A Unified Framework for Error Correction in On-Chip Memories

Frederic Sala
University of California,
Los Angeles (UCLA)
email: fredsala@ucla.edu

Henry Duwe
University of Illinois
at Urbana-Champaign
email: duweiii2@illinois.edu

Lara Dolecek
University of California
Los Angeles (UCLA)
email: dolecek@ee.ucla.edu

Rakesh Kumar
University of Illinois
at Urbana-Champaign
email: rakeshk@illinois.edu

***Abstract*—Many techniques have been proposed to improve the reliability of on-chip memories (e.g., caches). These techniques can be broadly characterized as being based on either error-correcting codes, side-information from built-in self test (BIST) routines, or hybrid combinations of the two. Although each proposal has been shown to be favorable under a certain set of assumptions and parameters, it is difficult to determine the suitability of such techniques in the overall design space.**

In this paper, we seek to resolve this problem by introducing a unified general framework representing such schemes. The framework, composed of storage, decoders, costs, and error rates, allows a full exploration of the design space of reliability techniques. We show how existing schemes can be represented in this framework and we use the framework to examine performance in the practical case of high overall and moderate BIST-undetectable fault rates. We show that erasure-based side-information schemes are less sensitive to BIST-undetectable errors compared to other techniques.

I. Introduction

Techniques correcting cell errors in on-chip memories have become increasingly important as process variation and voltage scaling have decreased reliability. Some techniques rely on exploiting side information about detectable cell faults gathered by built-in self test (BIST) routines. Others rely on error-correcting codes, which are designed to correct errors by ensuring sufficiently large redundancy in the stored data. There are also hybrid approaches that combine both concepts.

Unfortunately, it is very difficult to compare these reliability schemes against each other. The reliability schemes are typically validated against a fixed set of assumptions and may perform quite differently when the assumed parameters change. As an illustrative example, in Figure 1, we show the silent data corruption (SDC) rate for the VS-ECC technique [1] versus a comparable (further detail will be provided later on) Bit Fix scheme [2] for cell fault rates p_1 between 10^{-6} and 10^{-1}. Both VS-ECC and Bit Fix rely on identifying faulty cells using BIST and storing the relevant side information for decoding. When BIST routines are perfect and cannot fail to identify faults, as shown in the curves with thick lines, VS-ECC offers a slightly better SDC rate. On the other hand, when some fraction $p_2 > 0$ of faults are missed by BIST routines, as shown by the dashed lines for $p_2 = 10^{-1}$, the performances are dramatically changed, with VS-ECC nearly unusable. The value $p_2 = 10^{-1}$ represents the high end of practical range, as shown in [3].

This work was partially supported by NSF and CFAR, within STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, along with NSF grants 1029030 and NSF 1150212.

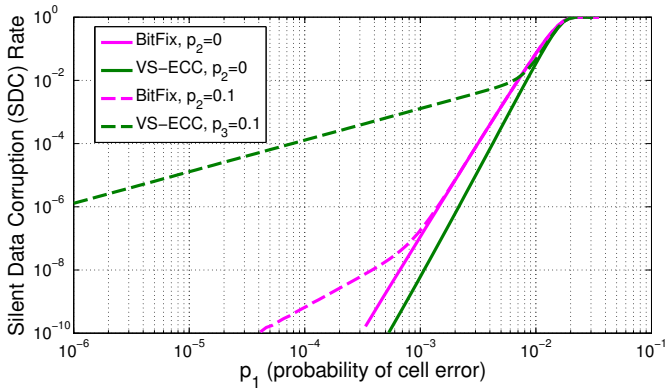

Fig. 1. Silent data corruption (SDC) rates for Bit Fix and VS-ECC. The p_2 parameter represents the probability a fault is not detected by a BIST-like routine; solid curves are for $p_2 = 0$ while dashed curves indicate $p_2 = 0.1$. Observe that p_2 has a significant effect on VS-ECC SDC.

Situations like the one shown in Figure 1 motivate us to better define and explore the rich design space of reliability techniques for memories. This is a challenging problem, due to the fact that there are many known techniques with very different approaches. Therefore, in this preliminary work, we introduce a general framework that can express previously introduced techniques (and can suggest novel ones.) The framework provides abstract data storage and decoding components and defines an overall "cost" function, allowing for a fair comparison between different schemes. Furthermore, the framework includes three classes of fault/error rates, capturing the complex fault behavior of systems. We use the framework to explore performance of schemes in the practical case with high cell fault rates and varying BIST fault miss rates.

II. Prior Work

There have been many proposals to tolerate high fault rates in on-chip memories (e.g., L1 and L2 caches). Broadly speaking these proposals fall into three categories— those that rely solely on prior knowledge of fault locations (*fault aware*), those that use no prior knowledge of fault locations (*fault agnostic*), and *hybrids* combining the two.

