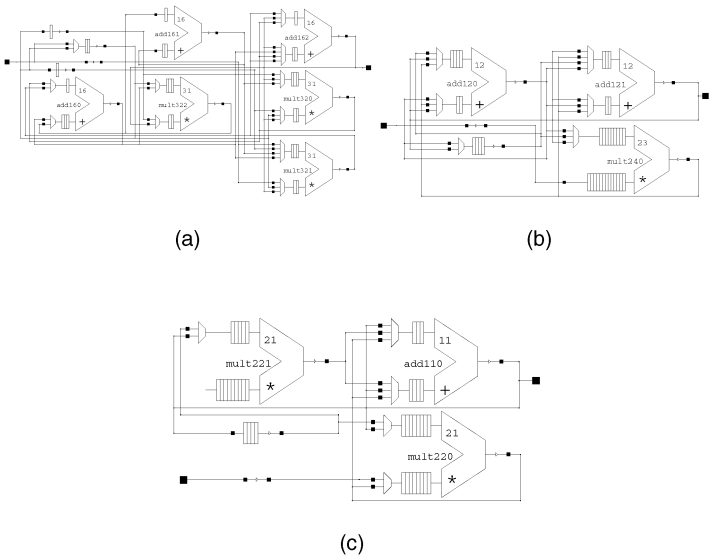


Fig. 3. Microarchitectures of dedicated ASIC implementations. (a) ADAPT. (b) CASCADE. (c) IIR8.

file or in the interconnect. A fault in a register file prevents its corresponding execution unit from receiving data and, thus, has the same effect as a fault in the execution unit. A faulty interconnect is tolerated by treating it as a failure in the execution unit at its data-sending connection. To make this possible, any two functional units that have a data transfer between them are connected by a dedicated bus. To reduce the interconnection overhead, all point-to-point buses emanating from a functional unit are merged without violating the fault tolerance constraints. We assume that the controller is fault-free. However, since the area of the controller is usually only a few percent (1-3) of the designs, it can be easily duplicated with a negligible impact on the ASPP area.

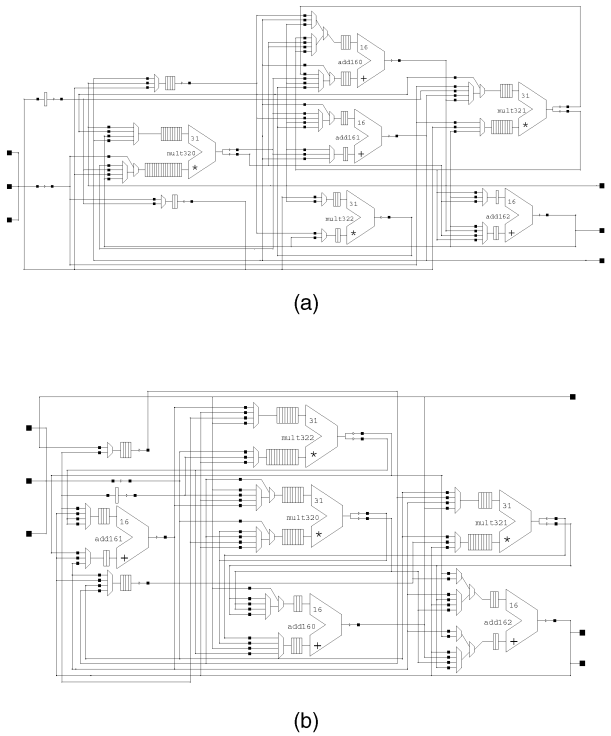


Fig. 4. Microarchitectures of basic ASPP and fault-tolerant ASPP implementations. (a) Basic ASPP. (b) Fault-tolerant ASPP.

4 Fault-Tolerant ASPP

We will motivate a behavioral synthesis approach for incorporating *k*-unit fault tolerance into ASPPs and discuss the relationship between ASPP designs, the degree of fault tolerance, and area and performance overheads. Consider an integrated digital signal processing system consisting of dedicated coprocessors for three signal processing applications—an adaptive filter (ADAPT) with latency of eight, a four-stage cascaded quadratic filter (CASCADE) with latency of 16, and an eighth order infinite impulse response filter (IIR8) with latency of 19, as shown in Fig. 3a, Fig. 3b, Fig. 3c.

ADAPT requires three adders and three multipliers, CASCADE requires two adders and one multiplier. Similarly, IIR8 requires two multipliers and one adder, although the word length is different from others. These dedicated ASICs cannot tolerate any functional unit failures. Consider an ASPP that can be *configured to run any one of these applications at any given time*. The resulting ASPP requires only three adders and three multipliers, as shown in Fig. 4a. The layouts of the dedicated coprocessors and the ASPP with their areas annotated are shown in Fig. 5a, Fig. 5b, Fig. 5c, Fig. 5d. All layouts are scaled to reflect their relative sizes. The area occupied by the dedicated coprocessors is 97.55 mm^2,[2] 35 percent larger than the ASPP area.

What are the potential benefits of such a multifunctional processor? Since six functional units are used, there are ${}^6C_1$ possible ways that a single functional unit can fail. Toward illustrating the key concepts and importance of fault tolerance in ASPP designs, consider the application-to-faulty-unit assignment summarized in Table 2. Prefixes A and M stand for adder and multiplier, respectively. The numbers after them are the instances.

2. 0.6 micron SCMOS technology library was used.

