Fault Tolerant Finite State Control using Low Density Parity Checking

ghoover@engineering.ucsb.edu, forrest@ece.ucsb.edu
Department of Electrical and Computer Engineering
University of California, Santa Barbara

Abstract

Digital technology in space-bound, nanometer scale systems create the need for reliable computation in the face of a small but finite error rate. We present a designer-driven, semi-automated technique for synthesizing fault tolerance in finite state machines. This is done with lower implementation overhead in terms of both machine redundancy and overall logic complexity than conventional methods. We further show that it is possible to generate efficient encodings that inherently provide tolerance to constant error rates in excess of one error per cycle and, through the use of a high level description language (HDL), can be performed with turn-key simplicity. The technique is demonstrated in the design of a practical multi-threaded microprocessor engineered specifically for real-time and high bandwidth control. Comparison between the basic and fault tolerant versions shows that as large as 18% error rates can be recovered with minimal space overhead and almost no additional delay.

1 Introduction

Fault tolerance forms the foundation for critical systems, providing reliable computation and control in a broad range of applications. The importance of such systems is of long-standing recognition, having been a major area of interest for aerospace and military applications. Space-bound applications typify the necessity for dependable systems, operating in environments characterized by levels of radiation easily capable of disrupting normal machine behavior. In such systems, high levels of redundancy are achieved through replication of system components, greatly increasing implementation complexity and cost. These levels of redundancy are not uncommon in non-space-bound applications where the cost of human life must outweigh the cost of system development. For example, the flight control system for the Airbus A330/340 contains 3 primary flight control computers and 2 backups, supporting quintuple redundancy. Even with extreme levels of redundancy it is not uncommon in applications, such as deep space reconnaissance, for systems to shutdown as they pass through regions characterized by large radioactive activity. The Van Allen Belt is one such well known region of space, where unpredictable system behavior has the potential to jeopardize mission success by turning on conflicting devices.

The need for fault tolerant systems in terrestrial applications is of growing importance. Unpredictability in system design, manufacture, and operation is of critical importance to the population that these systems affect. Whether controlling distribution of power amongst city grids, coordinating travel through traffic lights, or supporting the lives of patients in the ICU, control systems affect our lives on a daily basis. High expectations rest on these systems to provide their services without interruption and in a variety of environments. Continuing progress in electronic fabrication technologies will inevitably cause unpredictability in system behavior, as local radiation phenomena begin to plague systems in all applications and with greater impact. With shrinking feature sizes in integrated circuit technology comes the increased potential for multiple faults originating from a single source.

While dependability of electronic systems has been a concern for many years, progress in finite-state controller fault tolerance has been limited. This is despite the fact that faults in machine control often have a more significant impact on system execution than a similar data fault. This observation is likely based on the ad-hoc means by which most finite-state controllers are designed. In particular, the vast majority of FSM synthesis work describes mechanisms to minimize the information redundance in a design, often at the cost of increasing both design gate complexity and delay. On the other hand, data-path reliability techniques have benefited from techniques for developed for reliable communication in the face of noisy physical channels. Research in these areas has generated error detection and correction (EDC) strategies capable of providing excellent error resilience with acceptable overhead. Finite state control, however, presents a different problem since delay overhead in such control is nearly always on the implementaition crit-

ical path. Further, analysis of the reachable states of a practical impelmentation indicate that the set of valid states is in general rather sparse– creating an opprotunity for custom encodings and alternative solutions to single event fault tolerance.

Central to the discussion of fault tolerant techniques is the notion of the targeted fault model. The single event upset (SEU) model is commonly accepted as a reasonable representation of expected error rates. This model provides that an occurring fault will cause errors in as many as one bit per cycle. While this may underestimate the true error rates occurring in even current technology, we use it as a foundation, aiming to provide SEU tolerance at a minimum. While the model focuses on errors in terms of storage elements, or bits, it is also possible that faults occur in the underlying logic. We assume, however, that this logic is static and that its output is eventually correct. Thus we target likely error scenarios where a delay or noise pulse prevents the correct output from being stored.