A. Fault Aware Techniques

Several prior works observed that if a fault is persistent and can be detected during an offline built-in self test (BIST) routine or through online testing, then the corresponding logical bits can be remapped to fault-free bits (at a cost in capacity or area) [2], [4], [5], [6], [7]. Bit-fixing [2] and word disabling are prototypical examples. Bit-fixing remaps adjacent bits in a cache line to redundant bits in an unused cache line within the same set. The redundant bits store the location of the faulty patch and are used to shift the patch bits into the word for the faulty bits within a small number of cycles (i.e., two or three cycles). Word disabling operates at a coarser granularity; chunks of two cache lines are used to store the data for one cache line. If a chunk contains one or more faults, the chunk is disabled. The line will be functional if half or more of each of the chunks from the two original lines are fault free. Each chunk requires a bit to specify if it is disabled or not. The advantage of fault aware techniques is that they require a small amount of decoding complexity (i.e., hardware) to implement, making them low cost in both decoder area and latency. Unfortunately, fault aware techniques lack the ability to correct any faults which are not detected via a BIST-like routine.

B. Fault Agnostic Techniques

In order to correct both BIST-detectable and BIST-undetectable faults, other works have been proposed using error correction alone [3], [8]. MS-ECC [3] uses segmented Orthogonal Latin Square (OLS) codes to have low latency error correction, but requires doubling the number of bits stored. Hi-ECC [8] uses large, 1KB Bose-Chaudhuri-Hocquenghem (BCH) codewords to have a low number of redundant bits, but requires a large, long-latency decoder. Since fault agnostic error correction ignores the fault information from BIST, imperfect as it may be, the resulting schemes are inefficient either in terms of storage overheads or decoding complexity.

C. Hybrids

Due to the undesirability of using either completely fault agnostic or fault aware techniques, some past work has explored hybrid solutions [8], [1], [9]. [8] applied a bit-fixing approach prior to using its complex, multi-bit BCH decoder. Unfortunately, this adds the latency of both techniques making the resulting latency prohibitive for latency-critical on-chip memories like an L1 cache. VS-ECC [1] uses information about the number of BIST-detectable faults within a line to allocate a stronger, multi-bit code only for lines with multiple hard faults. This reduces both the average latency of decoding and the number of redundant bits while still being amenable to correcting some BIST-undetectable faults. However, VS-ECC still requires the hardware for a complex decoder and cannot easily maintain yield and SDC targets over different rates of BIST-undetectable faults. [9] proposes applying fault location information to a SECDED code in the form of erasures. By replacing faulty cells with erasures, a SECDED decoder can correct two faults per codeword and detect a third fault that may be BIST-undetectable. The scheme then uses a low-overhead, low-coverage checksum to correct BIST-undetectable faults. While EB-ECC makes efficient use of fault information, the storage of erasures for higher fault rates would become prohibitive and the BIST-undetectable correction solution cannot handle BIST-undetectable fault rates as high as those evaluated in [3]. We aim to use our framework to find techniques capable of efficiently handling high fault rates and moderate BIST-undetectable fault rates.

D. Comparisons and Evaluations Techniques

Past work's treatment of BIST-undetectable faults has generally been ad hoc, providing guarantees only for a single fault within a codeword ([1], [9]), rather than SDC rates for entire caches. Furthermore, past work often targets very specific technology (e.g., Hi-ECC targets eDRAM and MS-ECC targets estimated CMOS SRAM). Our work provides a framework to fully explore the design space of reliability techniques of current on-chip memory technologies in the context of imperfect BIST. The proposed framework will be useful to fault tolerant design with future on-chip memory technologies where absolute fault rates, BIST-detectable fault rates, and correction latencies significantly differ.

III. Unified Framework

In this section, we introduce a general, unified framework for representing error-correction schemes capable of incorporating side information for on-chip memories. Our goal is to provide an abstraction of a system that is sufficiently general to encompass any of the existing, proposed approaches. Moreover, the framework allows for a combination (or hybrid) consisting of multiple schemes. We begin by motivating the need for such a framework. We dedicate the following four subsections to a description of the framework, consisting of storage, decoding functions, costs, and error rates. Finally, we give examples of how existing schemes, such as Bit Fix or VS-ECC can be expressed in the framework.

A. Motivation

Previous works focus on proposing a particular system and showing (analytically or through simulation) superior performance to existing systems for some particular set of parameters. As we saw in Figure 1, if the parameters are changed, a very different result may arise. Similarly, for the comparisons to be fair, a number of different factors must be kept constant between different systems (similar areas, etc...) Changing some aspect will also change the comparison results. These notions motivate us to describe an abstract, general framework that can express as many systems and as much of the design space as possible.

