

# Building Robust Systems for the Energy Constrained Future: Application and Algorithm Aware Approaches

---

**Rakesh Kumar**

*University of Illinois*

*Urbana-Champaign*



# The Reliability Problem

---



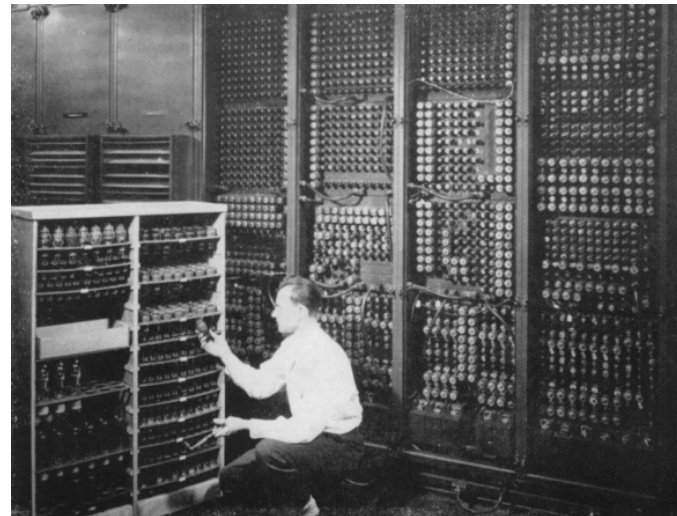
## PROBABILISTIC LOGICS AND THE SYNTHESIS OF RELIABLE ORGANISMS FROM UNRELIABLE COMPONENTS

J. von Neumann

logical structures.

Our present treatment of error is unsatisfactory and ad hoc.

It is a common belief, indeed even many years, that error should b



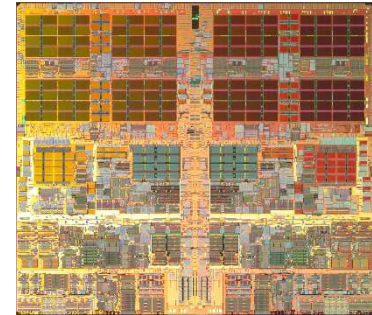
Eniac (1945)

# Conventional Solutions

---



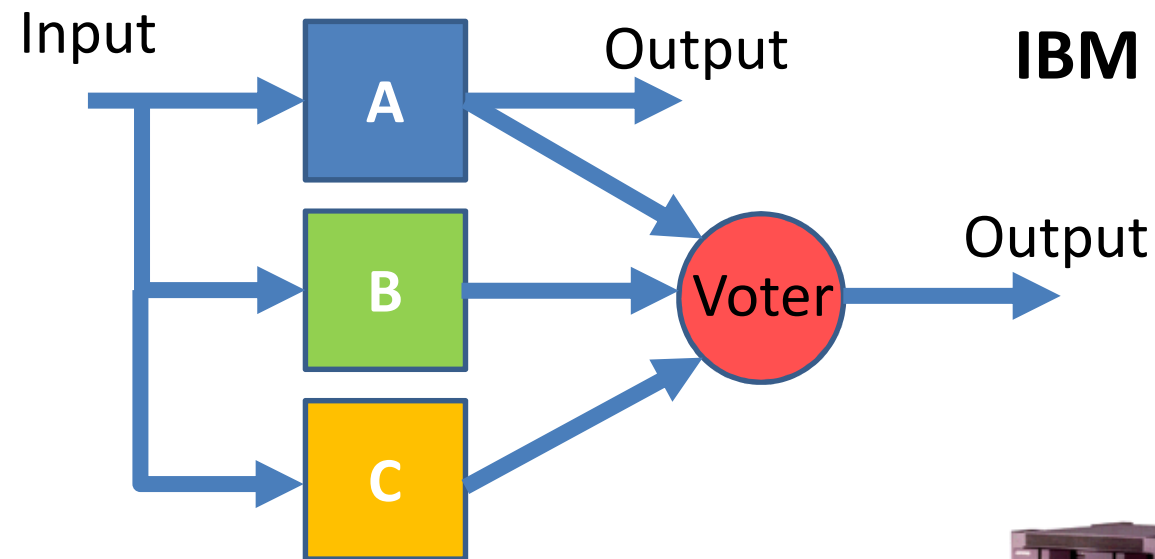
**Voyager (1977)**



**IBM G5**



**Boeing 777 (1994)**



**Compaq Himalaya**

# The Power Wall

## Mobile Systems



## High Performance Systems



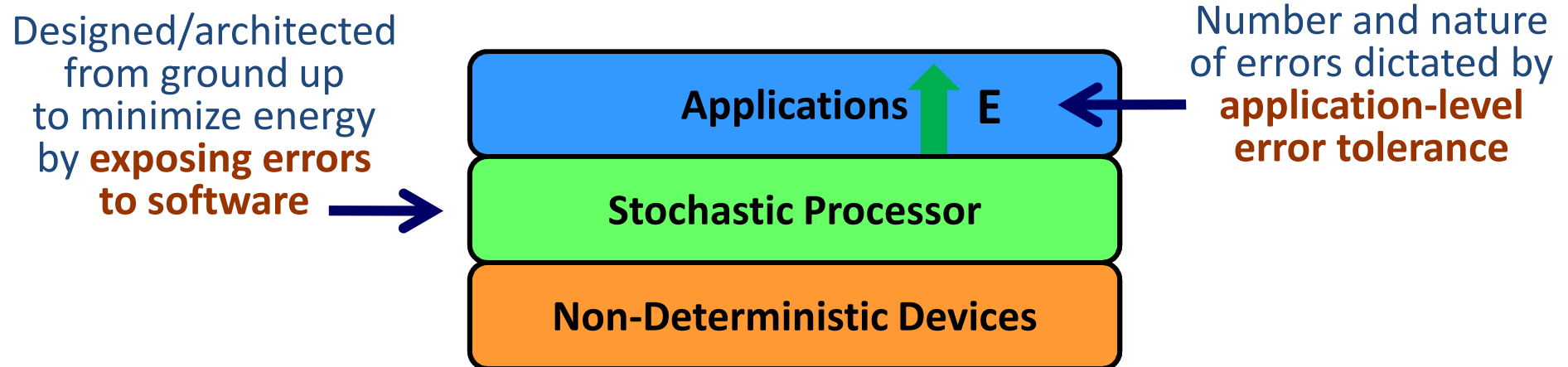
Clearly, low cost resilience techniques are needed



4 million devices by 2020

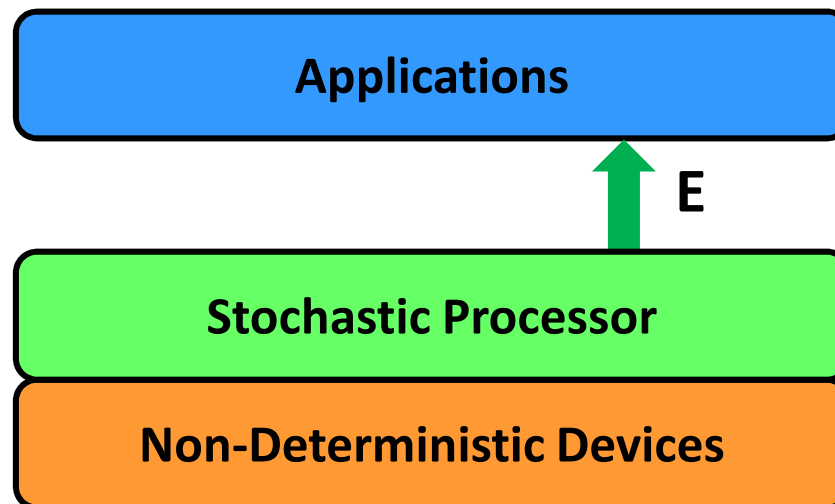
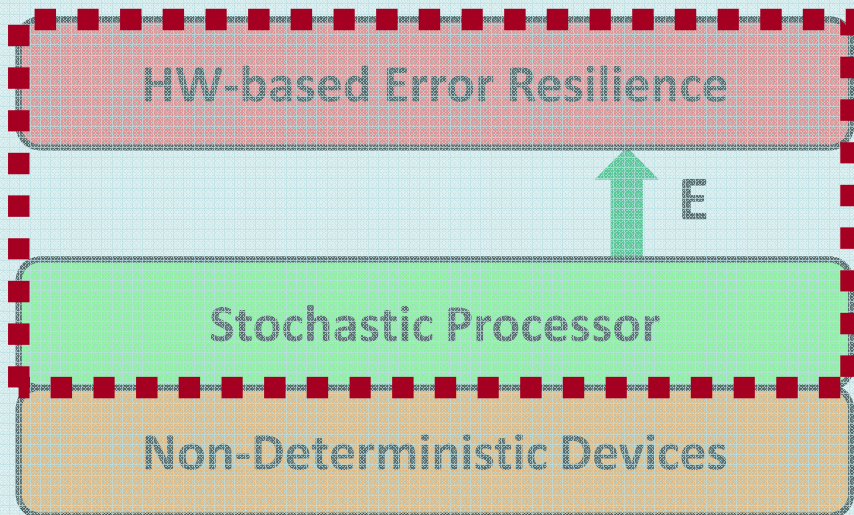
# Stop ignoring non-determinism, error tolerance

---



**Our research focuses on approaches to architect, design, and program stochastic processors**





# Application and Algorithm Awareness

---

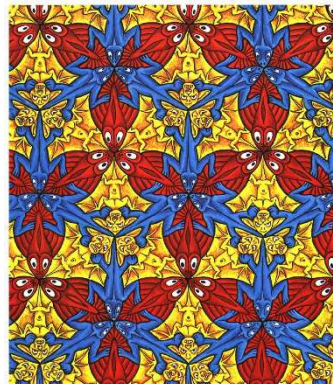
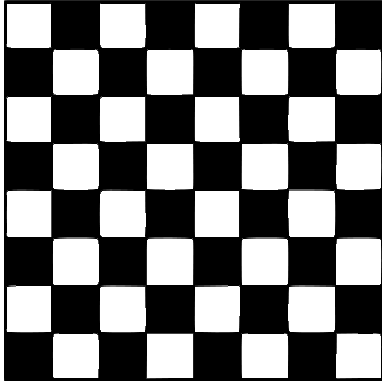
- Natural Error Tolerance