Conventional control techniques aim to curb the severity of faults in system execution by adding limited redundancy capable of both identifying and correcting a small number of faults. While effective in environments characterized by low error-rates, these techniques generally suffer from high cost in terms of both logic area and performance. Systems capable of correct execution in high error rate environments have yet to surface; most current implementations are designed to shut down in such environments. Shrinking feature sizes in integrated circuits (IC) raises concern about the effects of terrestrial radiation on future electronics and the severity of such interactions given that the source of a fault may disrupt a greater number of circuits. At the same time circuit scaling reduces the number of electrons used for signaling in modern designs. This increases the sensitivity of such systems to secondary radiation induction fault mechanisms such as substrate bias faults and induced timing errors.

This paper presents several novel techniques for synthesizing a range of FSM controllers that trade off implementation overhead with levels of fault tolerance. Through use of a subset of low density parity check (LDPC) codes (Section 2.1), we provide a low cost strategy capable of providing tolerance levels exceeding conventional (TMR/Hamming) techniques. At the local level our technique provides constant single error rate recovery such that error detection and correction occur transparently to machine execution. This technique is further enhanced by partitioning the state space, reducing logic complexity and increasing the total sustainable error rate given uniform error distribution. Optimizations in the fault tolerant construction reduce the overhead of logic complexity and take advantage of sparsity in the control encoding. At the local level, our technique is well suited for dense encodings as well as sparse, with worst case overhead better than that of a 1/2-rate code. Higher error rate environments benefit from a second level of checking capable of distinguishing multi-bit errors from single-bit errors with high probability and utilizes a novel decoding technique which takes advantage of both the sparsity and distance of state codewords.

We show that adding fault tolerance at the local level does not significantly impact area or performance and can achieve levels of machine resilience equal to or greater than conventional techniques. We further show that recovery of constant single error rates can be trivially provided to any control encoding, and that by trading off overhead in state size, we can generate control structures with very low performance overhead. Using the PyPBS specification tool we generate control structures for a multi-threaded processor that are well suited for application of our fault tolerant techniques and show that constant error rates as large as 18% of the state space can be transparently corrected in the FSM control logic. The remainder of this paper is organized as follows: Section 2 presents motivation for and related work pertaining to fault tolerant controllers, and relevant background on LDPC codes and the PyPBS specification language. Section 3 outlines both our local and global fault tolerant techniques, their unique decoding methods, and optimizations. Results for both the local and combined strategies are presented in Section 4, including relevant details of the demonstration processor design. Section 5 gives concluding remarks.

2 Motivation

Current techniques such as Hamming codes and triple-modular redundancy (TMR) are ill suited to provide the necessary levels of fault tolerance at a reasonable cost while maintaining acceptable performance levels. Prevalent techniques in the corresponding data-oriented arenas of fault tolerance have generated many ideas about application of error detection and correction schemes to system control but have often been at odds with the complexity inherent in implementations of many EDC codes. Hamming codes for instance, have been employed in the Single Independent Decoder (SID) architecture [9]. Each syndrome bit of a Hamming code, however, requires 1/2 of the codeword resulting in implementations that require $N/2-1$ binary XOR gates. The logic tree for each of these parity bits has depth $log_2 N-1$ and there are $log_2 N$ such trees for single error correction (SEC).

.

The prevailing technique, triple-modular redundancy (TMR), aims to provide reliable execution under the single event upset (SEU) model through 3x replication of system components. This technique can be implemented at various levels, creating redundancy at the architectural level down

to gate level [6]. In this scheme, performance penalties include the redundant voting circuitry, which functions after the computation is complete. A more substantial area overhead stems from the 3x replication of components and the necessary voting logic. While single faults can be identified and removed, this technique does not lend well to multi-fault events unless all faults occur within the same copy. The level of fault tolerance hardly mitigates the area overhead, forcing many systems to implement only partial-TMR schemes on critical components.

Drawing from technology in EDC codes, high level synthesis, and fault tolerant design, our technique aims to apply existing techniques in a novel way to achieve levels of fault tolerance superior to those available using conventional methods. Through use of a subset of low-density parity-check (LDPC) codes, we provide low-cost redundancy capable of correcting constant error rates in a manner transparent to machine behavior. Through use of the high level synthesis language PyPBS, we are able to quickly and automatically generate fault tolerant constructions that benefit from sparse encodings and optimization capabilities inherent in the language. The remainder of this section provides background on LDPC codes, their prevailing encoding and decoding strategies, and the PyPBS language.