In other words, we are interested in describing a framework consisting of "dials" that we can turn in order to reach different parts of the design space. These dials include errors and fault rates, but also abstract costs that correspond to area and latency. The framework we propose has the advantage of defining a rich design space, allowing for partitioning, search,

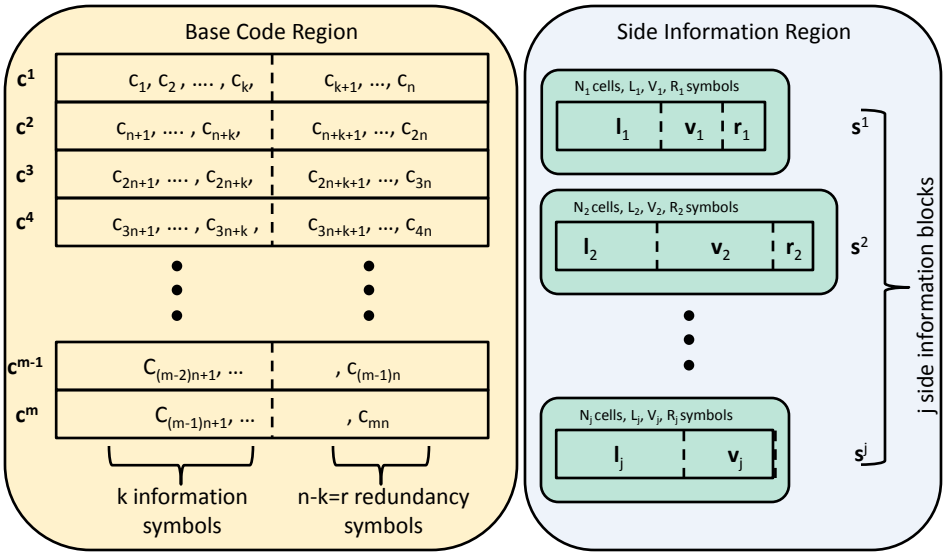

Fig. 2. Block diagram depicting storage components of a single sharing unit. On the left is the base code region, containing m codewords $\mathbf{c}^1, \mathbf{c}^2, \ldots, \mathbf{c}^m$ from a $[n, k, t]_q$ code $\mathcal{C}_B$. Each codeword stores k information symbols and r redundancy symbols with $k + r = n$. On the right is the side information region, consisting of j side information blocks. The ith block contains information referring to N_i cells; the location of these cells is stored in ℓ_i and the relevant information in $\mathbf{v}_i$. The pair $(\ell_i, \mathbf{v}_i)$ is denoted by $\mathbf{s}^i$. The costs (in symbols) of the two fields are L_i and V_i, respectively.

optimization, and extensive comparisons. It also allows us to examine questions that cannot be easily answered without the framework. For example, we might wish to explore when the reliability of side information is sufficiently low that systems should no longer rely on it. The answers to such questions cannot be found without an expressive underlying framework.

B. Storage Components

The first part of our framework are the storage components. We divide the system's storage into *sharing units* with two parts (or regions) each, as shown in Figure 2. An entire cache consists of many sharing units. The storage components of a sharing unit can only be used to correct information symbols within that sharing unit. The two regions are

1) A **base code** region, which includes the information symbols that store data and redundancy symbols used for decoding the base codewords, and the
2) **Side information storage** region, which is a set of cells storing information that is used to aid or improve decoding the base codewords, but is not part of the base codewords.

In other words, the two areas include cells storing codewords from an error-correcting code and an additional area that contains one or more types of side information. Specifically, we define side information as any information pertaining to the stored words that is not found in the base code information or redundancy symbols.

There is a separation between the two regions. The base code region is designed to have inherent redundancy which can correct errors with no knowledge about these errors. Conversely, the side information region is designed to store specific information (acquired from BIST during device operation) about cells that can be used (possibly in conjunction with the base code) to correct errors. Note that we can represent the extreme cases where there is no code (by allowing the code not to have any redundancy) or no side information (by leaving the side information region empty.)

Next we define these terms more precisely. We refer to cells capable of storing a symbol from $[0, q-1] = \{0, 1, \ldots, q-1\}$ as $c_1, c_2, \ldots$. We take the base code region to include m codewords from a $[n, k, t]_q$ (length n, q^k codewords, t-error correcting) code $\mathcal{C}_B$, where each codeword has k information symbols and r redundancy symbols from $[0, q-1]$. Therefore, the base code region stores mk symbols (or, equivalently, $mk\lfloor \log_2 q \rfloor$ bits) in an array of nk cells (representing $nk\lfloor \log_2 q \rfloor$ bits). Here, the cells are $c_1, c_2, \ldots, c_{mn}$, where the ith codeword $\mathbf{c}^i$ is given by $\mathbf{c}^i = c_{n(i-1)+1}, c_{n(i-1)+2}, \ldots, c_{n(i-1)+n}$.