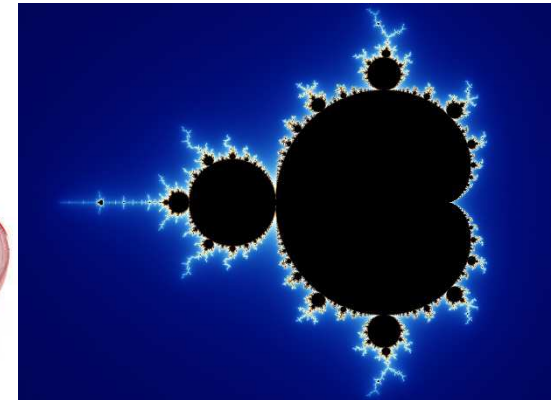
# Application and Algorithm Awareness

---

Spatial Reuse



Temporal reuse



Fault containment

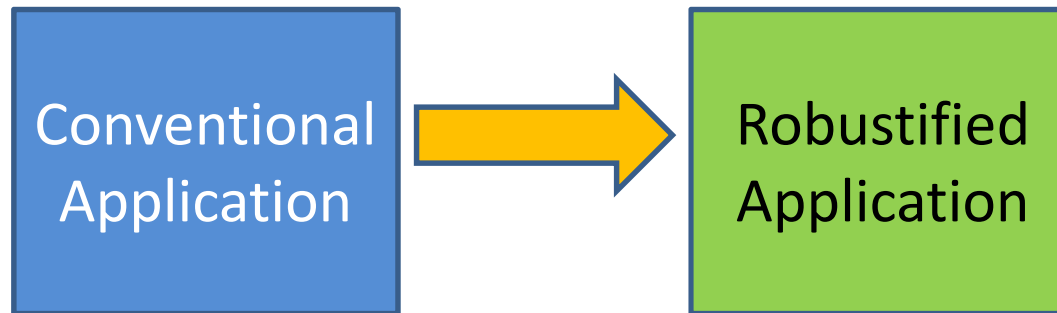


**This talk:** application and algorithm-aware approaches  
for low cost error resilience



# Application Robustification (AR)

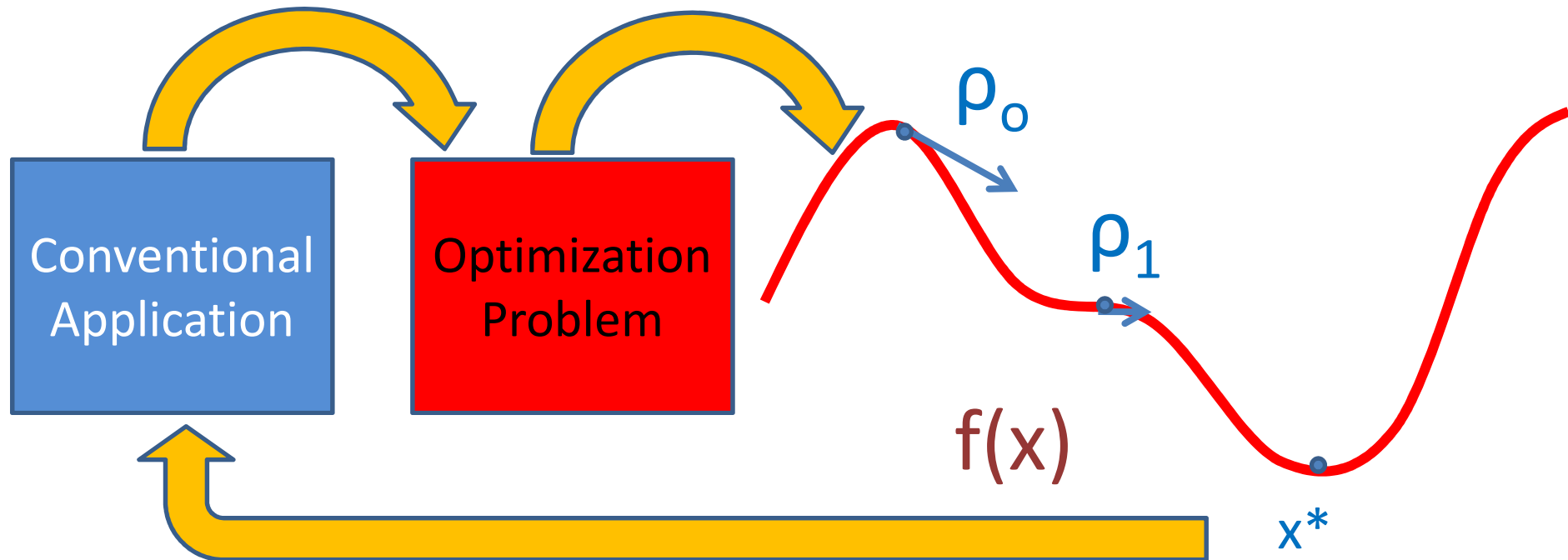
---



- **Goal:** Redesign applications to produce acceptable output in presence of errors
  - Same as output without errors for most applications
  - Within certain tolerance for other applications

# An Optimization-based Approach to AR

---



## Primary Issues

- How to construct  $f(x)$  when we don't know  $x^*$ ?
- What is the most efficient solver for  $f(x)$ ?

# Example 1: System of Equations (SOE)

---

- **Problem:** Solve a system of equations.

$$\begin{aligned} a_0x_0 + a_1x_1 + a_2x_2 &= y_0 \\ b_0x_0 + b_1x_1 + b_2x_2 &= y_1 \\ c_0x_0 + c_1x_1 + c_2x_2 &= y_2 \end{aligned} \quad \begin{bmatrix} a_0 & a_1 & a_2 \\ b_0 & b_1 & b_2 \\ c_0 & c_1 & c_2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}$$

- Traditional solution methods: SVD, QR, Cholesky factorization.

# Robustified System of Equations

---

$$Ax = b$$

$$\| Ax - b \|^2$$

- Equivalent Optimization Problem:

$$\begin{aligned} \min \quad f(x) &= \| Ax - b \|^2 \\ &\approx x^T A^T A x - 2b^T A x \end{aligned}$$

Appropriate Solver Used for ***Quadratic*** Problem

## Example Formulation 2: Sorting

---

- What is sorting?
  - Finding the correct relative position of each element in the unsorted list. [*Permutation matrix*]

- Example

- Input  $u = [5, 2, 8]^T$
  - $X$ : 3x3 Permutation Matrix

Permutation to reverse

$$Xu = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 5 \\ 2 \\ 8 \end{bmatrix} = \begin{bmatrix} 8 \\ 2 \\ 5 \end{bmatrix}$$

Permutation to sort

$$Xu = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 5 \\ 2 \\ 8 \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ 8 \end{bmatrix}$$

- Robustified Formulation

- The list which arranges the elements of list in ascending order will minimize the product  $-v(Xu)$

$$v = [1 \dots n]^T$$

$$\min_{x \in R^{n \times n}} -v^T Xu$$

# Robustified Sorting Example (contd)

---

**Unsorted list:**  $u = [5 \ 2 \ 8]^T$

$v = [1 \ 2 \ 3]^T$

Original Permutation:

$$X_{orig} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Correctly Sorted Permutation :

$$X_{sort} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Objective:

$$-v^T X_{orig} u = -33$$

Objective:

$$-v^T X_{sort} u = -36$$

**(lower than unsorted)**



# Robustified Sorting (contd)

---

- Constraints need to be set up correctly:

$$\begin{aligned} \min_{x \in R^{n \times n}} & -v^T Xu \\ \text{s.t. } & X_{ij} \geq 0, \sum_i X_{ij} \leq 1, \sum_j X_{ij} \leq 1 \end{aligned}$$

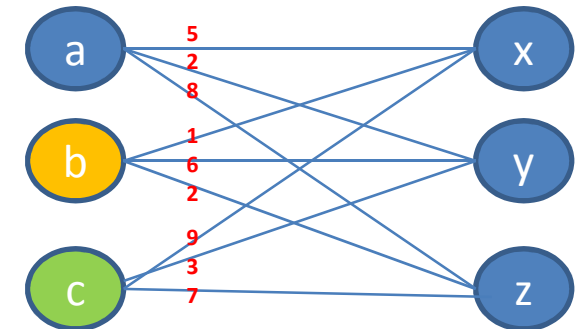
- As unconstrained problem:

$$-v^T Xu + \lambda \sum_{ij} [X_{ij}]_+^2 + \lambda \sum_i [\sum_j X_{ij} - 1]_+^2 + \lambda \sum_j [\sum_i X_{ij} - 1]_+^2$$

Penalty Function

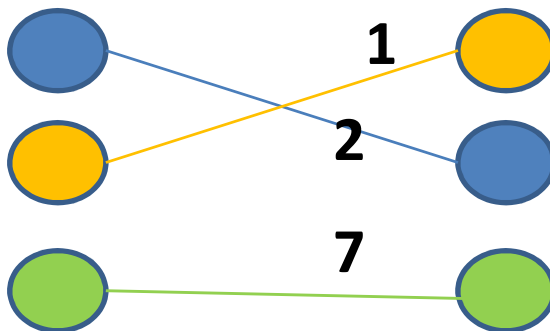
# Example 3: Bipartite Graph Matching

- **Input:**  $W$  is matrix of weights for all edges in the graph

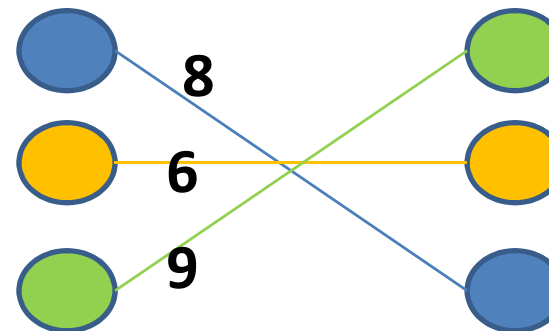


- What is Bipartite Graph Matching?
  - Find the assignment of edges which do not share any vertices and gives the largest total weight.