2.1 Low Density Parity Check Codes

Originally conceived in 1960, low density parity check (LDPC) codes fell out of attention for many years due to the computational effort involved in encoder and decoder implementations [4]. In roughly the last decade these codes have experienced a dramatic comeback offering both encoding and decoding algorithms with linear time complexity and efficiency near the Shannon limit [10]. A low-density parity-check code (or Gallager code) is a block code that has a parity-check matrix, H, every row and column of which is "sparse" [7]. Valid codewords satisfy the requirement that all check nodes are of even parity. Regular Gallager codes are low-density parity-check codes that satisfy the constraint that every row of H has the same weight k and every column of H has the same weight j (Figure 2). Figure 1 illustrates an LDPC matrix H and its corresponding check functions. It has been shown that efficient Gallager codes are easily found at random subject to constraints j and k. Irregular LDPC codes have been shown to provide better efficiency than regular codes with some added complexity [5].

2.1.1 LDPC Encoding

Encoding of LDPC codes is generally performed by assigning message bits to variables nodes and calculating the missing values for the remaining nodes. While a simple solution is achieved by solving the parity check equations, this method involves the whole parity-check matrix and has complexity quadratic in the block length [4]. By rearranging the parity-check matrix into an approximate lower-triangular form, it is possible to use six submatrices of H to perform encoding in near linear time [8]. Certain classes of LDPC codes give way to linear encoding complexity by maintaining column weights of 2 or less. These staircase codes, while trivially solvable, add acceptable levels of redundancy at rates of 1/2 or better.

2.1.2 LDPC Decoding

LDPC codes exhibit excellent efficiency provided an optimal decoder exists. In general, decoding LDPC codes is NP-complete and work thus far has yet to discover an optimal algorithm [7]. Several decoding strategies exist, however, that provide excellent error correction given sufficient processing. The most effective algorithms are message passing algorithms that implement a variation of belief propagation. Such algorithms have been shown to provide complete correction of 1/2 rate codes with 7.5% added noise [7].

While effective in correcting large error rates, belief propagation (or soft-decision) decoding is an iterative process requiring floating-point precision to achieve efficiency near the channel limit. In his original work, Gallager presented a simple decoding method for binary symmetric channels (BSC) at rates far below channel capacity [1]. In this algorithm, parity checks are computed and any message node that is contained in more than some threshold number of erroneous check nodes is flipped. This hard-decision decoding scheme can benefit from multiple iterations, potentially correcting multiple bit errors with reduced computation complexity.

2.2 PyPBS

PyPBS is a high level synthesis language targeted for specification of sequential designs. The language syntax draws from the compact and expressive nature of regular expressions, while providing a modular framework to facilitate design and integration goals. The language focus is expressiveness of the specification limited by simplicity of hardware implementation. The rationale for these choices stem from the practical issues of constrained hardware design where a designer commonly works with a rather restrictive set of sequential constraints– memory protocols, interfaces, and previously constructed designs. PyPBS allows the designer to develop designs within such constraints in an incremental and applicative way.

Machine semantics follow from a non-deterministic finite automaton (NFA) model whereby execution is assumed

v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	
0	0	1	0	0	1	1	1	0	0	0	0	$c_1 = v_3 \oplus v_6 \oplus v_7 \oplus v_8 = 0$
1	1	0	0	1	0	0	0	0	0	0	1	$c_2 = v_1 \oplus v_2 \oplus v_5 \oplus v_{12} = 0$
0	0	0	1	0	0	0	0	1	1	1	0	$c_3 = v_4 \oplus v_9 \oplus v_{10} \oplus v_1 = 0$
0	1	0	0	0	1	1	0	0	1	0	0	$c_4 = v_2 \oplus v_6 \oplus v_7 \oplus v_{10} = 0$
1	0	1	0	0	0	0	1	0	0	1	0	$c_5 = v_1 \oplus v_3 \oplus v_8 \oplus v_{11} = 0$
0	0	0	1	1	0	0	0	1	0	0	1	$c_6 = v_4 \oplus v_5 \oplus v_9 \oplus v_{12} = 0$
1	0	0	1	1	0	1	0	0	0	0	0	$c_7 = v_1 \oplus v_4 \oplus v_5 \oplus v_7 = 0$
0	0	0	0	0	1	0	1	0	0	1	1	$c_8 = v_6 \oplus v_8 \oplus v_{11} \oplus v_{12} = 0$
0	1	1	0	0	0	0	0	1	1	0	0	$c_9 = v_2 \oplus v_3 \oplus v_9 \oplus v_{10} = 0$