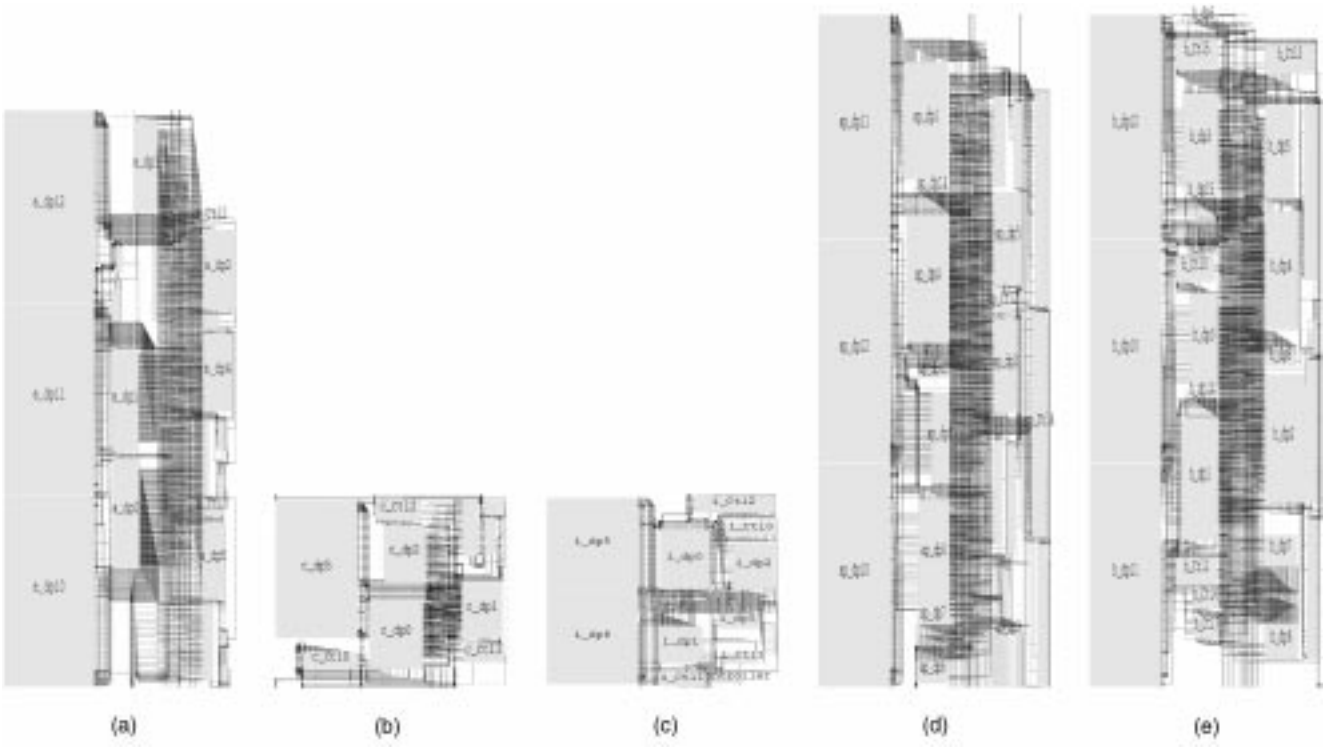


Fig. 5. Layout implementations of dedicated ASICs and ASPPs. (a) ADAPT (57.43 mm^2). (b) CASCADE (19.33 mm^2). (c) IIR8 (20.79 mm^2). (d) Basic ASPP (72.08 mm^2). (e) Fault-tolerant ASPP (72.43 mm^2).

CASCADE and IIR8 tolerate three single-unit faults each. The basic ASPP of Fig. 4a can tolerate only four out of the six one-unit faults. This is because faults in A2 and M2 are tolerated by both applications, while faults in A0 and M0 are not tolerated by either application. Consider the application-to-faulty-unit assignment shown in Table 3. All one-unit faults are tolerated either by CASCADE or by IIR8. They can also tolerate three two-unit faults and one three-unit fault. The corresponding fault-tolerant ASPP microarchitecture and layout are shown in Figs. 4b and 5e, respectively. Although there is an increase in interconnect, the area overhead for one-unit fault tolerance with respect to the basic ASPP implementation is less than 1 percent. For this fault-tolerant ASPP, the following reconfiguration strategy can be adopted: If all modules are operational, configure the ASPP to implement ADAPT. If adder A0, multiplier M0, or multiplier M1 is faulty, configure the ASPP to implement CASCADE. Similarly, if adder A1, adder A2, or multiplier M2 are faulty, configure the ASPP to implement IIR8.

The case of two-unit fault tolerance is more complex. There are 15 different ways in which two-units can fail. A CASCADE schedule can tolerate only one of three possible two-multiplier-unit faults. Similarly, an IIR8 schedule can tolerate one of the three possible two-adder-unit faults. To tolerate all possible two-unit faults, at least three schedules of CASCADE and three schedules of IIR8 (each with different application-to-faulty-unit assignments) are required. A two-unit tolerant hardware allocation is shown in Table 4. The two-unit faults (A1, M2), (A0, M1), and (A2, M0) are tolerated by more than one schedule.

The ASPP interconnect and controller overhead increases with the number of schedules implemented on it. The number of implemented schedules can be reduced without compromising the *k*-unit fault tolerance of the ASPP by increasing the latency of the application schedules. If the latencies of CASCADE and IIR8 are increased to 18 and 21, respectively, their hardware requirements decrease to one adder and one multiplier. The new fault-tolerant hardware allocation is summarized in Table 5. The fault-tolerant ASPP microarchitectures with and without graceful degradation are shown in Fig. 6a and Fig. 6b, respectively. Observe the reduction in the interconnect due to graceful degradation.

TABLE 2
One-Unit Fault Tolerance of the Basic ASPP

Appl'n	H/W Allocation		Tolerate
	Add	Mult	Faults in
ADAPT	A0, A1, A2	M0, M1, M2	–
CASCADE	A0, A1	M0	A2, M1, M2
IIR8	A0	M0, M1	A1, A2, M2

TABLE 3
One-Unit Fault Tolerance of the fault-Tolerant ASPP

Appl'n	H/W Allocation		Tolerate
	Add	Mult	Faults in
ADAPT	A0, A1, A2	M0, M1, M2	–
CASCADE	A1, A2	M2	A0, M0, M1
IIR8	A0	M0, M1	A1, A2, M2

TABLE 4
Two-Unit Fault-Tolerant Hardware Allocation

Appl'n	H/W Allocation		Tolerate
	Add	Mult	Faults in
ADAPT	A0-2	M0-2	–
CASCADE	A1, A2	M2	(M0, M1), (A0, M0), (A0, M1)
CASCADE	A0, A1	M1	(M0, M2), (A2, M0), (A2, M2)
CASCADE	A0, A2	M0	(M1, M2), (A1, M1), (A1, M2)
IIR8	A2	M0, M1	(A0, A1), (A0, M2), (A1, M2)
IIR8	A1	M0, M2	(A0, A2), (A0, M1), (A2, M1)
IIR8	A0	M1, M2	(A1, A2), (A1, M0), (A2, M0)

5 Guaranteeing K-Unit Fault Tolerance of ASPPs

K-unit fault tolerance can be guaranteed by combining the inherent redundancy of ASPPs with judicious application-to-faulty-unit assignment and application latency determination.

