A Framework for Coded Computation

ISIT 2008

Eric Rachlin and John E. Savage

Department of Computer Science
Brown University
Providence, RI

Fault-Tolerant Computing

- For 60 years the overwhelming success of digital computing has been facilitated by ever-shrinking, highly reliable hardware.

Example

If a 3GHz, 10^9 gates/processor, 1,024-processor computer runs for 1 year with a 1% chance of failure, the gate failure rate satisfies $p_f \leq 10^{-30}$!

- As feature sizes shrink and the number of cores per chip increase, it becomes increasingly burdensome to maintain such an astronomically high level of reliability.
- Nanoscale devices, multicore architectures, and distributed computing all suggest an impending need for fault-tolerant computation.

Computation vs. Communication

- Beginning with von Neumann in 1956, many theorists have studied how to implement an arbitrary circuit, C, with unreliable gates.
  - If gates fail independently at random with probability p_f, von Neumann asked whether we can construct a circuit C' such that for any input, $\mathbf{x}$, $Prob(C(\mathbf{x}) \neq C'(\mathbf{x})) = \delta$, for some $\delta < 1/2$.
  - His main idea: Repeat each gate in C many times and periodically suppressed errors by applying constant-size majority gates to random subsets of repeated outputs.
  - Subsequent analysis showed that $|C'|/|C| = \Omega(\log |C|)$.
- This result contrasts sharply with digital communication.
  - Repetition is a very inefficient error control mechanism.
  - We use a reliable encoder and decoder to control errors.

Differential Reliability

- How can we use ideas from coding theory to compute reliably?
  - Build circuits using both reliable and unreliable gates.
- **Differential reliability** allows expensive, large, power-hungry, highly-reliable gates to "supervise" less reliable technology. This allows us to:
  - Encode the input (and decode the output) of a lengthy computation. This is called **Coded Computation**.
  - Exploit the fact that many algorithms have relatively simple checks. This is used in **Algorithm-Based Fault Tolerance**.
- In this talk we investigate coded computation, since the latter approach is highly algorithm specific.

Some Previous Work

- The earliest work on coded computation considered only bitwise operations performed on pairs of codewords. This is overly restrictive.
- Later certain algorithm specific encodings were considered
  - Arithmetic codes for addition and multiplication
  - Check-sums for Matrix operations.
- More recent work by Spielman has suggested that a more general approach is worth pursuing.

Our Model of Unencoded Computation

- Our goal is to "encode" the computation performed by a circuit, C.
- We organize C into levels (slices) and consider a single level of C.

Computation on Circuit Slices

- Consider binary inputs $\mathbf{x}, \mathbf{y} \in \{0,1\}^k$ and "instructions" $\mathbf{w} \in \mathcal{I}$. The computation performed in a slice of k 2-input gates is

$$\mathbf{z} = \boldsymbol{\kappa}^{(\mathbf{k})}(\mathbf{x}, \mathbf{y}, \mathbf{w}) = (\kappa(x_1, y_1, w_1), \ldots, \kappa(x_k, y_k, w_k))$$

- More generally, if $\mathbf{z_t}$ is the output of level t, the computation performed by the next level can be expressed as,

$$\mathbf{z_{t+1}} = \boldsymbol{\kappa}^{(\mathbf{k})}(\pi_{1,t+1}(\mathbf{z_t}), \pi_{2,t+1}(\mathbf{z_t}), \mathbf{w_{t+1}})$$

- Here each $\pi_{i,t+1}$ is a permutation describing which outputs from level t correspond to which inputs at level $t+1$.

A Naive Approach

- For simplicity, consider $\kappa(x, y, w) = \text{NAND}(x, y)$. In otherwords, all gates are NAND gates. Let $\mathbf{z} = \text{NAND}^{(k)}(\mathbf{x}, \mathbf{y})$.
- To provide fault-tolerance, we would like to identify an encoding function, $E : \{0, 1\}^k \mapsto \mathcal{A}^n$, and a function F such that:

$$E(\mathbf{z}) = F(E(\mathbf{x}), E(\mathbf{y}))$$

- Easy! Just decode $E(\mathbf{x})$ and $E(\mathbf{y})$, compute $\mathbf{z}$, then encode $\mathbf{z}$. But this doesn't guarantee fault-tolerance and the overhead is large.
- F should be simple, fault-tolerant, errors musn't propagate too much.
- Ideally each output would only depend on a small number of inputs.
  - For example, can F be of the form $F = \mathbf{f^{(k)}}$ for some function f?