$$W = \begin{matrix} & \begin{matrix} x & y & z \end{matrix} \\ \begin{matrix} a \\ b \\ c \end{matrix} & \begin{bmatrix} 5 & 2 & 8 \\ 1 & 6 & 2 \\ 9 & 3 & 7 \end{bmatrix} \end{matrix}$$



**Total Weight= 10**



**Total Weight= 23**

## Example 3: Robustified GM

---

- Constraints need to be set up correctly:

$$\begin{aligned} \min_{x \in R^{n \times n}} & - \langle W, X \rangle \\ \text{s.t. } & X_{ij} \geq 0, \sum_i X_{ij} \leq 1, \sum_j X_{ij} \leq 1 \end{aligned}$$

- As unconstrained problem:

$$- \langle W, X \rangle + \lambda \sum_{ij} [X_{ij}]_+^2 + \lambda \sum_i [\sum_j X_{ij} - 1]_+^2 + \lambda \sum_j [\sum_i X_{ij} - 1]_+^2$$

Penalty Function

# Scope of Transformations

---

**SOE:**  $\min_{x \in R^{nx1}} x^T A^T A x - 2b^T A x$  (as Quadratic Program)

**Sort:**  $\min_{x \in R^{nxn}} -v^T X u$  ( as Linear Program)

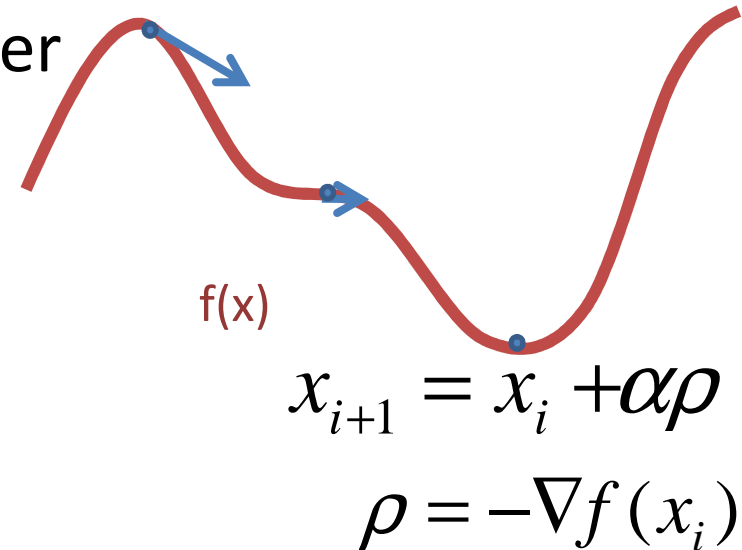
**GM:**  $\min_{x \in R^{nxn}} -\langle W, X \rangle$  ( as Linear Program)

- Large class of problems can be solved as LP.
  - Any polynomial algo can be emulated in polynomial time
- Applicable for harder problems as well!
  - NP, ILP, discrete/combinatorial optimizations

# Identifying the Best Solver

- Desirable attributes of a good solver

- Fast convergence
- Good error tolerance
- Low power
- User controllable degree error tolerance
- Applicable to a wide range of problems



## Option 1: Gradient Descent (GD)

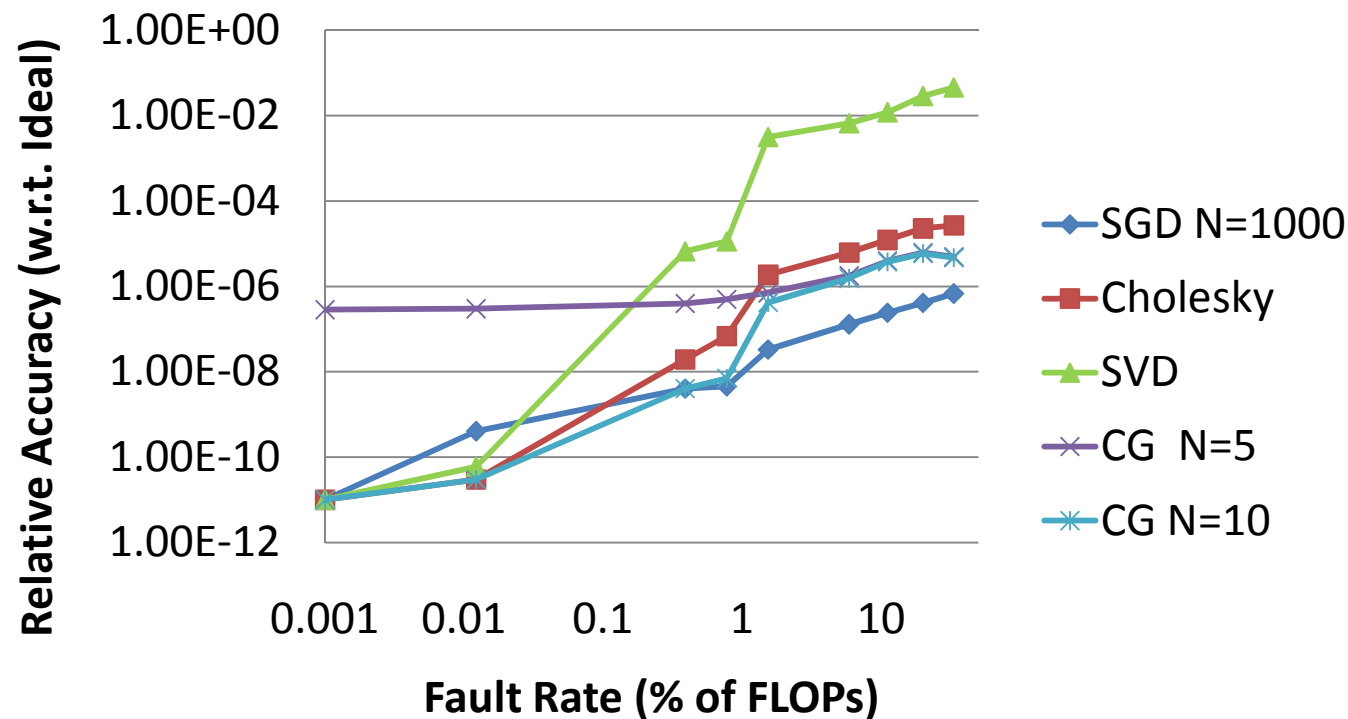
- **Advantage:** shown to be robust under errors
- **Disadvantage:** can take many iterations to converge

## Option 2: Conjugate Gradient(CG)

- **Advantage:** relatively fast
- **Disadvantage:** objective function specific [quadratic]