Figure 1. Low Density Parity Check Code: H matrix and check functions

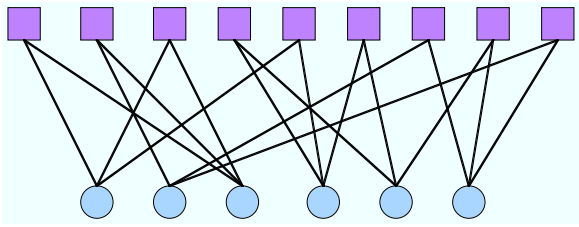

Figure 2. Low Density Parity Check Code: A graphical view

to be as general as possible, making PyPBS a natural fit for specification of sequential protocols and pipeline structures [2]. While PyPBS machine encodings are typically register-heavy, they prove to be well suited for high performance designs and not uncharacteristically large for designs generated from high level languages. These sparse encodings offer unique opportunities for optimization while allowing use of an applicative synthesis technique that limits the scope of design circuit changes for corresponding changes in the control hierarchy.

3 Technique

Adding fault tolerance to control structures using conventional techniques has proven to be a daunting task, generally requiring significant overhead and performance degradation. In this section we outline our local and global techniques, and discuss how the unique properties of state machines are used to reduce complexity and overhead. We emphasize the ability to generate a family of behaviorally equivalent machines that trade off levels of fault tolerance, performance, and area.

3.1 Local Fault Tolerant Construction

A state machine can be described as a set of state bits and their corresponding transition functions. These functions generate the valid states for the machine behavior, effectively decoding the current state and performing translation based on machine inputs. Through addition of a small layer of decoding logic to these functions, we can achieve reliable single error correction at low-cost in terms of both area and performance. Logic integration in this way provides error correction transparently to machine execution, but requires that the logic depth not be significantly impacted as to not affect performance.

LDPC codes provide the unique ability to encode and decode at low cost, requiring shallow gate depths and constant time complexity. To achieve these requirements, a subset of LDPC codes called staircase codes is employed. These codes effectively add redundant parity bits, creating a systematic code thereby reducing both encoding and decoding effort. Integration of encoding and machine functionality result in implementations with nearly zero encoding overhead.

Constructing the additional logic layer follows from generation of a parity-check matrix subject to constraints, addition of parity-check equations and necessary redundant state bits, and addition of a constant time, hard-decision decoder. For performance-driven designs, this layer is often implemented using logic only three gates deep with two and three input gates. Within this logic layer three sub-layers exist providing calculation of check functions, error detection, and error correction respectively.

3.1.1 Parity-Check Matrix Generation

The parity-check matrix is the combination of two matrices: D which describes the interconnection of check nodes and message nodes and P which specifies the connections of added parity nodes. Matrix D is created subject to heuristics specified in Figure 3, while P is trivially represented by a square matrix with ones along the main diagonal. Through manipulation of these heuristics it is possible to generate equivalent machines that trade off logic complexity with the number of added parity bits.