A Lower Bound

Theorem

Let C be an $[n, k, d]_{\mathcal{A}}$ code with encoding function $E : \{0,1\}^k \mapsto \mathcal{A}^n$ and minimum distance d, and let F be a function such that

$$F(E(\mathbf{x}), E(\mathbf{y})) = E(\text{NAND}^{(k)}(\mathbf{x}, \mathbf{y})).$$

If each output symbol of F is a function of at most c inputs of $E(\mathbf{x})$ and $E(\mathbf{y})$, the following inequality must hold:

$$n \geq kd/(c \log_2 |\mathcal{A}|)$$

- When $c = 1$, this bound implies that we cannot do better than using a repetition code.
- NAND can be replaced with any other function whose output only sometimes depends on a given input (e.g. OR, but not XOR).
- This theorem is a substantial generalization of several old results.

A More General Approach

- Our lower bound shows that it is not generally possible to select a code, C, with large minimum distance, and identify a very simple F such that $F(E(\mathbf{x}), E(\mathbf{y}), E(\mathbf{w})) = E(\boldsymbol{\kappa}^{(\mathbf{k})}(\mathbf{x}, \mathbf{y}, \mathbf{w}))$
- Instead we can consider a second (larger) code, C^*, with encoding function E^*, and identify a function Φ such that

$$\boldsymbol{\Phi}^{(\mathbf{n})}(E(\mathbf{x}), E(\mathbf{y}), E(\mathbf{w})) \quad = \quad E^*(\boldsymbol{\kappa}^{(\mathbf{k})}(\mathbf{x}, \mathbf{y}, \mathbf{w}))$$

  where $\boldsymbol{\Phi}^{(\mathbf{n})}(\mathbf{u},\ \mathbf{v},\ \mathbf{t}) = (\phi(u_1, v_1, t_1), \ldots, \phi(u_n, v_n, t_n))$.
- To obtain F from Φ, we must then "transcode", meaning project $\boldsymbol{\Phi}^{(\mathbf{n})}(E(\mathbf{x}), E(\mathbf{y}), E(\mathbf{w}))$ back to C.

A More General Approach (cont.)

$$\mathbf{\Phi}^{(\mathbf{n})}(E(\mathbf{x}), E(\mathbf{y}), E(\mathbf{w})) = E^*(\boldsymbol{\kappa}^{(\mathbf{k})}(\mathbf{x}, \mathbf{y}, \mathbf{w})).$$

- Since C^* is larger than C, a given output can have multiple encodings.
- To ensure that Φ performs the desired computation, we choose C and C^* to be systematic. Then we have $\Phi = \kappa$ on the information symbols. As shown on next slide, this is obtained using interpolation.
- Fault-tolerance relies on the error correcting capability of C^*, as well as the structure of the transcoding operation.
- To transcode from C^* back to C in a fault-tolerant manner, Spielman suggested using 2D codes. This requires that C and C^* be linear.

Extension Polynomials

- In a linear code C, $\mathbf{x} \in \mathcal{F}^k$ is encoded as $E(\mathbf{x}) \in \mathcal{G}^n$, where $\mathcal{F} \subseteq \mathcal{G}$ are both finite fields.
- If E is chosen such that C is systematic, then we can define Φ using interpolation over the values of $\mathcal{F}^3$ on which κ is defined.
- This ensures that $\mathbf{\Phi^{(n)}}(E(\mathbf{x}), E(\mathbf{y}), E(\mathbf{w}))$ computes $\kappa^{\mathbf{(k)}}(\mathbf{x}, \mathbf{y}, \mathbf{z})$, along with $n - k$ check symbols in $\mathcal{G}$.

Example

Let $\mathcal{F} = \{0, 1\}$ and $\mathcal{F} \subseteq \mathcal{G}$. If $\kappa(x, y, 1) = \text{NAND}(x, y)$ and $\kappa(x, y, 0) = x$, then we have:

$$\Phi(x, y, w) = w(1 - xy) + (1 - w)x.$$

If κ had a larger domain (for example, more than two instructions), we could chose a larger $\mathcal{F}$, or use more than three variables.

Selection of C and C^*

- In order to select C and C^* such that $\mathbf{\Phi}^{(\mathbf{n})}(E(\mathbf{x}), E(\mathbf{y}), E(\mathbf{w})) \in C^*$, Spielman suggested Reed-Solomon codes.
- We observe that Reed-Muller codes are also viable, as well as other codes based on bounded-degree polynomials.