Problem Statement. *Given an underlying hardware model and N applications, each with its execution time bound, synthesize a high-performance and minimum area design so that any one of these N applications can be executed at any given time, and, for any k-unit failure, the ASPP design is still operational (i.e., at least one of these N applications is still working).*

The design flow is outlined in Fig. 7. Initially, the applications are bundled together based on their hardware and structural similarity. In the process, the area overhead is minimized. Following application bundling, the latency determination phase is entered, wherein the latencies of the individual applications are determined while ensuring the desired k-unit fault tolerance. Next, the hardware allocations of applications are matched to the k-faulty-unit combinations using a branch and bound technique. This step determines the hardware not usable by a given schedule of an application. Based on this information, ASPP allocation, assignment, and scheduling algorithms are invoked on the applications in a bundle. The fault-tolerant ASPP synthesis trajectory is completed by invoking the Hyper hardware mapper and layout generator. Fault tolerance constraints are incorporated during the highlighted phases in the design trajectory.

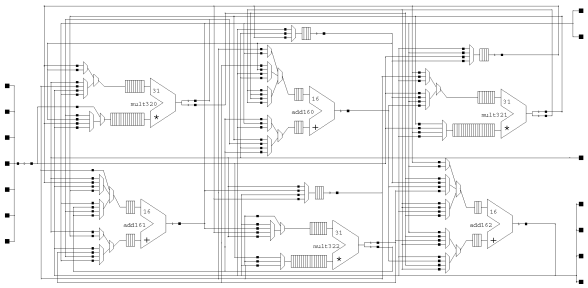
The input to the Hyper behavioral synthesis system is a design specified using the SILAGE language [42]. The input is translated into the hierarchical control data flow graph (CDFG) format, where each node corresponds either to arithmetic, logic, memory access, or input/output transfer operation and each edge denotes either data or control flow dependency. When the Hyper's estimation subroutines are applied to the CDFG, the resulting requirements on datapath components are used as initial resource allocation. After that, there is a provision for applying a number of transformations, such as pipelining, retiming, associativity, and common subexpression elimination. Next, clock cycle length and module selections are performed. The two final mandatory steps, before hardware mapping and layout generation using the Lager silicon compiler, are constraint-driven scheduling and assignment. For almost all synthesis steps, the user can specify desired run time/quality of results trade-off. The default option uses $O(n^2)$ heuristics and usually requires less than one minute for all the tasks [42].

TABLE 5
Trade-Off between the Number of Schedules and Performance Degradation

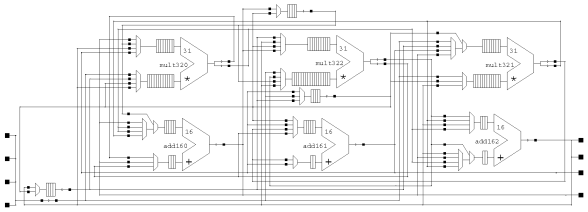
Appl'n	H/W Allocation		Tolerate
	Add	Mult	Faults in
ADAPT	A0-2	M0-2	–
CASCADE	A0	M0	(A1, A2), (A1, M1), (A1, M2) (M1, M2), (A2, M1), (A1, M2)
CASCADE	A1	M1	(A0, A2), (A0, M0), (A0, M2) (M0, M2), (A2, M0), (A2, M2)
IIR8	A2	M2	(A0, A1), (A0, M0), (A0, M1) (M0, M1), (A1, M0), (A1, M1)

5.1 Latency Determination

The latency of each application in the bundle is determined so as to ensure that the resulting ASPP processor can



(a)



(b)

Fig. 6. Microarchitectures of two-unit fault-tolerant ASPPs. (a) Without graceful degradation. (b) With graceful degradation.

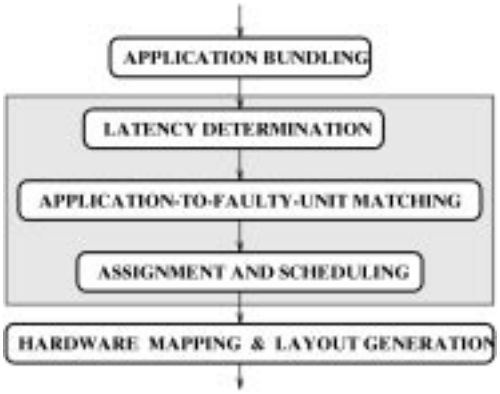


Fig. 7. Fault-tolerant ASPP synthesis flow.

tolerate all k-unit faults. K-unit fault tolerance is ensured by tuning the hardware utilization of individual applications. Decreasing the system throughput (by increasing the latency) reduces hardware utilization and increases the fault tolerance capability of an application.

As a first step, all possible k-unit faults are collapsed into smaller sets of faults based on the type of hardware units that fail. For example, in a design that uses adders and multipliers, the two-unit faults are collapsed into the following sets:

- Two-multiply faults,
- Two-adder faults,
- One-adder-One-multiply faults.

Collapsing faults into smaller sets can be represented by the set of tuples:

$$R = \{(r_1, r_2, \ldots r_n) | \sum_{i=1}^{n} r_i = k\},$$

where, $r_1, r_2, \ldots, r_n$ are the number of faulty units of type $1, 2, \ldots, n$, respectively. For an application bundle with n hardware types and k-unit fault tolerance, the number of tuples in the set R can be computed by the recursion:

$$| R | = f(n,k) = \sum_{j=0}^{\min(H_n,k)} f(n-1, k-j)$$
$$f(1,j) = 1, 1 \leq j \leq k$$
$$f(i,1) = i, f(i,0) = 1, 1 \leq i \leq n.$$

The set R, together with the unused hardware distributions of the individual applications (obtained using a similar recursion), is used to quickly verify if the selected latencies for the applications can ensure k-unit fault tolerance. Toward this end, it is checked if, for each tuple in the set R, there is at least one application in the bundle whose unutilized hardware units for each type of hardware is larger than the number of faulty units of that type in the tuple. If not, the latencies of selected applications are increased. Although increasing latency succeeds for low k, additional hardware units are added to guarantee k-unit fault tolerance.