A parity-check matrix of height $|c|$ and width $|v|+|c|$ implements a code of rate $|v|/(|v|+|c|)$. The upper bound for this rate is determined by the lower bound of $|c|$ such that the number of unique combinations is greater than or equal to $|v|+|c|$ (Heuristic 2). By requiring that connections between check nodes and parity nodes be of weight one, all single errors occurring in these bits appear as a single unsat-

Heuristic	Description
$w = \|v\| = \|delta\|$	width is number of state bits
$w = \|v\| \leq fact(\|c\|)/((fact(deg_c - 1) * fact(\|c\| - deg_c - 1))$	the number of combinations of $\|c\|$ checks of degree deg_c
$deg_v \geq 2$	message node degree greater than or equal to 2
$c_i \neq c_j \forall c_i, c_j \in c$	all check node functions are unique
$v_i \neq v_j \forall v_i, v_j \in v$	all message nodes connect to unique check node set

Figure 3. Parity-check matrix construction heuristics

isfied check node. To disambiguate errors in message nodes from those in parity nodes it must hold that each message node be of degree greater than one (Heuristic 3). Heuristic 4 ensures that no two check nodes implement the same function, while Heuristic 5 provides unambiguous identification of errors in any bit.

With a check node degree of three, these heuristics provide single error identification using two-input gates, with minimal logic complexity and maximal performance. While accurate determination of the lower bound of $|c|$ requires factorial calculations based on the number of message nodes and the degree of check nodes, it can be estimated by $ceil(log_2|v|)$. Variation of check node degree directly effects the degree of message nodes, impacting logic complexity of check calculations as well as fault identification.

3.1.2 Satisfying Parity Function Constraints

While increasing the number of check functions serves to reduce logic complexity, each function incurs the cost of an additional parity bit for satisfying its function. When the transition functions for the state machine are known, it is possible to compute unique transition functions for each of these added parity bits rather than cascade logic on top of the existing circuits. In this way, we aim to alleviate the encoding time by computing parity values in parallel with next state functions. The complexity of these functions varies greatly depending on the functionality of the machine and the designated connections between message and check nodes. In general, reduced complexity in parity functions can be achieved through a reduction in the degree of the check node. Several optimizations exist for reducing both the logic complexity and parity bit overheads.

3.1.3 Fault Identification and Correction

Unambiguous identification of faults is possible only when all single faults sponsor unique patterns of satisfied check nodes. As described in prior sections, generation of the parity-check matrix ensures that this constraint hold and that the complexity of such identification is a function of the degree of each check node. Gallager's simplistic decoding method as presented in Section 2.1.2 serves as the basis for identifying and correcting faults in the construction. Rather than use a threshold mechanism to identify the fault, we use the fact that each fault creates a unique pattern that can be statically decoded. By maintaining small node degrees for all message nodes, we ensure that the complexity of this decoding is small. The designer can optionally specify that a dense encoding is desired, resulting in a complete binary decoding of the $|c|$ bits. It should be noted that some 2-bit error scenarios can also be recovered with local redundancy. Correction of 2-bit errors occurs when both bits generate non-overlapping output patterns for check functions. This is possible because only partial cubes are used for identifying bit errors, therefore any two disjoint patterns are capable of simultaneously correcting two errors.

Fault correction is trivially recognized by an XOR gate used to flip the erroneous bit as identified by the fault identification logic. Figure 4 illustrates the relationship between message and check nodes, fault identification decoding, and fault correction. Equivalence between basic and fault tolerant implementations can be guaranteed, as no modifications to the behavioral logic are performed. Optimizations to this structure exist for reducing logic complexity of fault identification and correction logic.

3.2 State Space Partitioning

The technique described above corrects single faults at constant rates without significantly impacting system performance. This is only the case if locality and message size are restricted such that wire length and fanin do not dominate the construction. For large state vectors, the cost of both interconnect and logic complexity become prohibitive. Partitioning the state vector provides small local logic blocks and increases fault coverage. Each local block benefits from low-cost single error recovery, thereby increasing the total coverage to k errors per cycle for k blocks. While rates of k errors per cycle can only be successfully handled given uniform error distribution among blocks, it is possible to increase the likelihood of such occurrence through use of interleaved partitioning. In an interleaved partitioning, localized multi-bit faults are distributed among local blocks thereby increasing the probability that they can