Linear Codes Based on Polynomials

- Let $C_{\mathcal{G}}(m, r)$ denote a code in which each codeword is value of r-degree, m-variable polynomial $p(v_1, ..., v_m)$ evaluated at n points in $P \subseteq \mathcal{G}^m$.
- For $C_{\mathcal{G}}(m, r)$ to be systematic, $p(v_1, ..., v_m)$, which encodes $E(\mathbf{x})$, is an interpolation polynomial, that is, the first k values of $p(v_1, ..., v_m)$ for $(v_1, ..., v_m) \in S$ is $\mathbf{x}$ where $S \subset P$, $|S| = k$.
- In a Reed-Muller Code, $\mathcal{G} = \{0, 1\}$, $n = 2^m$, and $k = \Sigma_{i=0}^{r} \binom{m}{i}$. In a Reed-Solomon Code, $m = 1$, $n \leq |\mathcal{G}|$, and $k = r + 1$.

Application of $\mathbf{\Phi^{(n)}}$ and Transcoding

- If $E(\mathbf{x}), E(\mathbf{y}), E(\mathbf{w}) \in C_{\mathcal{G}}(m, r)$, and $\Phi(x, y, w)$ has degree s, then:

$$\mathbf{\Phi^{(n)}}(E(\mathbf{x}), E(\mathbf{y}), E(\mathbf{w})) \in C_{\mathcal{G}}(m, rs)$$

- If rs isn't too large, we still have the ability to correct for errors.
  - In the case of Reed-Muller codes, $d = 2^{m-rs}$.
  - For arbitrary $C_{\mathcal{G}}(m, r)$, we if $S \subseteq \mathcal{G}$ and $P = S^m$, then $d \geq (1 - (rs/|S|))$
- To transcode, we can treat $C_{\mathcal{G}}(m, r)$ and $C_{\mathcal{G}}(m, rs)$ as a 2D code, and transcode first by rows, then by columns.

Data Movement

- Using polynomial codes, we can encode $\mathbf{x}$ and $\mathbf{y}$, the inputs to a level of a circuit, and apply $\mathbf{\Phi^{(n)}}$, then transcode the result.
- Before the encoded output, $E(\mathbf{z})$, can be supplied as input to the next level of the circuit, it must be copied, and each copy must be permuted. Recall our model:

$$\mathbf{z_{t+1}} = \boldsymbol{\kappa}^{\mathbf{(k)}}(\pi_{1,t+1}(\mathbf{z_t}), \pi_{2,t+1}(\mathbf{z_t}), \mathbf{w_{t+1}})$$

- Which when encoded should become:

$$E(\mathbf{z_{t+1}}) = T(\mathbf{\Phi^{(k)}}(\pi_{1,t+1}(E(\mathbf{z_t})), \pi_{2,t+1}(E(\mathbf{z_t})), E(\mathbf{w_{t+1}})))$$

where T denotes the transcoding operation.

- If $\pi_{1,t+1}$ and $\pi_{2,t+1}$ are arbitrary, they cannot necessarily be applied them directly. Instead they can be decomposed into permutations that can be applied.

Permuting Codewords

- Polynomial codes are closed under a number of permutations. For example, both Reed-Solomon and Reed-Muller codes can both be permuted along the dimensions of a hypercube.
- Additional permutations can be applied during 2D transcoding.
- If needed, an arbitrary permutation can be realized as a series of allowed permutations.
  - Hypercube-style data movement is enough to implement arbitrary permutations via a switching network formed from back-to-back butterfly graphs.
  - This network can in turn be implemented using only cyclic shifts via a shuffle-exchange protocol.

Putting it all together

- Given an arbitrary circuit C, it can be converted to a leveled circuit C_R such that all gates have fan-in and fan-out 2.
- Using the techniques we have presented, the input to C_R can be encoded, and the computation performed by each level can be made fault-tolerant.
- Between each step of coded computation, transcoding is required. This constitutes the major overhead of this approach.
- Still, this approach can still potentially outperform basic reputation if C is sufficiently deep. The overhead required is a logarithmic (or polylogarithmic) in the width of C, where as with repetition it is logarithmic in $|C|$.

Conclusions and Future Work

- Coded-computation appears to have the potential to outperform repetition, but currently the overhead is still high.
- Reed-Muller codes allow us to implement the binary operations of a circuit using binary codes. This is a significant improvement over Spielman's Reed-Solomon based-approach.
- The structure of specific computations may allow them to be encoded with lower overhead.