The side information region is organized as follows. We store j **side information (SI) blocks**. Each block contains some type of information that refers to a particular group of cells. In our framework, we require each separate block to refer to an aligned, contiguous group of cells. In order to take advantage of side information, the system requires both the locations of the cells being referred to and the pertinent information. Therefore, each block must contain several fields:

1) The **location field** for the ith block contains the location z of a group of N_i contiguous cells $c_z, c_{z+1}, \ldots, c_{z+N_i-1}$. This location information is stored as ℓ_i. The total number of symbols required is defined to be L_i.
2) The **value field** stores the relevant side information. There are V_i symbols used for the ith field. We define

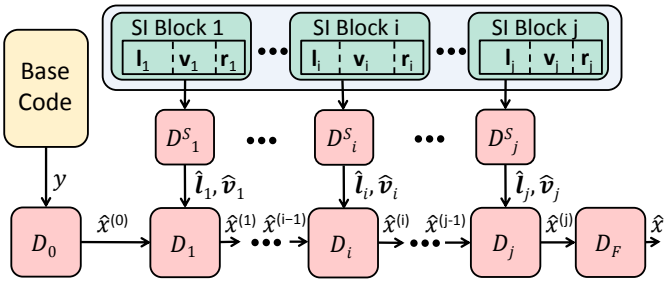

Fig. 3. Block diagram illustrating the framework's decoding process.

the information in the value field to be $\mathbf{v}_i$.

3) The **redundancy field** $\mathbf{r}_i$ stores R_i redundancy symbols used when side information is protected with an additional error correcting code.

The total quantity of side information in symbols is then $\sum_{i=1}^{j}(L_i + V_i + R_i)$. Side information blocks need not be the same size. This enables the framework to represent a system that uses a combination (a hybrid) of multiple schemes.

C. Decoding Functions

The previous subsection described the storage components of a general system. In addition, we have circuitry that is needed to decode the stored data. Naturally, each of the components in our system (the base code and the side information blocks) will have an associated decoding function. Although slightly complex to state formally, these decoding functions are needed to maintain full generality in expressing different side information schemes. Later on we give illustrative examples to clarify the idea.

Let us say that we have written the value $\mathbf{x} = x_1, x_2, \ldots, x_{mn}$ to the cells $c_1, c_2, \ldots, c_{mn}$ in the base code. After some time, these cells are read as $\mathbf{y} = y_1, y_2, \ldots, y_{mn}$, where possibly $y_i \neq x_i$ due to an error.

We have an initial decoding function $\mathcal{D}_0$ that does not use any of the side information blocks. This function may (but is not required to) use the decoder $\mathcal{D}_B$ of the code $\mathcal{C}_B$. Applying $\mathcal{D}_0$ produces an estimate $\hat{\mathbf{x}}^{(0)} = \mathcal{D}_0(\mathbf{y})$. In addition, with the ith SI block we associate the two decoding functions $\mathcal{D}_i^S$ and $\mathcal{D}_i$. The function $\mathcal{D}_i^S$ decodes the side information (when it is protected by an error-correcting code using R_i redundancy bits), providing an estimate $(\hat{\ell}_i, \hat{\mathbf{v}}_i)$. Afterwards, $\mathcal{D}_i$ operates on the (estimated) side information block data $(\hat{\ell}_i, \hat{\mathbf{v}}_i)$, the read data $\mathbf{y}$, and the output $\hat{\mathbf{x}}^{(i-1)}$ of the previous decoding function $\mathcal{D}_{i-1}$ to produce an updated estimate $\hat{\mathbf{x}}^{(i)} = \mathcal{D}_i(\mathbf{y}, \hat{\mathbf{x}}^{(i-1)}, \hat{\ell}_i, \hat{\mathbf{v}}_i)$.

Finally, we allow for a "final" decoding map $\mathcal{D}_F$ that uses the result of the last side information decoding estimate to produce a final output read $\hat{\mathbf{x}} = \mathcal{D}_F(\mathbf{y}, \hat{\mathbf{x}}^{(j)})$. The purpose of the final decoder is to allow the system the flexibility to use the base code decoder either before (as $\mathcal{D}_0$) or after (or both before and after) side information decoding. We illustrate the abstract decoding process in Figure 3.