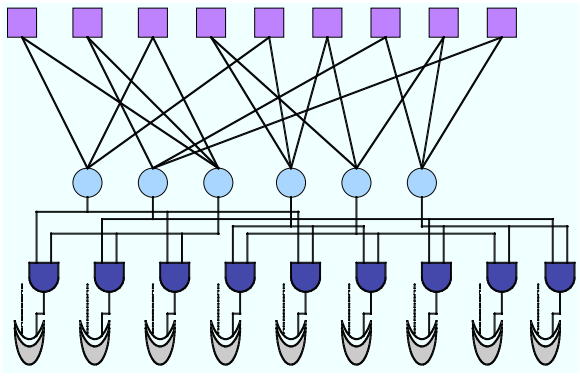

Figure 4. Error detection and correction circuits

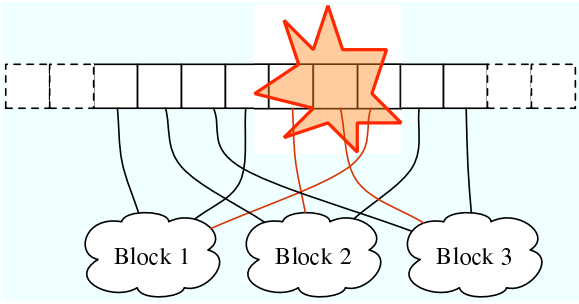

Figure 5. Interleaved partitioning

be successfully recovered (Figure 5).

The state partitioning has direct consequences on the area overhead of each local block, and hence the total design. Since the number of message bits that can be unambiguously identified grows exponentially with the number of check functions, it is apparent that larger blocks result in more efficient usage of added parity bits. A single partition, for instance, provides the best efficiency in terms of added parity bits, but suffers from poor performance due to wire length and logic complexity. Example partitionings of a state vector with 33 bits are shown in Figure 6, outlining the number of partitions, their respective sizes, the required overhead in terms of added parity bits, and the resulting number of per-cycle errors that can be tolerated. The partition sizes shown are chosen to maximize space efficiency given that message nodes are of degree two. Though there exists a broad range of acceptable partitionings, we target performance driven designs and as such attempt to partition the design into equal blocks of length 10 (6 message bits/4 parity bits).

Partitions	Sizes	Overhead	Faults
1	1 x 33	9	1
2	1 x 28, 1 x 6	12	2
3	2 x 15, 1 x 3	15	3
4	3 x 10, 1 x 3	18	4
5	4 x 6, 1 x 9	21	5
6	5 x 6, 1 x 3	23	6

Figure 6. Partitioning options for a 33 bit state vector

3.3 Optimization

Several optimizations exist for simplifying logic and reducing the overhead of added parity bits. By simply noting that all parity functions are regenerated on each cycle and are unused as inputs to machine behavior, we can safely remove both their associated fault identification and correction logic. The required decoding logic is thus reduced by nearly a factor of two for those stages.

Optimizations for sparse state encodings allow logic reduction at several levels. Through reachable state analysis it is possible to identify message bits whose parity combinations always satisfy the even-parity constraint. Parity bits for these functions are then unnecessary and can be removed from the construction. More common, however, is the ability to reduce logic complexity of check functions and fault identification logic. Sparsity in both of these encodings can help to streamline logic and reduce overhead.

3.4 Non-local Fault Tolerant Construction

Fault detection and correction for multiple faults in a single local block proves to be challenging, requiring greater overhead and increased performance impact. By taking advantage of redundancy in local blocks, however, we are able to provide greater levels of tolerance with limited area overhead. Accurate detection of multi-fault error scenarios is difficult given that detection must occur within the current clock cycle, inevitably affecting performance. Integration with our HDL synthesis process and the sparse nature of our output machine encodings allow delayed fault correction over multiple cycles using a novel decoding technique.

3.4.1 Non-local Redundancy

At the non-local level, redundancy is provided via a secondary staircase LDPC code over all local message and parity bits. The density of this encoding directly affects the overhead of the implementation, as well as the ability to distinguish multi-bit fault conditions. While best results are achieved when all local and non-local check functions are used for distinguishing error scenarios, the logic complexity of this approach is prohibitive for practical implementations. For instance, a design consisting of 33 bits of state can draw from as many as 43 check functions when constructed for block sizes of 6. For this reason it is suitable to use a subset of the available check functions thereby reducing logic complexity with decidable affects on distinguishability.