**Other solvers possible as well: subject of future work**

# System of Equations (100x10) using GD / CG

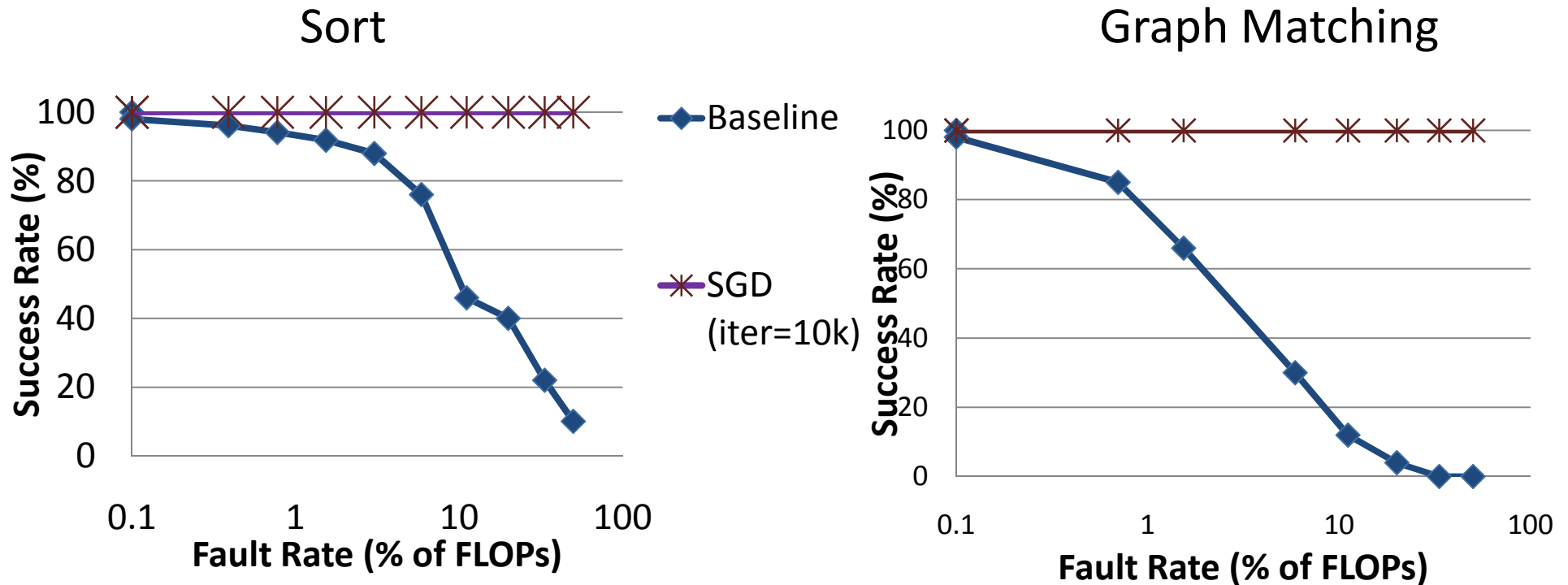


**CG and GD significantly more robust than SVD or QR at high fault rates.**



# Sorting (size=10) and Graph Matching(5x6) using GD

---



**100% Accuracy even for large error rates.**

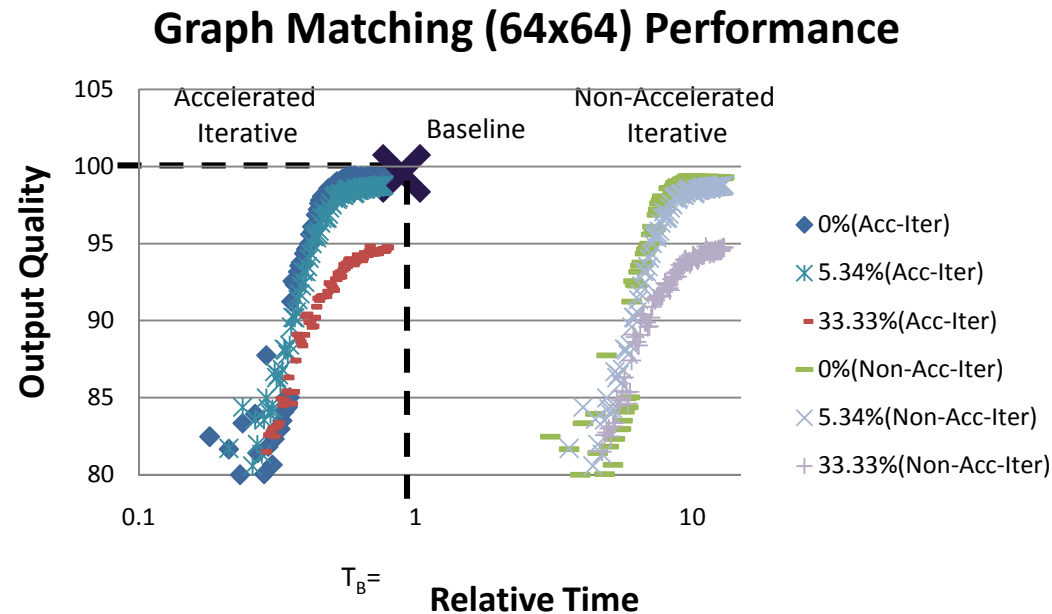
# Stepping Back

**Advantage:** Generality. Many applications can use this formulation

**Disadvantage:** Some applications may take a long time to converge

**Solution:** An accelerator architecture that speeds up linear algebra operations [optimization-based design is often more parallelizable]

Output Quality =  
completeness of  
matching (%)



**Long convergence times still a  
limitation for certain app classes (sort).**

# Sparse Linear Algebra

---

- **Future Applications:** A large class utilize sparse linear algebra algorithms (e.g. graph-based, data mining, and recognition)

- **HPC Applications:** PDE/ODE Solvers

- Linear System Solvers (e.g. Conjugate Gradient)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

- Common linear operation for many of these kernels:

- Matrix Vector Product:  $y = Ax$

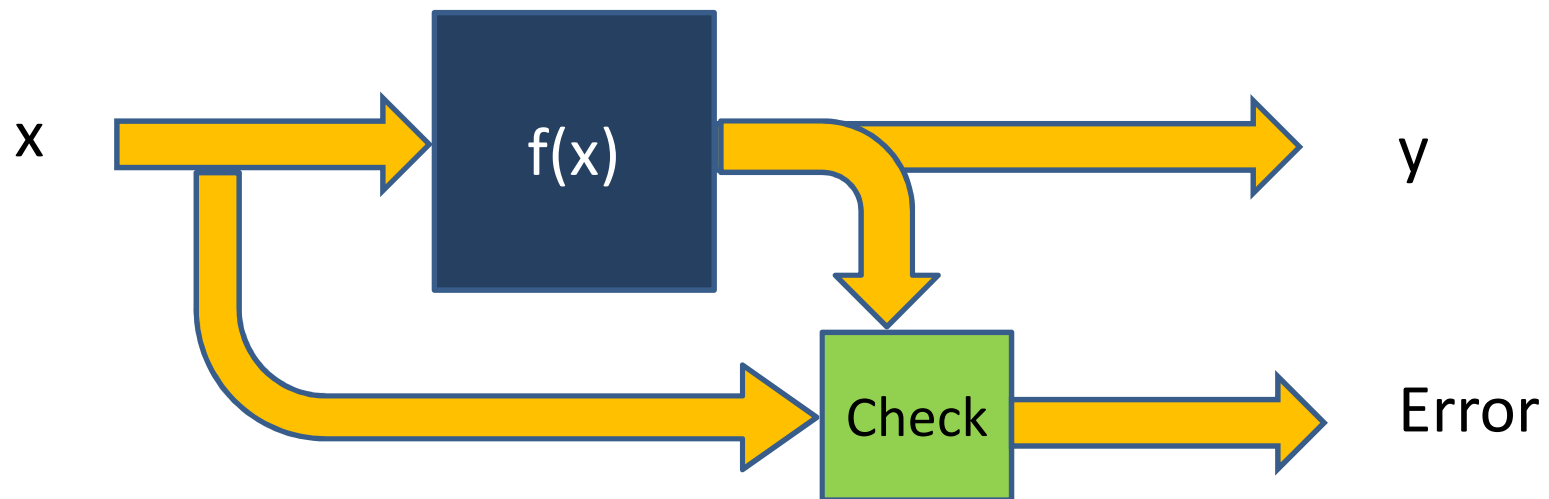
**Goal:** Techniques for fortifying sparse linear algebra for unreliable hardware.

$f(x)$

# Detection for Linear Algebra: Past Approaches

---

Design a low-complexity mathematical invariant that can be used to check computation



Example: **Matrix vector multiplication (MVM)** uses traditional linear error correcting codes to develop invariant

$$\mathbf{A} \mathbf{x} = \mathbf{y}$$

# Past Approach: Matrix Vector Multiplication

---

Sparse  
Matrix

$O(n^2)$

$$\mathbf{A} * \mathbf{x} = \mathbf{y}$$

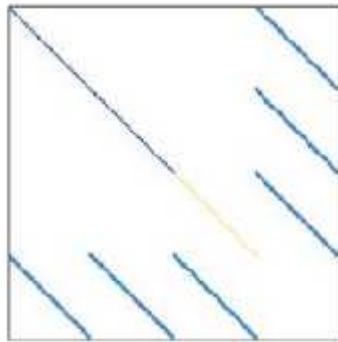
For sparse problems, the complexity of this dense check is identical to the protected operation!  $O(n)$

$$|\mathbf{1}^T(\mathbf{y}) - (\mathbf{1}^T \mathbf{A})\mathbf{x}| > \tau$$

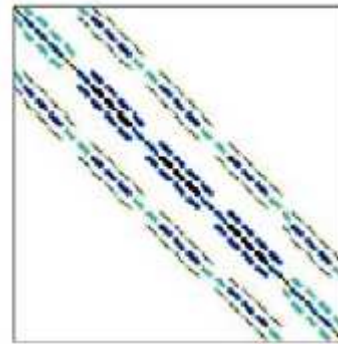
$O(n)$

# Structure in Sparse Problems

---



Qpband



msc00726

$$1^T (Ax) = \underbrace{(1^T A)}_{\text{Sampled (s)}} x \begin{pmatrix} n \\ s \end{pmatrix}$$

**Uniformity in the column sums allows for sampling of the MVM check for these problems.**

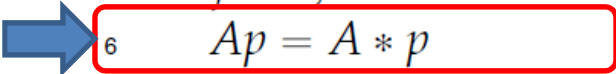


# Frequent Reuse in Sparse Problems

---

- Many linear algebra applications use the same data as a part of many individual operations.

```
1 Input:  $x_0$   
2  $r = b - Ax_0$   
3  $p = r$   
4  
5 for  $j = 1, \dots$  do  
6  $Ap = A * p$   
7  $\alpha = (r, r) / (p, Ap)$   
8  $x = x + \alpha p$   
9  $r_{old} = r$   
10  $r = r - \alpha Ap$   
11  $\beta = (r, r) / (r_{old}, r_{old})$   
12  $p = r + \beta p$   
13
```



# Conditioning: Identity

---

**Key Insight:** Frequent reuse may allow for preconditioning in spite of high setup costs

- How can the check be preconditioned?
  1. Observe: basic approach is special case with a code:  $c=1$
  2. Choose code s.t. checksum is smoothed

$$c^T(Ax) = (c^T A)x = (1^T)x = \sum x$$

## Identity Conditioning

$$c^T A = 1$$

$$\min ||c^T A - 1||$$

# Conditioning: Null

---

- Conditioning can also make the problem more applicable for sampling or clustering