For all of the decoding maps $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_j, \mathcal{D}_F$, we introduce corresponding decoding area complexities (measuring the worst case number of operations required for decoding) $\gamma_0(m,n), \gamma_1(m,n,L_1,V_1), \ldots, \gamma_j(m,n,L_j,V_j), \gamma_F(m,n)$ and decoding latency complexities (measuring the worst case number of operations needed to be performed in series) $\delta_0(m,n), \delta_1(m,n,L_1,V_1), \ldots, \delta_j(m,n,L_j,V_j), \delta_F(m,n)$.

Parameter	Value	Parameter	Value
ρ	$16\text{KB}/(mk)$	α	1.0
β_1	1.0	β_2	1.0

Fig. 4. Cost function parameters to model an L1 cache built in a 65nm CMOS technology.

D. Costs

We also introduce an overall **cost** associated with our system. The cost measures resource consumption in a real system by mapping different quantities (area, latency, complexity, etc...) into a single abstract number for easy comparison. The first part of the cost involves the storage for the base code and side information cells. We set each base code cell cost to 1 and each side information cell to a parameter α. This models robust side information cells with a lower fault rate (in which case $\alpha \gg 1$.) The total cost of storage for a sharing unit is given by $mk + \alpha(\sum_{i=1}^{j}(L_i + V_i + R_i))$.

We also consider the decoding complexity, which we further break down into two components: added decoder area and added decoder latency. We weight the components by two parameters, β_1 and β_2. β_1 defines the relative importance of decoder latency while β_2 defines the relative importance of decoder area. Depending on what type of on-chip memory is being designed, these values will be different (e.g., an L1's β_1 will have a higher value than an L2's β_1 because latency is more critical to an L1's functionality[1]).

The area per sharing unit and latency terms are multiplied by the number of sharing units, ρ, because they are applied equally to each sharing unit. Since the decoding area complexity is amortized over the entire cache, it is not multiplied by the number of sharing units. With this, we may write the total cost of our system as

$$\rho\left(mk + \alpha\left(\sum_{i=1}^{j} L_i + V_i + R_i\right) + \beta_1\left(\delta_0(m,n) + \sum_{i=1}^{j}\delta_i(m,n,L_i,V_i) + \delta_F(m,n)\right)\right) + \beta_2\left(\gamma_0(m,n) + \sum_{i=1}^{j}\gamma_i(m,n,L_i,V_i) + \gamma_F(m,n)\right).$$

E. Error Rates

Next, in order to give a performance comparison or analysis, we must describe the system error rates. Towards this end we introduce three error rate parameters p_1, p_2, p_3:

[1]If a processor supports speculation across cache error correction, the latency parameter, β_1, may be set very low.

1) p_1 is the input bit error rate probability. This is the probability that one of the base code cells c_i $(1 \le i \le mn)$ experiences a fault, so that $y_i \neq x_i$.
2) p_2 is the probability that a fault or error is not caught by a self-test (BIST) routine (and is thus not available to be described as side information).
3) p_3 is the probability that a symbol is in error in the side information. If "robust" cells are used as the storage for the side information blocks, we will have that $p_3 < p_1$.

F. Illustrative Examples

We give examples of existing systems for error correction and describe how they can be expressed in the framework.

1) MS-ECC (using OLS codes). No side information is used so that $j = 0$ and we use the single decoding function $\mathcal{D}_0$ which corresponds to the OLS code $\mathcal{C}$. This case represents one of the extremes of our system setup.

2) Bit Fix. In Bit Fix, the locations and correct values of pairs of adjacent faulty binary cells are stored. There are two cells referred to by each SI block, so that $N_i = 2$ for all i. The location of a pair of cells requires $L_i = \log_2 \frac{mn}{2}$ bits and the values $\mathbf{v}^i = (v_1, v_2)$ require $V_i = 2$ bits. ℓ_i and $\mathbf{v}_i$ are protected by a single error-correcting code. The total cost is then $j(\log_2(mn/2) + 2 + R)$ bits for j pairs, where R is the needed amount of redundancy bits to protect $\ell_i, \mathbf{v}_i$. Here, the decoding function $\mathcal{D}_i$ is almost immediate: If the two cells specified by the L_i location bits are x_z and x_{z+1} and the value field contains v_1 and v_2, then,

$$\begin{aligned}\hat{\mathbf{x}}^{(i)} &= \mathcal{D}_i(\mathbf{y}, \hat{\mathbf{x}}^{(i-1)}, \ell_i, \mathbf{v}_i) = \mathcal{D}_i(\mathbf{y}, \hat{\mathbf{x}}^{(i-1)}, z, v_1, v_2) \\ &= \hat{x}_1^{(i-1)}, \ldots, \hat{x}_{z-1}^{(i-1)}, v_1, v_2, \hat{x}_{z+2}^{(i-1)}, \ldots, \hat{x}_{mn}^{(i-1)}.\end{aligned}$$

In other words, each side information decoding function simply replaces (fixes) the relevant bits.