Schedules in ASPP	k-unit fault combinations				
	C_1	C_2	C_3	...	C_n
A_1S_1	x	x			
A_1S_2		x		x	
..					
A_1S_{n1}		x		x	
A_2S_1	x	x			
A_2S_2			x		x
..					
A_2S_{n2}	x		x		
..					
A_mS_1			x		
A_mS_2		x			x
..					
A_mS_{nm}		x			

Fig. 8. Application-to-faulty-unit matching to guarantee k-unit tolerance.

5.2 Matching Applications with Faulty Units

A single schedule for each application may not be sufficient to implement k-unit fault tolerance. This is because, although this single schedule may cover a tuple in the set R, it may not tolerate all the k-unit fault combinations that the tuple represents. If, out of the H_t hardware units of type t available for the bundle, s_t^i are not utilized by the application i, then $\prod_{t=1}^{n} {}^{H_t}C_{s_t^i}$ schedules of the application may be necessary. The problem is to match the schedules of the applications with the k-faulty-unit combinations so that:

- There is at least one schedule that can operate in the presence of a k-unit fault,
- At least one schedule of each application is implemented, and
- The total number of implemented schedules is minimized.

The problem is illustrated in Fig. 8. Each column in the two-dimensional table corresponds to a k-faulty unit combination (denoted as $C_1, C_2, \ldots, C_n$) and each row corresponds to a schedule of an application (denoted as $A_1S_1, A_1S_2, \ldots, A_mS_{nm}$). If a schedule of an application covers a k-faulty-unit combination, the corresponding cell is marked with an x. The objective is then to ensure that there is at least one x in each column while minimizing the number of rows. This problem can be transformed to the vertex covering problem [35] by identifying each of the k-faulty-unit combinations as a vertex and each of the possible application schedules as an edge.

A branch and bound technique outlined in Fig. 9 is used to solve this problem. The `MinimumSetOfSchedules` is the current best solution that tolerates all the combinations of k faulty units and includes all the schedules, initially. The `CurrentSetOfSchedules` tolerates only some of the combinations of k-faulty units and is initially null and grows when the schedule inclusion branch in Step 9 is taken. The branching step is invoked when a candidate `schedule` is either included into or excluded from the `CurrentSetOfSchedules` by the recursive calls in lines 10 and 13. `schedules` are selected one after the

```
MinimumMatch(CurrentSetOfSchedules, ListOfSchedules,
                                          CurrentCover)
{
1:  if (( schedule ← car(ListOfSchedules)) = φ) return;
2:  appl ← GetApplication(schedule);

    /* Bound the Schedule Inclusion Branch */
3:  if ( |CurrentSetOfSchedules| + Σ_{i=appl+1}^{#Applications} LowerBound[i]
        + 1 < | MinimumSetOfSchedules | ) {
4:      NewCover ← CurrentCover ⋁ Cover(schedule);
5:      if (NewCover covers all k faulty unit combinations) {
6:          MinimumSetOfSchedules←CurrentSetOfSchedules
                                          ∪ {schedule};
7:          return;
        }
8:      if ( | NewCover | > |CurrentCover | ) {
9:       CurrentSetOfSchedules ← CurrentSetOfSchedules
                                          ∪ {schedule};
10:     MinimumMatch(CurrentSetOfSchedules,
                        cdr(ListOfSchedules), NewCover);
11:     CurrentSetOfSchedules ← CurrentSetOfSchedules
                                          - {schedule};
        }
    }
    /* Bound the Schedule Exclusion Branch */
12: if ( | CurrentSetOfSchedules ∩ {schedules of appl-1 } |
        ≥ LowerBound[appl-1] )
13:     MinimumMatch(CurrentSetOfSchedules,
                        cdr(ListOfSchedules), CurrentCover);
    }
```

Fig. 9. Finding the minimum set of schedules tolerating all *k*-unit faults.

other (Step 1) from the `ListOfSchedules`. Associated with each `schedule` is a vector identifying the *k*-faulty-unit combinations that are tolerated (covered) by it. `Cover()` returns this vector. The set union of the coverage vectors of all schedules in the `CurrentSetOfSchedules` is the `CurrentCover`.

Upper and lower bounds on the number of schedules are used to prune the solution branches. For example, the cardinality of the `MinimumSetOfSchedules` is an upper bound on the number of schedules. If the cardinality of the `CurrentSetOfSchedules` is greater than that of the `MinimumSetOfSchedules`, this schedule inclusion branch and its branches are pruned. For each tuple in the set R, there are $\prod_{t=1}^{n} {}^{H_t}C_{r_t}$ combinations of k faulty units. The sets of *k*-unit combinations represented by tuples are disjoint. Based on this observation, the `LowerBounds` on the number of schedules for each application are determined as follows: Let the faults represented by a tuple $(r_1, r_2, \ldots, r_t, \ldots, r_n)$ in the set R be covered by application i. A lower bound on the required number of schedules of application i is $\prod_{t=0}^{n} \lceil {}^{H_t}C_{r_t} / {}^{s_t^i}C_{r_t} \rceil$, where H_t, s_t^i, and r_t have been previously defined. If the tuple is covered by more than one application, the aggregate number of schedules of the applications which cover the tuple must be considered. Some of the schedules in the `ListOfSchedules` that have not been visited are absolutely necessary to satisfy this lower bound requirement. Hence, these schedules must also be added to the cardinality of the `CurrentSetOfSchedules` in line 3 when comparing with the upper bound. In line 12, if the number of schedules corresponding to an application is less than the `LowerBound`, all successive branches are pruned. `car()` and `cdr()` are two lisp-like functions used to return the first and remaining elements in a list, respectively.

5.3 Assignment and Scheduling

At the end of the matching phase, for each schedule of an application, hardware units that are excluded from its allocation are finalized. This determines the hardware units that are available for use by a given schedule of an application. An assignment and scheduling algorithm is invoked on each of the schedules of the applications using its usable hardware allocation. Applications can be synthesized in any order as the hardware requirements are determined prior to this step. The resulting ASPP is then mapped and synthesized using the Hyper back-end system.