- Choose code that eliminates half of check entirely.  $A^T c = \cancel{1} 0$

Find vector in null space  $A^T c = \sigma u$   
 $\sigma \ll 1$

## Null Conditioning

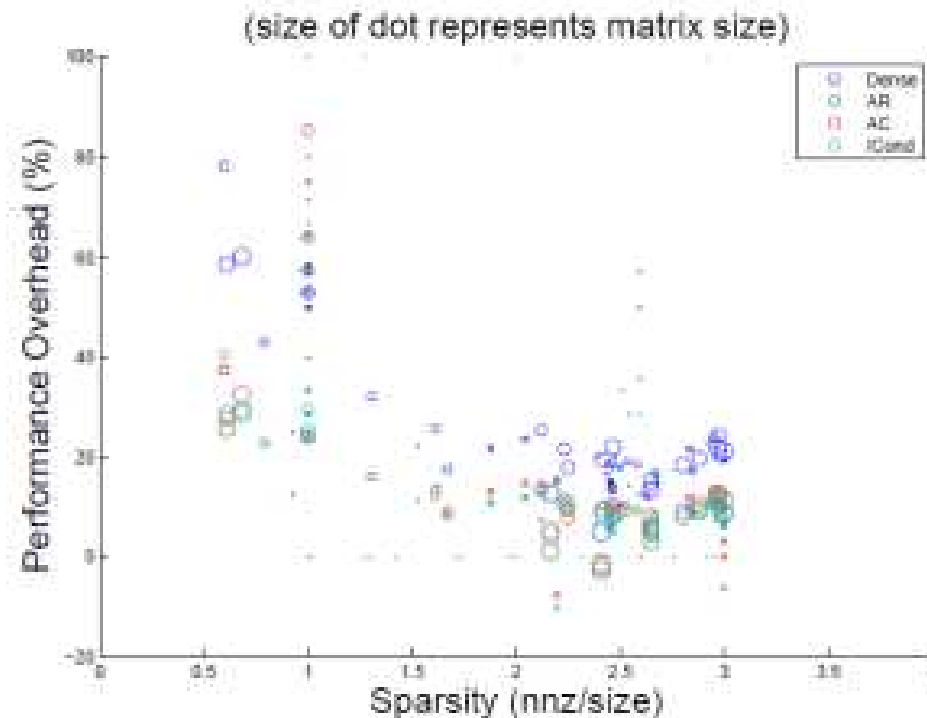
$$A^T c = 0$$

$$c^T (Ax) = 0$$

**Applicable for Dense Problems as well!**

# Matrix Vector Multiplication Results

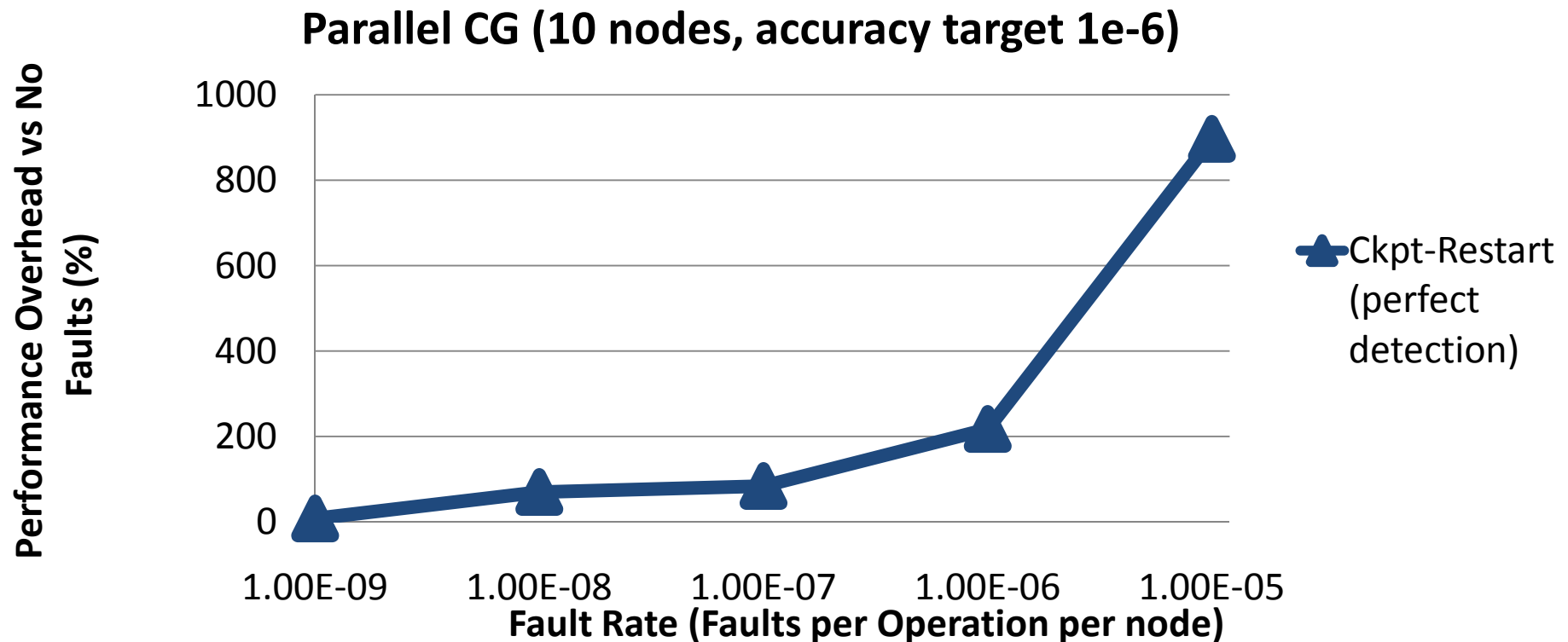
---



**Detection Overhead 50-60% lower than dense checks for sparse problems.**

# Detection Doth Not Efficient Fault Tolerance Make

---

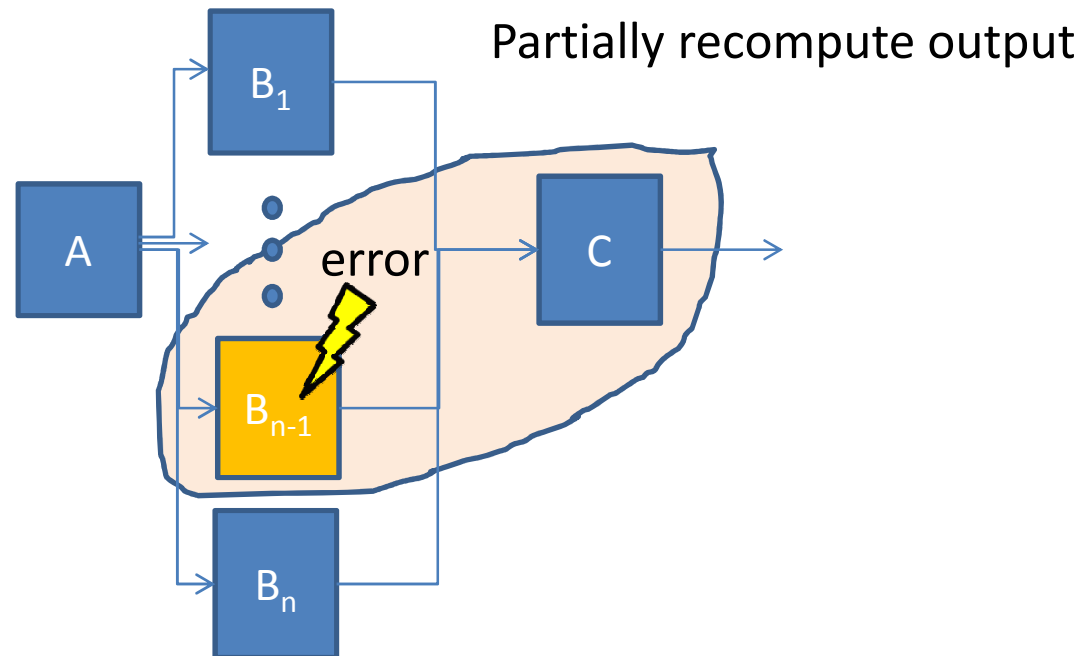


**Under high error rates, the overhead of Error Correction may become prohibitive**

# Partial Recomputation

---

- Insight: outputs/state are usually only partially wrong when faults occur.

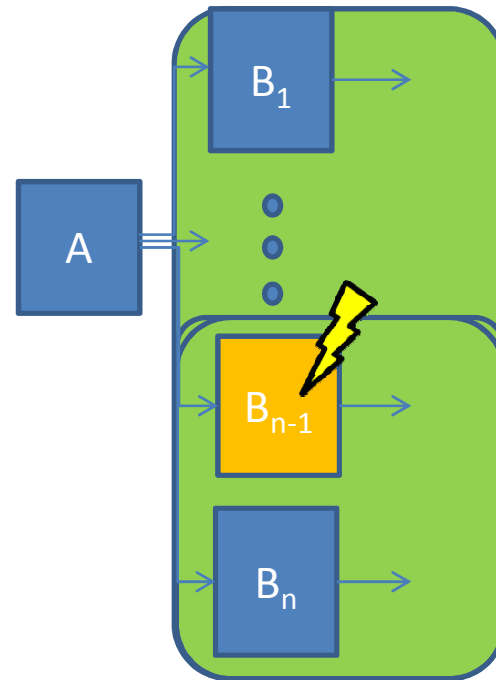


Strategy: Upon detecting an error, only partially recompute the output (i.e. the segment which contains the error)



# Error Localization

---



Low overhead check  
across all outputs

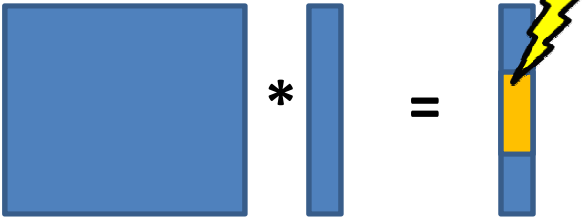
Apply sub-checks

Localized error

**Compute and verify sequence of checks such that errors are localized to specific subcomputations (cone-analysis in hardware).**