3) VS-ECC. With VS-ECC, codewords that contain more detected faults are targeted for additional protection with a very strong code (in excess of the base SECDED code. First, the initial decoding function $\mathcal{D}_0$ corrects only those codewords that are not targeted for extra protection. These codewords are decoded with $\mathcal{D}_B$. Next, each side information block refers to a full codeword of n cells, so $N_i = n$. The location of a particular codeword requires $L_i = \log_2 \frac{mn}{n} = \log_2 m$ bits. Next, $\mathbf{v}_i$ consists of the additional redundancy bits needed for the codeword. Let $\mathcal{D}_S$ be the strong code decoder. With this, if the zth codeword is the one being targeted by the ith block, $\hat{\mathbf{x}}^{(i)} = \mathcal{D}_i(\mathbf{y}, \hat{\mathbf{x}}^{(i-1)}, \ell_i, \mathbf{v}_i) = \mathcal{D}_i(\mathbf{y}, \hat{\mathbf{x}}^{(i-1)}, z, \mathbf{v}_i) = \hat{x}_1^{(i-1)}, \ldots, \hat{x}_{(z-1)n}^{(i-1)}, \mathcal{D}_S(y_{(z-1)n+1}, \ldots, y_{zn}, \mathbf{v}_i), \hat{x}_{zn+1}^{(i-1)}, \ldots, \hat{x}_{mn}^{(i-1)}$.

Here too, $\mathcal{D}_F$ is an identity function.

4) Single-cell erasure scheme. In this scheme individual faulty cells are marked with erasures and decoded using an errors-and-erasures decoder. Thus, $N_i = 1$ for all $1 \le i \le j$, $L_i = \log_2(mn)$, and, interestingly, we do not need any information about cells other than their locations, so that $V_i = 0$ and $\mathbf{v}_i$ is an empty vector. The total bit cost is thus $j \log_2(mn)$. The decoding function $\mathcal{D}_i$ marks the locations given by the location fields as erasures: $\hat{\mathbf{x}}^{(i)} = \mathcal{D}_i(\mathbf{y}, \hat{\mathbf{x}}^{(i-1)}, \ell_i, \mathbf{v}_i) = \mathcal{D}_i(\mathbf{y}, \hat{\mathbf{x}}^{(i-1)}, z) = \hat{x}_1^{(i-1)}, \ldots, \hat{x}_{z-1}^{(i-1)}, \text{'?'}, \hat{x}_{z+1}^{(i-1)}, \ldots, \hat{x}_{mn}^{(i-1)}$.

It is also interesting that with this scheme, the main decoding is done entirely by the final decoder. That is, we take the base decoder to be $\mathcal{D}_B(\mathbf{y}) = \mathbf{y}$, so that no errors are corrected. Afterwards, the erasures are marked with the use of side information. Finally, the decoding (of both errors and erasures) is performed with $\mathcal{D}_F$.

Scheme	Type	n	k	m	j	V_i	L_i	R_i
MS-ECC	FA	128	64	1	0	0	0	0
Bit Fix	FW	64	64	1	5	2	5	4
VS-ECC	H	21	16	4	4	5	2	0
EB-ECC	H	72	64	1	4	0	7	0
OLS/Erasures	H	96	64	1	2	0	7	0

Fig. 5. Parameters n, k, m, j for five different error correction techniques, including fault agnostic, fault aware, and hybrid techniques. The OLS/Erasures scheme is novel. FA indicates a fault agnostic technique. FW indicates fault aware. H represents hybrid.

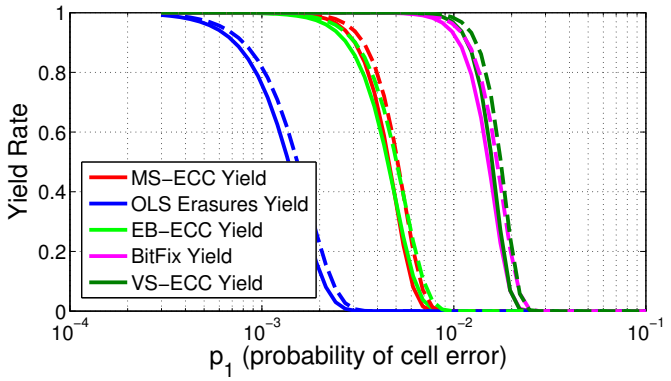

Fig. 6. Yields for fault rates p_1 for five techniques for $p_2 \in \{0, 0.1\}$.

IV. Results

In this section, we use the framework to derive practical results. We are especially interested in the realistic case where we have a relatively high fault rate p_1, imperfect BIST routines $(p_2 > 0)$, and moderate side-information fault rates p_3.