Initial exploration of this technique shows that it is possible to distinguish more than 99% of all 5-bit error possibilities from all single error scenarios when all check functions are considered. When only non-local check functions are considered, the accuracy of distinguishing multi-bit error scenarios drops buts is still well over 90% of 4-bit errors.

3.4.2 Sparse State Decoding

The resulting size of the ensemble of local blocks makes direct decoding over the non-local state vector impractical. However, by observing that the resultant state spaces are sparse and exhibit good distance, we can decode an erroneous state to its nearest codeword in a straightforward manner. We proceed by generating relations for each bit using the known reachable states. By expressing each bit as a function of the other bits in the state vector, we add yet another layer of decoding logic. This logic can then be generalized to provide error correction by gradually *relaxing* the input cube for each bit. By requiring that the *relaxed* input cube does not overlap that of any other valid codeword, we ensure that a deterministic decoding occurs.

Guaranteeing that the nearest codeword is always returned is non-trivial, as this is directly affected by the method in which the input cubes are *relaxed*. We propose an iterative approach which gradually expands the input cubes. Figure 7 is a well known representation for representing distance between codewords; it additionally depicts the process of gradually expanding the input cubes of codewords. Unlike Hamming codes, these state codewords do not exhibit uniform distance, making iterative expansion a suitable solution. In this way we aim to provide fairness in decoding by initially expanding cubes uniformly, while we achieve greater coverage of the binary space by allowing subsequent non-uniform expansion.

Clearly these input cubes have the potential to be both large and complex, resulting in sizable delays. While this technique cannot be applied directly to the input of the behavioral logic, as in local recovery, integration with our high level synthesis process allows machine operation to be suspended for multiple cycles while a valid codeword is recovered. This is accomplished by using multi-bit error detection circuitry to prevent machine outputs from firing. The machine is effectively held in a suspended state pending successful recovery of the nearest valid state.

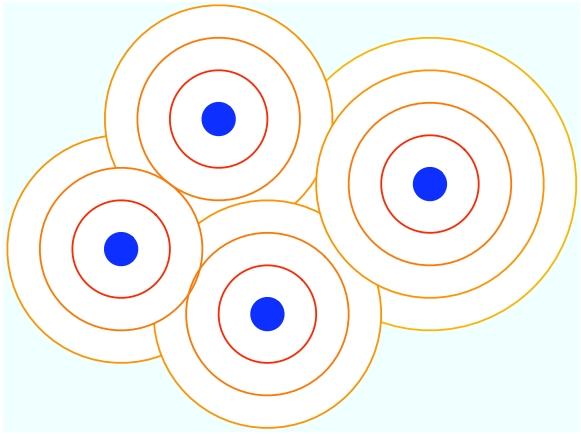

Figure 7. Iterative relaxation of input cubes

4 Implementation Results

ELLA is a dynamic-interleaved multi-threaded microcontroller targeting low-latency applications [3]. The control specification comprises 44 PyPBS productions, with the entire design consisting of 6,500 lines of synthesizable Verilog HDL. The control logic facilitates operation of the processor pipeline, memory and peripheral interfaces, as well mediation of conflicts arising from concurrent execution of multiple threads using 33 state bits to represent the behavioral NFA model.

The use of PyPBS as a specification language allows for simple application of the fault tolerant techniques presented. State vector partitioning targets high performance, maintaining message node degrees of two and local block sizes no larger than six message bits. The resulting partitioning consists of six partitions, adding 23 bits of overhead to the implementation and increasing total logic depth by only four levels of 2-input gates. Local fault tolerance provides detection and correction of error rates as high as six errors per cycle given uniform distribution of errors among blocks. Interleaved partitioning was also applied to the design, providing better coverage for multi-bit events resulting from densely packed circuits.