# Matrix Vector Multiplication (MVM)

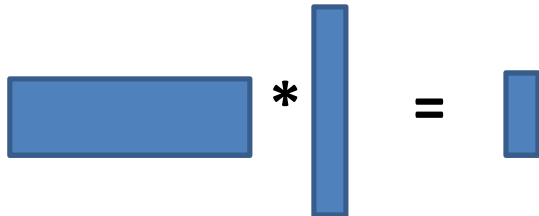
---

$$\mathbf{A} * \mathbf{x} = \mathbf{y}$$


Output error in fraction of  
output vector



Localize errors

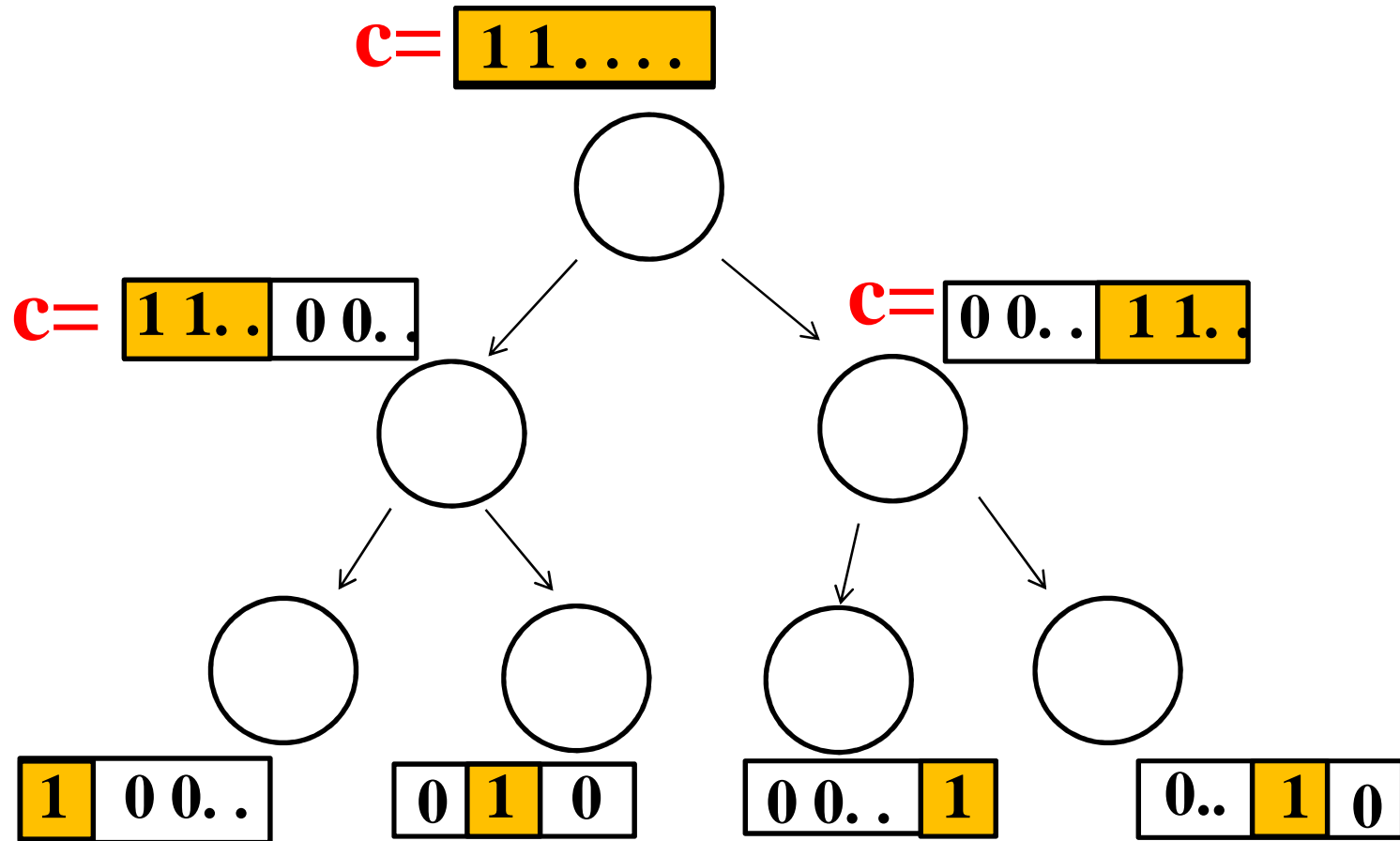
$$\mathbf{A}_s * \mathbf{x} = \mathbf{y}_s$$


Partially recompute output

# Error Localization for Matrix Vector Multiply

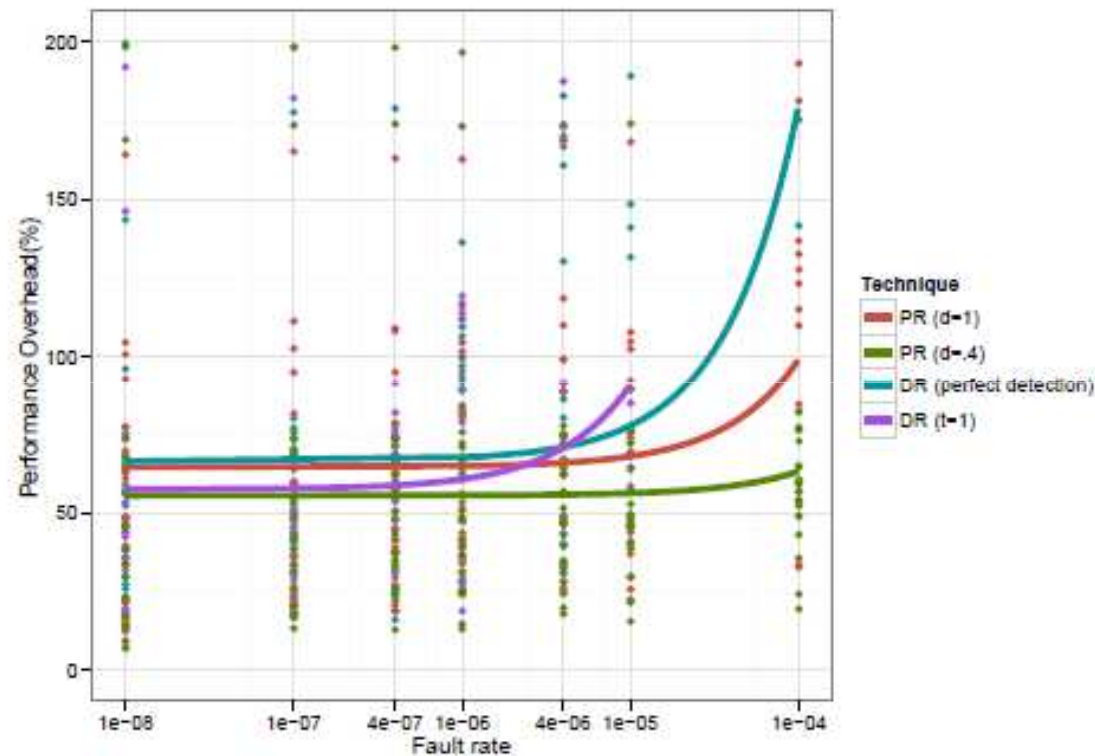
---

$$c^T (Ax) = (c^T A)x$$



# Results with increasing fault rates

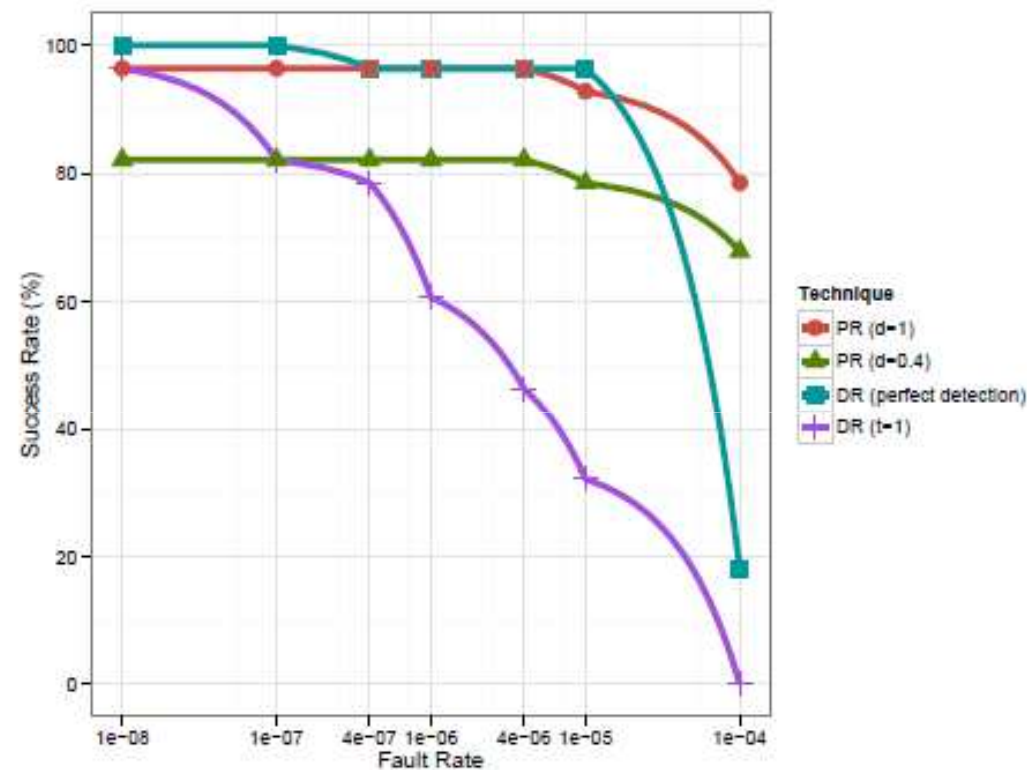
---



**Partial recomputation leads to 2x-3x less overhead for high fault rates.**

# Results with increasing fault rates

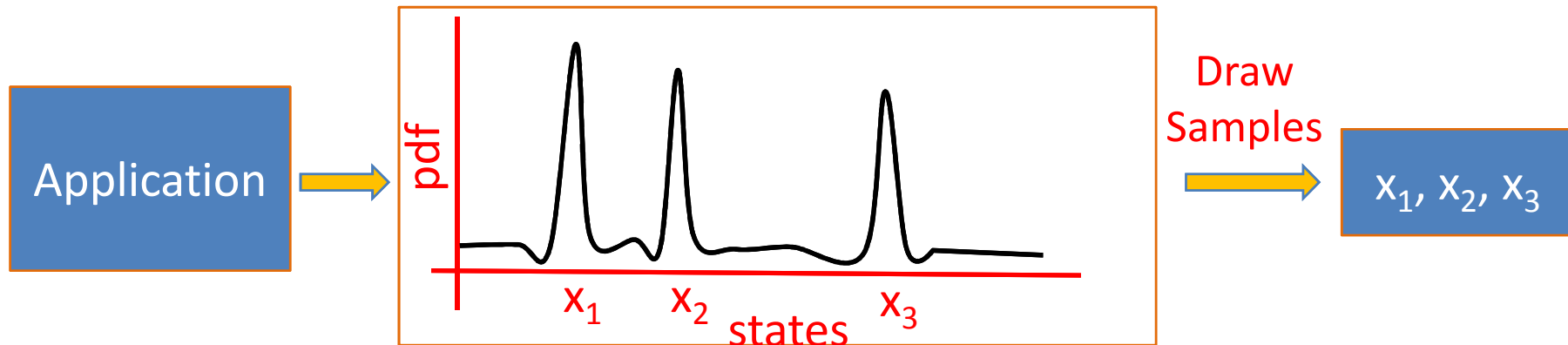
---



**Under high error rates Partial recomputation converges 70% more often given maximum iteration limit.**

# Building Robust Applications via Statistical Inference

## Stochastic Information Processing Framework



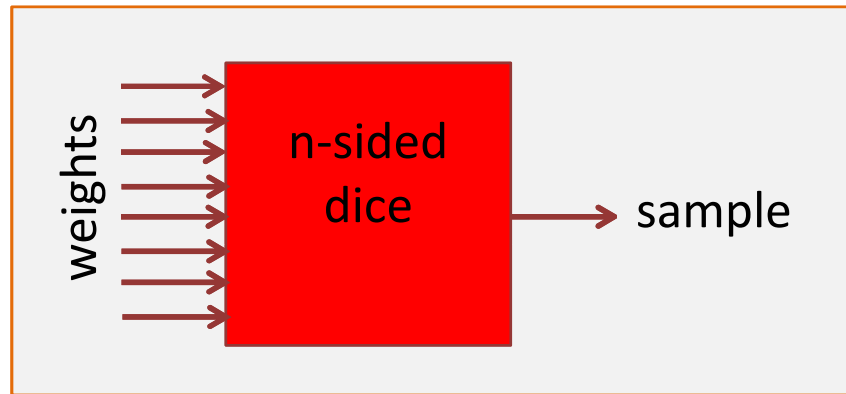
Converting applications to a  
stochastic local search framework

1. How do we generate this distribution for different applications?
2. What sampling techniques should we use?
3. Other issues :  
generality, programmability, completeness, complexity

# Robust and Efficient Architectures

---

- What hardware components do these algorithms map to naturally?