5.4 Evaluation of Fault Tolerance Constrained Synthesis

The fault-tolerant ASPP synthesis techniques proposed in this section were validated on the set of DSP, video, control, and communication applications summarized in Table 6. For each application, columns 2-5 show the number of nodes, the number of edges, the word length, and the critical path, respectively. The input latency for each application is shown in column 6. The next four columns give the hardware allocation, assuming that additions and subtractions are carried out on distinct adders and subtractors, respectively. Assuming that additions and subtractions are carried out by add-subtract units will yield a different set of results. The column titled "reg" shows the number of registers used in the implementation. The numbers in parentheses are the register counts for constants. The hardware utilization of each type is shown in the next three columns. The last column reports the area in mm^2 when the application is implemented as a dedicated ASIC. This is used to evaluate the area overhead of ASPPs.

The set of applications shown in Table 6 is bundled into ASPPs with the objective of minimizing the ASPP overhead for each IC. The resulting bundles were synthesized to guarantee *k*-unit fault-tolerant ASPPs for $k = 1, 2$.

The results of one-unit fault-tolerant ASPP synthesis for the eight application bundles are summarized in Table 7. The number of schedules used in the ASPP processor is shown in column 2. As the number of schedules increases, so does the number of registers for constant coefficients and the interconnection requirement. These are the major sources of area overhead. The next four columns are the hardware allocations used in synthesizing the ASPP processor. The area overhead vis-a-vis a dedicated implementation of the largest application is summarized in the last column. The areas reported are all in mm^2. The area overhead is 14.5 percent on average and varies from 5 percent to 26 percent. The sources of the overhead are 1) the increased interconnect requirements and 2) the coefficient registers in the current dedicated register file hardware model. The low overhead is due to the inherent redundancy and the application-to-faulty-unit matching.

When supporting multiple unit fault tolerance, straightforward replication entails significant area or performance overhead. In contrast, the proposed approach can support

TABLE 6
Example Applications to Validate Fault-Tolerant ASPP Synthesis Techniques

application	|N|	|E|	wl	cp	t	allocation +	-	*	reg	utilization +	-	*	area (mm^2)
ARAI	51	63	22	8	10	7	5	2	43(13)	32	35	65	34.77
CASCADE	34	47	12	10	10	4	0	2	39(13)	40	-	65	11.34
DIR	124	138	22	9	22	4	0	4	83(38)	63	-	68	53.05
FFT8	30	31	16	5	6	3	3	1	22(1)	55	55	33	11.21
FIR20	32	42	16	3	12	5	0	2	46(10)	33	-	41	18.03
GM1M	20	27	20	14	14	2	1	2	24(7)	32	21	25	21.81
IIR7	29	39	24	10	10	2	2	2	35(10)	35	35	50	32.26
IIR8	39	57	11	9	12	3	0	2	47(18)	66	-	75	9.97
LEE	57	66	22	10	12	5	3	2	32(11)	35	33	83	31.51
MCM	102	111	22	9	20	4	4	2	68(17)	38	34	75	35.26
PR1	56	65	22	8	10	3	3	4	42(17)	43	43	55	52.16
PR2	66	73	22	9	15	2	2	3	34(12)	43	43	71	38.50
VOLTERRA	30	39	24	12	12	1	0	2	28(9)	83	-	66	28.47
WANG	56	65	22	8	14	4	3	2	44(17)	33	33	78	31.64
WAVELET	53	65	16	14	15	3	1	2	57(14)	42	46	47	18.68
WDF5	23	29	16	12	12	2	2	1	25(6)	25	41	50	9.80
WDF7	31	37	22	12	12	2	4	2	33(7)	33	29	33	28.68

TABLE 7
Guaranteeing One-Unit Fault Tolerance

#	Application Bundles	# sch'	allocation +	-	*	reg	area (mm^2)	over head
1	{CASCADE, DIR}	5	4	0	4	118(68)	58.86	10.95%
2	{IIR7, VOLTERRA}	3	2	2	2	50(21)	33.97	5.30%
3	{WANG, WDF5}	4	3	3	2	65(30)	35.35	11.73%
4	{FFT8, LEE}	4	4	3	2	44(15)	36.10	14.57%
5	{MCM, WDF7}	3	4	5	2	95(27)	40.26	14.18%
6	{ARAI, FIR20, GM1M}	4	5	4	2	89(37)	40.35	16.05%
7	{CASCADE, PR2}	6	3	2	3	81(38)	45.37	17.84%
8	{IIR8, PR1, WAVELET}	4	3	4	4	153(89)	65.60	25.77%

two-unit fault tolerance with negligible performance penalty and modest area overhead. The results are summarized in Table 8 for the previously used application bundles. On an average, the area overhead is 42.2 percent and varied from a minimum of 14 percent to a maximum of 69 percent. The increase in area overhead is mainly due to additional units that were added during the latency determination phase. This is one of the first synthesis systems that has demonstrated the feasibility of automatically synthesizing multiple-fault-tolerant designs with modest area overheads. The one-unit and two-unit fault-tolerant ASPPs have significantly superior hardware utilization characteristics when compared to the less than 5 percent utilization of general purpose processors [28].

6 Maximizing Fault-Tolerance of an ASPP

Consider the scenario wherein *k*-unit fault tolerance need not necessarily be guaranteed. Fault tolerance can still be maximized using the available hardware resources.

Problem Statement. *Given an underlying hardware model, N applications, each with its execution time bound, and an overall hardware constraint, synthesize an ASPP design so that any of these N applications can be executed at any given time and the fault tolerance of the ASPP design is maximized.*

Initially, an ASPP allocation, assignment, and scheduling step is carried out [40] to determine the exact hardware allocation for the entire design and for each application in the design. Starting with this definitive hardware allocation, the fault tolerance of the ASPP can be maximized by reallocating and reassigning the available units in this hardware allocation to the applications. This is illustrated in Fig. 10. $A_1, A_2, \ldots A_m$ are the m applications implemented in the ASPP. $H_1^1, H_2^1, \ldots, H_{n1}^1$ are the possible hardware allocations for application A_1 and $H_1^m, H_2^m, \ldots, H_{nm}^m$ are the possible hardware allocations for application A_m.

If the hardware allocation H_1^1 is used to implement application A_1, then three one-unit faults (C_1^1, C_2^1, C_n^1) and one *k*-unit fault (C_2^k) can be tolerated. The fault tolerance maximization problem is to *choose one hardware allocation for each application so that the number of distinct faulty unit combinations that are tolerated is maximized*.