A. Methodology

We compare several implementations of existing schemes in terms of yield and reliability. The idea is to produce a fair yield and reliability comparison by matching the costs. As an initial evaluation, we set a constant useable space (i.e., 16KB) and set the other cost parameters to 1.0 as shown in Figure 4. We leave a more rigorous evaluation of cost parameters to future work. Under these cost assumptions, the code parameters shown in Figure 5 have equal cost. The fifth scheme, called OLS/Erasures is a novel technique that we propose in this work, motivated by the possibility that $p_2 > 0$. The idea is to use an OLS code capable of correcting several errors and to mark BIST-detected faults by erasures. These erasures (along with possibly undetected errors) are corrected by an errors-and-erasures OLS decoder.

We note that although our framework enables us to derive parameters for different schemes with comparable cost, we must still derive yield and reliability results analytically or through simulation[2]. Below, we detail how to derive the yield rate and silent data corruption (SDC) rates for the five schemes we have described at comparable cost. In many cases, it is possible to derive formulas based on the parameters p_1, p_2, p_3 from our framework. For example, in the case of yield rate for Bit Fix, we write p_U for the probability a patch (that is, one side information block) is usable, so that $p_U = (1 - p_3\bar{p_2})^a + ap_3\bar{p_2}(1 - p_3\bar{p_2})^{a-1}$. Here, $\bar{p_2} = 1 - p_2$ and $a = L_i + V_i + r_i$. This result comes from the fact that Bit Fix patches are protected with a single error-correcting code. Then, if p_b is the probability that there are b ($0 \leq b \leq j$) usable patches, $p_b = \binom{j}{b} p_U^b (1 - p_U)^{j-b}$. The overall yield rate is then

$$1 - \sum_{b=0}^{j} p_b \left(\sum_{i=b+1}^{n/2} \binom{\frac{n}{2}}{i} (1 - (1 - p_1\bar{p_2})^2)^i (1 - p_1\bar{p_2})^{2(\frac{n}{2} - i)} \right).$$

We perform similar calculations for yield and SDC rates for each of the five schemes in order to derive the plots described in the yield and reliability comparison section below.

B. Yield and Reliability Comparisons

First, in Figure 6 we show a plot of the yields for five schemes with two curves for each scheme. The solid line indicates the case of $p_2 = 0$, so that BIST is capable of detecting all faults. The dotted lines indicate $p_2 = 0.1$, so that many faults are missed by BIST. All schemes follow a similar pattern: the yield is 100% until a narrow increase in p_1 yields a dramatic drop in yield to 0. A surprising observation is that **a higher BIST miss rate p_2 improves the yield**. However, this means that a higher fraction of faults exist during operation, resulting in a higher SDC rate. Therefore, as we will see, in the high p_2 case, we pay for the yield with a higher SDC rate.

Next we plot silent data corruption (SDC) rates for the schemes in Figures 7 and 8. We sweep p_1 over the range 10^{-6} to 10^{-1} and include the two extreme cases $p_2 \in \{0, 0.1\}$ for BIST reliability. In addition, we let the side information storage fault rate p_3 maintain a constant ratio with p_1, so that higher p_1 also yields higher p_3. The two ratios are 10^{-5} (top plot) and 10^{-1} (bottom plot). Naturally, the MS-ECC scheme that does not use side information is identical in all cases. However, we observe that **erasure-based schemes are less sensitive to BIST misses (high p_2)** compared to other schemes. On the other hand, the p_2 BIST failure parameter is much less important when the p_1/p_3 ratio is large (reliable side information storage) compared to when p_1/p_3 is low (unreliable side information storage). These observations show the power of our proposed framework: we can see the effect of key parameters such as p_3 and p_2 (and their interplay) on various techniques.

[2]The performance of using our framework to derive parameters is dependent on the efficiency of analytic solutions or of simulations. It took us approximately 10 minutes to gather the results presented here.

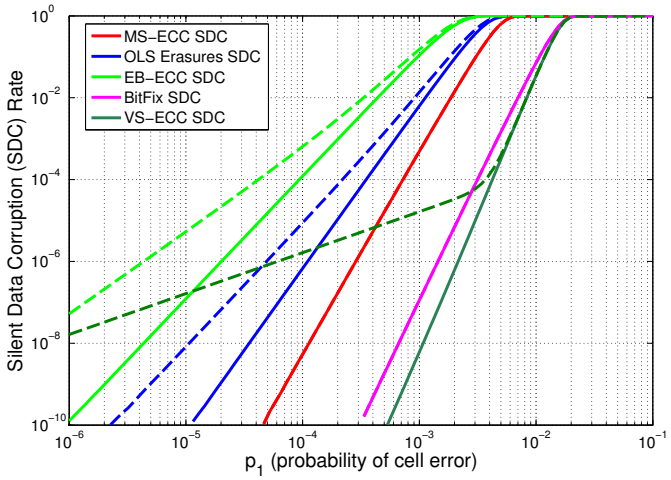

Fig. 7. SDC rates for different fault rates p_1 for five techniques in the cases of $p_2 \in \{0, 0.1\}$. The fault probability of the side information is $p_3 = p_1 \times 10^{-5}$. This case models relatively reliable side information storage.