- Which components need to be robust?
- How can these be mapped to nano-blocks?
- Energy, performance, quality tradeoffs

# Summary

---

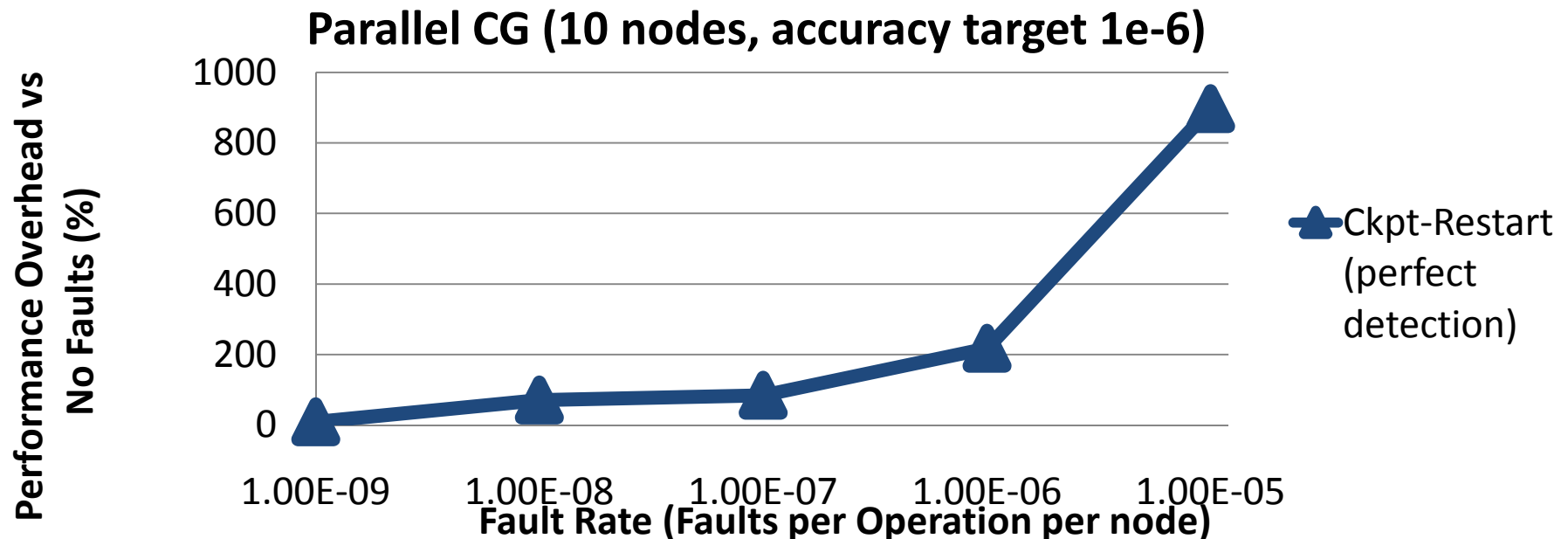
- Conventional Fault Tolerance Techniques not practical for future power-constrained systems
- Even disregarding power constraints, the techniques do not suffice when fault rates are high
  - Too much cost to detection/recomputation
- Applications Robustification
  - Algorithmic Techniques to build inherently robust Applications that “roll forward” on errors
- Error Localization and Partial Recomputation support the same goal



---

# Conventional Fault Tolerance Approaches

- Hardware-based fault tolerance approaches may be impractical for severely power-constrained systems .
  - Duplication and TMR, expensive
  - Typically based on worst-case and conservative designs.
- General software-based approaches may also be impractical.
  - Redundancy-based, costs have been well-documented



## Slide 42

---

### JHP1

What are the X-axis units? Error rate as probability? ie. In each cycle, one of 10 nodes makes a error with probability  $10^{-08}$ . Or is it each node and hence the overall rate is 10-times for the system?

Traditional FT methods:

Hardware - duplication or TMR.

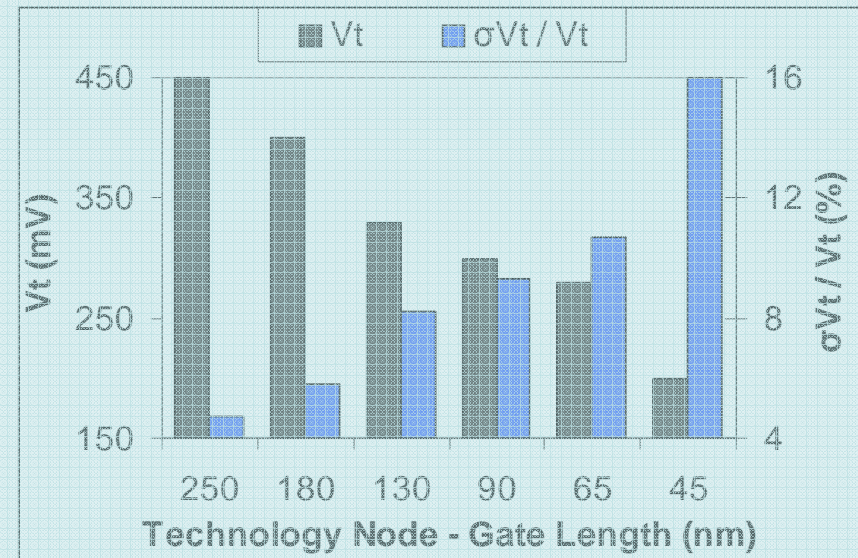
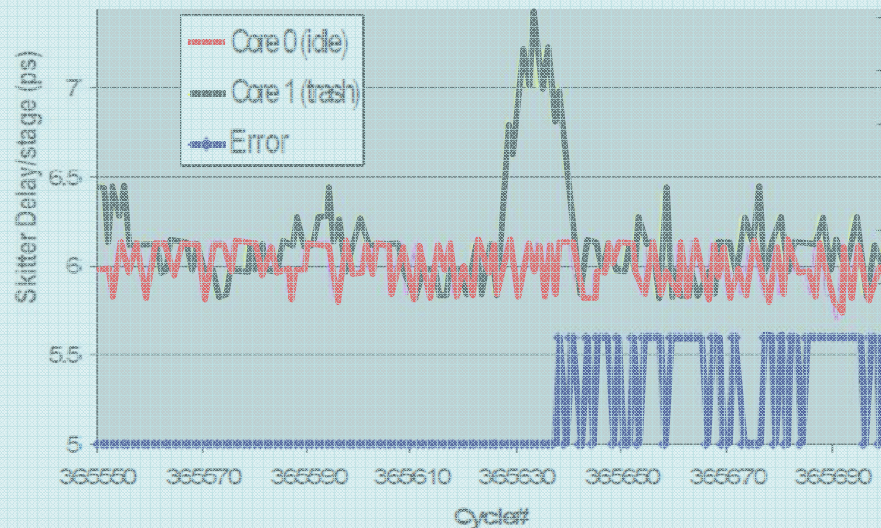
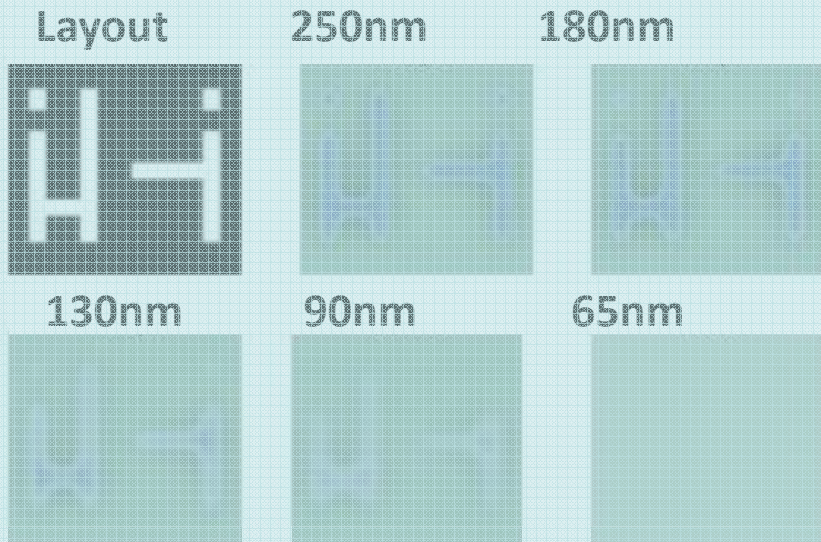
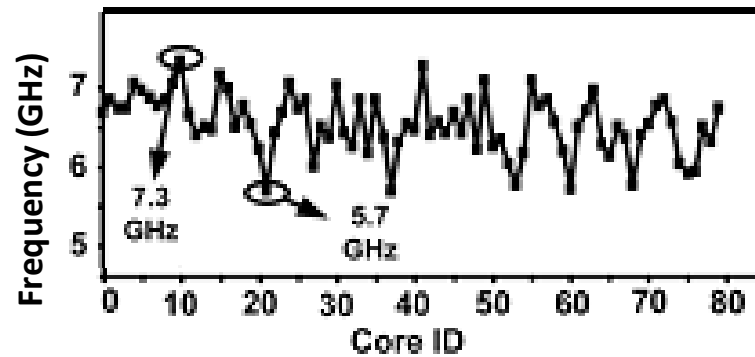
Time: recompute one or more times, same computation or altered implementation of same function.

Information: Coding, Assertion checks

Software methods use Time and/or Informtion redundancies.