6.1 Incremental Fault-Tolerant Hardware Allocation

We propose an incremental fault-tolerant (IFT) hardware allocation algorithm. The fault tolerance of the ASPP is

TABLE 8
Guaranteeing Two-Unit Fault Tolerance

Application Bundles	# sch'	allocation				area (mm^2)	over head
		+	-	*	reg		
{CASCADE, DIR}	7	4	0	4	139(81)	61.29	13.72%
{IIR7, VOLTERRA}	4	3	2	3	84(37)	50.59	56.82%
{WANG, WDF5}	4	3	3	3	91(36)	42.58	34.58%
{FFT8, LEE}	7	4	4	3	52(16)	45.27	43.67%
{MCM, WDF7}	7	4	5	3	108(33)	54.01	53.01%
{ARAI, FIR20, GM1M}	6	5	4	3	100(44)	51.21	35.58%
{CASCADE, PR2}	11	5	2	4	122(59)	64.94	68.68%
{IIR8, PR1, WAVELET}	8	3	3	4	173(83)	68.45	31.23%

maximized, starting from one-unit fault tolerance $(k = 1)$ and going up to the maximum achievable k-unit fault tolerance $(k = MaxK)$. Such an incremental strategy is justified because k-unit failures are more probable than $k+1$-unit failures. $MaxK$ is obtained from the hardware units used in the ASPP and the hardware units required for each of the applications. For each value of k, we select an allocation for each application and evaluate the allocation by counting the number of distinct k-faulty unit combinations covered by it. If the number of distinct k-faulty unit combinations covered by the candidate allocation is better than that of the best allocation(s), it becomes the best allocation. However, if the candidate allocation covers the same number of distinct k-faulty unit combinations as the best allocation(s), it is added to this list of best allocation(s). This list of best allocations is then used as the starting point for improving the $k+1$-unit fault tolerance.

6.2 Evaluation of Resource Constrained Fault Tolerance

Table 9 summarizes the results of seven resource constrained fault-tolerant ASPPs. While column 2 shows the applications in the bundles, column 3 shows the throughput constraints of applications. Since fault tolerance is maximized (and not guaranteed), it is important to identify the faults that are tolerated. Consequently, the modules that are not used by an application are listed in column 7. Prefixes A, S, and M stand for adder, subtracter, and multiplier, respectively. The area of the synthesized ASPP and the area overhead vis-a-vis the area of the largest dedicated ASIC design are summarized in the last column. The area of the synthesized ASPPs includes the area of the composed controller. In these ASPP implementations, we assume that the composed ASPP controller is fault-free. The first term within parentheses in the last column shows the percent overhead of a non-fault-tolerant ASPP. The second term shows the percent overhead of rendering these ASPPs fault-tolerant. The additional area overhead of rendering a basic ASPP fault-tolerant is less than 7 percent. The average area overhead for these designs is consistent with those of guaranteed one-unit fault tolerance summarized in Section 5.4. This is because all of the designs (except for design #5) turned out to be one-unit fault-tolerant.

	H/w alloc	Faulty unit combinations tolerated								
		1-unit				..	k-unit			
		C_1^1	C_2^1	...	C_n^1	..	C_1^k	C_2^k	...	C_n^k
A_1	H_1^1	x	x		x			x		
	H_2^1		x	x			x	x		x
	..									
	H_{n1}^1		x	x			x			
A_2	H_1^2	x	x				x		x	
	H_2^2			x	x	x		x		
	..									
	H_{n2}^2	x		x	x	x				x
.....										
A_m	H_1^m			x			x	x		
	H_2^m		x		x	x		x	x	
	..									
	H_{nm}^m		x							

Fig. 10. Hardware allocation to maximize ASPP fault tolerance.

The first two designs in Table 9 are truly multifunctional in that all applications are distinct. The next three ASPPs are partially multifunctional in that some of the applications have two schedules. For example, design #5 has one schedule each of ADAPT and WAVELET and two WDF9 schedules. Since, the two WDF9 schedules have different hardware allocations, they can tolerate faults in different functional units. Finally, designs #6 and #7 implement multiple schedules of a single application (with different latencies). Consequently, these fault-tolerant designs yield *gracefully degradable* implementations of a single application. The performance of design #7 degrades by six clock cycles in the presence of any single module failure.

6.3 *K*-unit Fault Tolerance of ASPP Designs

The k-unit fault tolerance capabilities of the designs synthesized using the IFT allocation are summarized in Table 10. Except for #5, all designs can tolerate all single functional unit faults. In addition, these designs can tolerate some k-unit failures as well. For example, design #1 can tolerate 84.8 percent of all two-unit faults and 58.6 percent of all triple functional unit failures and so on.

7 Conclusion

We presented a computer aided design approach to synthesize ASPP designs that can tolerate single and multiple functional unit failures. We showed how fault tolerance can be incorporated at the behavioral level by combining the flexibility of multiple applications in an ASPP with judicious application-to-faulty-unit assignment. We described a fault-tolerant synthesis system to design ASPPs that are guaranteed to be k-unit fault-tolerant. Then,

TABLE 9
Resource Constrained Fault-Tolerant ASPP Synthesis

#	Application Bundles	t	#units +	-	*	tolerates faults in	area (mm^2)
1	DIF IIR7 LDILP WAVELET	11 10 6 15	3 2 2 2	3 2 2 1	3 2 1 1	 A0,S2,M0 A1,S0,M0,M2 A2,S1,S2, M1,M2	31.05 (17.64% + 4.83%)
2	DIT 8IIR CASCADE FIR20	11 23 11 17	5 1 3 4	2 0 0 0	3 2 2 1	 A1,A2,A3,A4, S0,S1,M2 A0,A4,S0, S1,M1 A0,S0,S1, M0,M2	 33.45 (24.35% + 1.54%)
3	IIR7 VOLTERRA VOLTERRA	10 17 17	2 1 1	2 1 1	2 1 1	 A0,S0,M0 A1,S1,M1	35.28 (4.56% + 4.60%)
4	PR1 IIR8 WAVELET WAVELET	8 11 14 14	3 2 2 2	4 0 1 1	4 3 2 2	 A0,M0,S0, S1,S2,S3 A2,M1,M3, S0,S1,S3 A1,M1,M2, S0,S1,S2	 64.55 (19.44% + 3.60%)
5	ADAPT WAVELET WDF9 WDF9	10 14 9 9	2 2 2 2	0 1 1 1	2 2 1 1	S0 M1 M0	20.27 (15.50% + 2.62%)
6	DIR	22 27 27 27 27	4 3 3 3 3	0 0 0 0 0	4 3 3 3 3	 A3,M3 A2,M2 A1,M1 A0,M0	 16.86 (4.90% + 4.58%)
7	MCM	19 25 25 25 25	3 2 2 2 2	4 3 3 3 3	3 2 2 2 2	 A0,S0,M0 A0,S1,M1 A1,S2,M0 A2,S3,M2	 15.76 (7.34% + 6.04%)