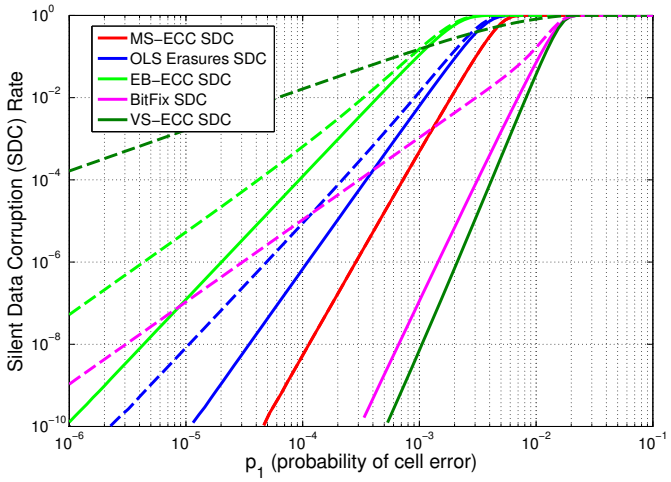

Fig. 8. SDC rates for different fault rates p_1 for five techniques in the cases of $p_2 \in \{0, 0.1\}$. Here, the fault probability of the side information is $p_3 = p_1 \times 10^{-1}$. This case models very unreliable side information storage.

V. Conclusion

In this work, we introduced a general framework for error-correcting techniques in on-chip memories. The proposed framework defines a rich design space and suggests different approaches when selecting techniques for a particular set of parameters or targets. As a proof-of-concept, we showed that erasure-based schemes are less sensitive to BIST detection failures and are therefore suitable for systems facing this issue.

For future work, there are many possible applications of the framework to system design. The framework can also be extended in various ways. For example, we may seek a principled technique or algorithm to derive allowable scheme parameters for a fixed cost. In addition, we can perform a theoretical analysis of the framework. Although a full optimization is computationally intractable, the framework can be analyzed and results derived for extreme cases. The resulting insights can be used as heuristics helpful in performing system design.

REFERENCES

[1] A. R. Alameldeen, I. Wagner, Z. Chishti, W. Wu, C. Wilkerson, and S.-L. Lu, "Energy-efficient cache design using variable-strength error-correcting codes," in *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, (New York, NY, USA), pp. 461–472, ACM, 2011.

[2] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, "Trading off cache capacity for reliability to enable low voltage operation," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, (Washington, DC, USA), pp. 203–214, IEEE Computer Society, 2008.

[3] Z. Chishti, A. Alameldeen, C. Wilkerson, W. Wu, and S.-L. Lu, "Improving cache lifetime reliability at ultra-low voltages," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 89–99, 2009.

[4] D. Palframan, N. S. Kim, and M. Lipasti, "ipatch: Intelligent fault patching to improve energy efficiency," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 428–438, Feb 2015.

[5] J. Abella, J. Carretero, P. Chaparro, X. Vera, and A. Gonzalez, "Low vccmin fault-tolerant cache with highly predictable performance," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 111–121, Dec 2009.

[6] P. P. Shirvani and E. J. McCluskey, "Padded cache: a new fault-tolerance technique for cache memories," in *VLSI Test Symposium, 1999. Proceedings. 17th IEEE*, pp. 440–445, IEEE, 1999.

[7] A. Ansari, S. Feng, S. Gupta, and S. A. Mahlke, "Archipelago: A polymorphic cache design for enabling robust near-threshold operation.," in *HPCA*, pp. 539–550, IEEE Computer Society, 2011.

[8] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-l. Lu, "Reducing cache power with low-cost, multi-bit error-correcting codes," *SIGARCH Comput. Archit. News*, vol. 38, pp. 83–93, June 2010.

[9] J. Kim, H. Yang, M. P. Mccartney, M. Bhargava, K. Mai, and B. Falsafi, "Building fast, dense, low-power caches using erasure-based inline multi-bit ecc," in *Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim International Symposium on*, pp. 98–107, IEEE, 2013.

[10] J. Kulkarni and K. Roy, "Ultralow-voltage process-variation-tolerant schmitt-trigger-based sram design," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 20, pp. 319–332, Feb 2012.