By comparison, a Hamming code over the entire state vector results in less overhead in terms of parity bits, requir-

ing only six additional bits. However, this approach results in significant performance overhead, with parity logic functions requiring as many as 17 inputs. Each of these logic trees requires five levels of logic when implemented with 2-input XOR gates, as compared to the two levels required by our implementation. The complexity of each Hamming parity function likely obviates the ability to merge behavioral logic with parity encoding as is done in our LDPC technique. In addition to encoding overhead, the cost of fault detection requires an additional set of XOR trees, while fault identification requires binary decoding of the syndrome. These levels of logic increase logic depth by eight for an implementation using 2-input gates, bringing the total performance delay to 14, more than three times that of our technique. The total area overhead of a Hamming implementation comprises 390, 2-input gates- 93 gates for parity functions and 99 gates for parity checking, with the remaining 198 gates for decoding the syndrome and correcting the error. Though our implementation adds over 500 gates to the final implementation, it is important to note that only 89 of these are added to the logic front end, with the majority of logic delay being masked by the behavioral logic. In a Hamming implementation, however, all logic overhead adds to the total logic depth, thereby increasing the length of any critical path.

A more direct comparison is possible with a multi-block Hamming implementation. A comparable implementation requires 23 parity bits to provide single local fault correction. The resulting logic depth for such an implementation adds six levels for XOR parity generation and checking trees, with decoding and correction adding another three levels. The resulting implementation is comparable in terms state size, but comes in five gate levels slower than our implementation. Partitioning for optimal Hamming block sizes, such as (7,4) and (15,11), result in overheads of 26 and 12 bits respectively, with non-optimal block lengths resulting in inefficiency in syndrome decoding. Both cases for optimal Hamming lengths suffer from performance degradation greater than that of our approach. The main difference between these techniques stems from the ability to vary the complexity of decoding logic. Hamming implementations lie at one end of the spectrum, while our performance targeted implementation lies at the other.

A comparable TMR implementation requires 3x area overhead, with performance degradation coming from the additional voting logic. While the added delay of voting logic is less than that of the corresponding fault identification and detection logic in our implementation, TMR requires nearly three times the overhead in terms of state bits and the corresponding logic dwarfs that of our implementation. The addition of triplicate voting negates any performance advantage of TMR. Further, we believe the fault coverage of our technique lends better to correction of multiple faults. The likelihood of faults occurring in different blocks in our implementation is greater than that of faults occurring in a single copy of a TMR implementation.

5 Conclusions

We have presented a novel technique for semi-automatic generation of fault tolerant controllers capable of sustaining constant error rates with minimal area and performance overhead. We have additionally shown that through partitioning and optimization it is possible to significantly increase the effectiveness of this approach in terms of recoverable error rates, while simultaneously reducing logic complexity. At a higher level, our non-local technique can provide tolerance levels well in excess of conventional techniques and integrates seamlessly with our high level synthesis process, providing a range of designer-driven implementations that trade off fault tolerance levels with implementation overhead.

While our initial work has shown promising results, subsequent work is necessary to explore additional applications. Identification of multi-bit error conditions remains a challenging aspect of the approach; additional work is necessary to identify an optimal strategy for selecting check functions that maximize identification of multi-bit errors. We believe that their may be potential in the use of non-systematic LDPC codes rather than those employed herein. The ability to integrate the encoding logic of such codes with the behavioral logic may help reduce the obvious delays inherent in such a scheme, decoding however, would remain a challenge.

References

[1] R. G. Gallager. Low-density parity-check codes. In *IRE Transactions on Information Theory*, pages 21–28, 1962.

[2] G. Hoover and F. Brewer. Pypbs design and methodologies. In *MEMOCODE '05*, 2005.

[3] G. Hoover, F. Brewer, and T. Sherwood. Ella: A multi-threaded low latency processor for embedded systems. 2005.

[4] B. M. Leiner. Ldpc codes - a brief tutorial. 2005.

[5] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman. Analysis of low density codes and improved designs using irregular graphs. In *need to get*.

[6] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Resarch and Development 6(2)*, pages 200–209, 1962.

[7] D. J. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[8] T. J. Richardson and R. L. Urbanke. Efficient encoding of low-density parity-check codes. pages 638–656, 2001.

[9] R. Rochet, R. Leveugle, and G. Saucier. Efficient synthesis of fault-tolerant controllers. In *The European Design and Test Conference (EDTC '95)*, 1995.

[10] A. Shokrollahi. Ldpc codes: An introduction. 2003.