we described a hardware reallocation algorithm to maximize the fault tolerance of ASPP designs subject to resource constraints.

Inherent redundancy of multiapplication ASPPs has been used to yield low cost one-unit and two-unit fault tolerance. These fault-tolerant ASPPs in addition yield multiunit fault tolerance for free. However, extending this approach to single application fault tolerance will result in significant area overheads. The area overhead of multi and single-application ASPPs can be reduced via graceful performance degradation. The fault-tolerant ASPPs had very low area overheads for tolerating single unit failures and modest overheads for tolerating multiple unit failures.

TABLE 10
Multiple Unit Fault Tolerance

ex	k-unit fault tolerance $k=1$	$k=2$	$k=3$	$k=4$
1	100.0%	84.8%	58.6%	33.9%
2	100.0%	52.8%	17.9%	4.8%
3	100.0%	73.3%	43.3%	21.4%
4	100.0%	67.9%	36.4%	19.0%
5	60.0%	0.0%	0.0%	0.0%
6	100.0%	14.3%	0.0%	0.0%
7	100.0%	26.6%	0.0%	0.0%

ACKNOWLEDGMENTS

R. Karri's research was supported by US National Science Foundation CAREER award MIP-9702676.

REFERENCES

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

[2] A. Antola, V. Piuri, and M. Sami, "Semi-Concurrent Error Detection Data Paths," *Proc. IEEE Symp. DFT in VLSI Systems,* pp. 298-306, Oct. 1997.

[3] A. Antola, V. Piuri, and M. Sami, "High Level Synthesis of Data Pats with Concurrent Error Detection," *Proc. IEEE Symp. DFT in VLSI Systems,* pp. 292-299, 1998.

[4] D.M. Blough, F.J. Kurdahi, and S.Y. Ohm, "Optimal Recovery Point Insertion for High Level Synthesis of Recoverable Micro-architectures," *Proc. 25th Fault-Tolerant Computing Symp.,* 1995.

[5] M. Breternitz and J.P. Shen, "Architecture Synthesis of High Performance Application-Specific Processors," *Proc. 27th Design Automation Conf.,* pp. 542-447, 1990.

[6] H. Choi, J. Kim, C. Yoon, I. Park, S.H. Hwang, and C.M. Kyung, "Synthesis of Application Specific Instructions for Embedded DSP Software," *IEEE Trans. Computers,* vol 48. no. 6, pp. 603-614, June 1999.

[7] D. Clark and B. Hutchings, "Supporting FPGA Microprocessors through Retargetable Software Tools," *Proc. IEEE Symp. FPGAs for Custom Computing Machines,* Apr. 1996.

[8] A. DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century," *Proc. IEEE Symp. FPGAs for Custom Computing Machines,* Apr. 1994.

[9] G. DeMicheli, *Synthesis and Optimization of Digital Circuits.* New York: McGraw-Hill, 1994.

[10] Fujitsu Microelectronics Inc., "AFP MB86975 Data Sheet," Aug. 1987.

[11] A. Doumar, S. Kaneko, and H. Ito, "Defect and Fault Tolerance FPGAs by Shifting the Configuration Data," *Proc. IEEE Int'l Symp. Defect and Fault Tolerance in VLSI Systems,* pp. 377-385, 1999.

[12] S. Dutt, V. Shanmugavel, and S. Trimberger, "Efficient Incremental Rerouting for Fault Reconfiguration in Field Programmable Gate Arrays," *Proc. Int'l Conf. Computer-Aided Design,* pp. 173-176, 1999.

[13] D.D. Gajski, N.D. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic, 1992.

[14] G. Goossens et al., "Integration of Medium-Throughput Signal Processing Algorithms on Flexible Instruction-Stes Architectures," *J. VLSI Signal Processing,* vol. 9, nos. 1-2, pp. 49-65, 1995.

[15] L.M. Guerra et al., "High Level Synthesis Techniques for Efficient Built-In Self Repair," *Proc. IEEE Workshop DFT in VLSI Systems,* pp. 41-48, 1993.

[16] G. Hadley and B. Hutchings, "Design Methodologies for Partially Reconfigured Systems," *Proc. IEEE Symp. FPGAs for Custom Computing Machines,* Apr. 1995.

[17] B. Iyer, R. Karri, and I. Koren, "Phantom Redundancy: A High-Level Synthesis Approach for Manufacturability," *Proc. Int'l Conf. Computer-Aided Design 95,* pp. 658-661, 1995.

[18] R. Karri and A. Orailoglu, "Synthesis of Optimal Self-Recovering Microarchitectures," *Proc. 23rd Int'l Symp. Fault-Tolerant Computing,* June 1993.

[19] J. Lach, W.H. Mangione-Smith, and M. Potkonjak, "Low Overhead Fault-Tolerant FPGA Systems," *IEEE Trans. VLSI Systems,* vol. 6, no. 2, pp. 212-221, June 1998.

[20] E.A. Lee and D.G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. Computers,* vol. 36, no. 1, pp. 24-36, Jan. 1987.

[21] M. Lee, V. Tiwari, S. Malik, and M. Fujita, "Power Analysis and Minimization Techniques for Embedded DSP Software," *IEEE Trans. VLSI Systems,* vol. 5. no. 1, pp. 123-135, May 1997.

[22] R. Leupers, W. Schednk, and P. Marwedel, "Retargetable Assembly Code Generation by Bootstrapping," *Proc. Int'l Symp. High-Level Synthesis,* pp. 88-93, 1994.

[23] M.C. McFarland, A.C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems," *Proc IEEE,* vol. 78, no. 2, pp. 301-317, 1990.

[24] F.J. Meyer, X. Chen, J. Zhao, and F. Lombardi, "Fault Tolerance of One-Time Programmable FPGAs with Faulty Routing Resources," *Proc. Int'l Conf. Innovative Systems in Silicon,* pp. 155-164, 1997.

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

[26] A. Orailoglu and R. Karri, "Coactive Scheduling and Checkpoint Determination during the High Level Synthesis of Self Recovering Microarchitectures," *IEEE Trans. VLSI Systems,* vol. 2, no. 3, pp. 304-311, 1994.

[27] A. Orailoglu and R. Karri, "Automatic Synthesis of Self-Recovering VLSI Systems," *IEEE Trans. Computers,* Feb. 1996.

[28] D.A. Patterson and J.L. Hennessy, *Computer Architecture: A Quantitative Approach.* San Mateo, Calif.: Morgan Kaufmann, 1990.

[29] P.G. Paulin, C. Liem, T.C. May, and S. Sutarwala, "CodeSyn: A Retargetable Code Synthesis System," *Proc. Int'l Symp. High-Level Synthesis,* p. 94, 1994.

[30] G.D. Hillman, "DSP56200: An Algorithm-Specific Digital Signal Processor Peripheral," *Proc. IEEE,* vol. 75, no. 9, pp. 1,185-1,191, year?

[31] J.J. Raffel, A.H. Anderson, G.H. Chapman, K.H. Konkle, B. Mathur, A.M. Soares, and P.W. Wyatt, "A Wafer-Scale Digital Integrator Using Restructurable VLSI," *IEEE Trans. Electronic Devices,* no. 32, pp. 479-486, 1985.

[32] C.C. Stearns, D.A. Luthi, P.A. Ruetz, and P.H. Ang, "A Reconfigurable 64-Tap Transversal Filter," *IEEE Custome Integrated Ccircuits Conf.,* pp. 8.8.1-8.8.4, 1988.

[33] R.A. Walker and D.E. Thomas, "Behavioral Transformation for Algorithmic Level IC Design," *IEEE Trans. Computer-Aided Design,* vol. 8, no. 10, pp. 1,115-1,128, 1989.

[34] A.K. Yeung and J.M. Rabaey, "A 2.4 GOPS Data-Driven Reconfigurable Multiprocessor IC for DSP," *Proc. 1995 IEEE Int'l Solid-State Circuits Conf. ISSCC,* pp. 108-109, 1995.

[35] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: W.H. Freeman, 1979.

[36] R. Narasimhan, D. Rosenkrantz, and S.S. Ravi, "Efficient Algorithms for Analyzing and Synthesizing Fault-Tolerant Data Paths," *Proc. IEEE Symp. DFT in VLSI Systems,* pp. 81-89, Nov. 1995.

[37] G. Lakshminarayana, A. Raghunathan, and N. K. Jha, "Behavioral Synthesis of Fault Secure Controller/Datapaths Using Aliasing Probability Analysis," *IEEE Int'l Symp. Fault-Tolerant Computing,* June 1996.

[38] K. Kim, R. Karri, and M. Potkonjak, "Maximizing the Fault-Tolerance of Application Specific Programmable Signal Processors," *Proc. IEEE Workshop VLSI Signal Processing,* pp. 283-292, Nov. 1996.

[39] K. Kim, R. Karri, and M. Potkonjak, "Heterogeneous Built-In Resiliency of Application Specific Programmable Processors," *Proc. Int'l Conf. Computer-Aided Design,* pp. 406-411, Nov. 1996.

[40] K. Kim, R. Karri, and M. Potkonjak, "Synthesis of Application Specific Programmable Processors," *Proc. 34th Design Automation Conf.,* pp. 353-358, June 1997.

[41] S. Malik, M. Martonosi, and Y Li, "Static Timing Analysis of Embedded Software," *Proc. Design Automation Conf.,* pp. 147-152, June 1997.

[42] J. Rabaey et al., "Fast Prototyping of Data Path Intensive Architectures," *IEEE Design and Test,* vol. 8, no. 1, pp. 40-51, 1991.

[43] J.E. Vuillemin, P. Bertin, and D. Ronchin, "Programmable Active Memories: Reconfigurable Systems Come of Age," *IEEE Trans. VLSI Systems,* Mar. 1996.

Ramesh Karri received the PhD degree in computer science from the University of California at San Diego in 1993. From 1988-1989, he worked as a research engineer at CMC Ltd. in India. From 1993-1998, he was an assistant professor in the Department of Electrical and Computer Engineering at the University of Massachussetts at Amherst. From 1997-1998, he was a member of the technical staff at Bell Labs, Lucent Technologies. Since August 1998, he has been an associate professor of electrical engineering at Polytechnic University, Brooklyn, New York. His research interests include CAD for fault tolerance, reliability, and manufacturability, reconfigurable computing, and hardware-software co-design. He was the recipient of a US National Science Foundation CAREER award.

Kyosun Kim received the BS and MS degrees in electronic engineering from Yonsei University, Seoul, Korea, in 1986 and 1988, respectively, and the PhD degree in electrical and computer engineering from the University of Massachusetts, Amherst, in 1998. From 1988-1994 and since 1998, he has been with the Semiconductor R&D Center, Samsung Electronics, Yong-In, Korea. His research interests are in high-level synthesis, fault-tolerant systems, logic synthesis links to layout, RTL analysis, automatic layout, and CAD frameworks.

Miodrag Potkonjak received his PhD degree in electrical engineering and computer science from the University of California, Berkeley, in 1991. He is a professor in the Computer Science Department, School of Engineering and Applied Science at the University of California, Los Angeles, where he has been since 1995. In 1991, he joined C&C Research Laboratories, NEC USA, Princeton, New Jersey.

He has published more than 150 papers in leading CAD, real-time, and signal processing journals and conferences. He holds five patents. He received the Okawa Foundation Grant, the US National Science Foundation CAREER award and a number of best paper awards. He also received the TRW/School of Engineering and Applied Science at UCLA Excellence in Teaching Award in 1998. His research interests include system design, embedded systems, computational security, and intellectual property protection.