Janak H. Patel, 12/7/2012

# The Reliability Problem

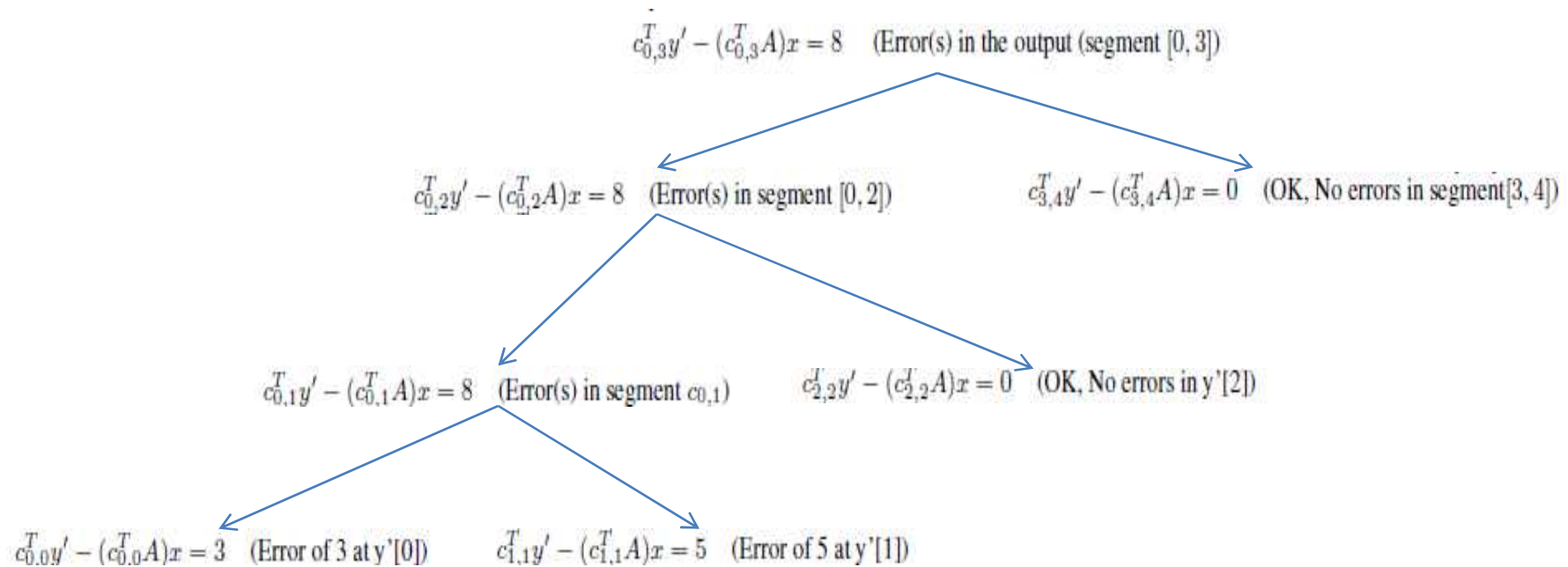


# Example: Error Localization

$$A = \begin{pmatrix} 3 & 0 & 2 & 3 & 4 \\ 2 & 1 & 0 & 2 & 5 \\ 0 & 3 & 2 & 1 & 6 \\ 1 & 0 & 3 & 2 & 2 \\ 3 & 1 & 0 & 0 & 2 \end{pmatrix}, x = \begin{pmatrix} 5 \\ 5 \\ 7 \\ 1 \\ 2 \end{pmatrix} \quad y = Ax = \begin{pmatrix} 40 \\ 27 \\ 42 \\ 32 \\ 24 \end{pmatrix} \quad y' = y + e, e = \begin{pmatrix} 3 \\ 5 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

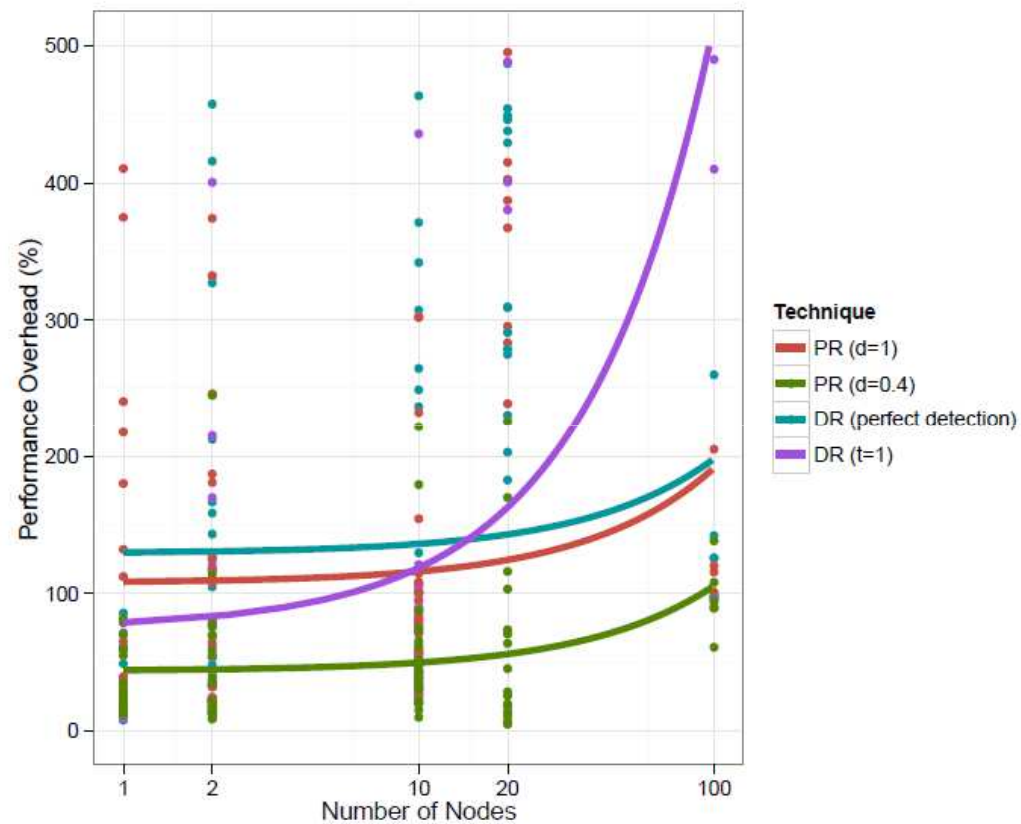
$c_{i,j} = \{ \text{vector with 1's between indices } i \text{ and } j \}$

$c_{i,i} = \{ \text{vector with exactly one 1 at index } i \}$



# Scaling

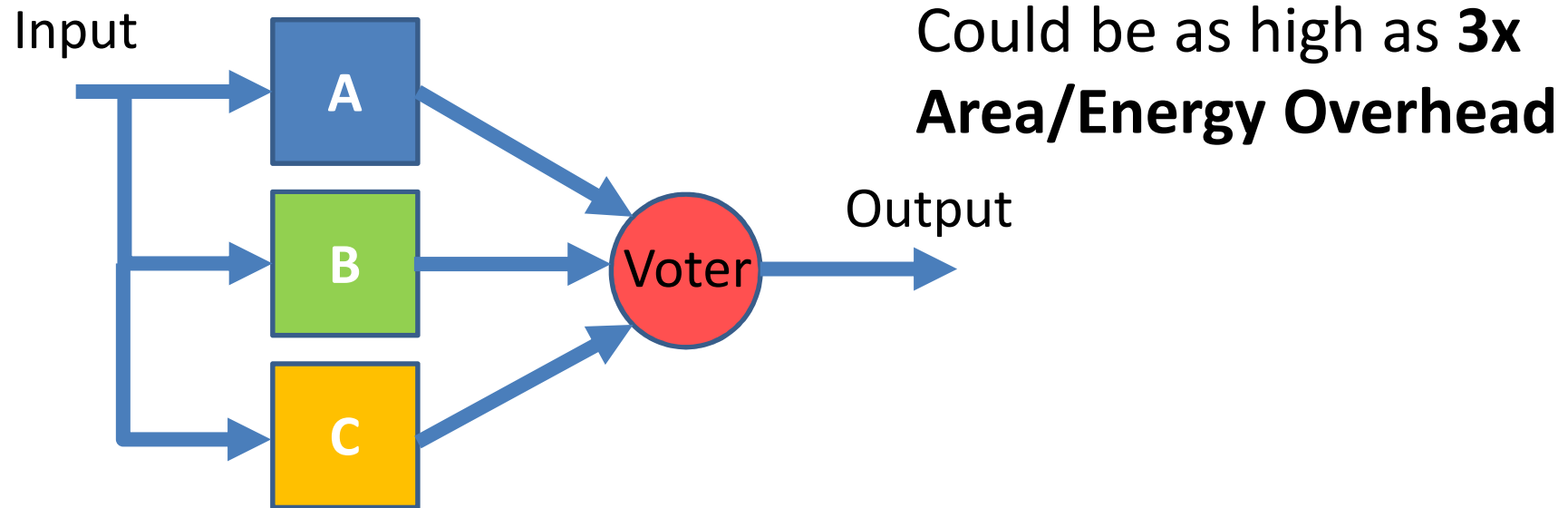
---



**As the number of nodes increases the benefits from partial recomputation only increase.**

# Limitation

---



- Why am I talking about this now?
  1. Power
  2. Power
  3. Power



- 
- Opportunities lie in compilers and architectural approaches to exploit application-level error tolerance



# Summary

---

- Traditional HW/SW contract expects perfect HW
- HW is increasingly non-deterministic

Guaranteeing correctness is expensive,  
especially when correctness is not required

- Opportunistically exposing non-determinism affords significant energy benefits
  - Novel physical design methodologies [HPCA'10, ASPDAC'10, DAC'10, CASES'11, TCAD'11]
  - Microarchitectural optimizations [ICCD'10, DATE'10, TECS'11, CASES'11]
  - Compiler optimizations [DAC'12A, DAC'12B]
- An early prototype confirms significant potential [DAC'12B]
- Future work will explore truly stochastic computing [DAC'12B] , energy-efficient exascale systems [DATE'09, HPCA'12, DAC'12] , predictably timed systems, and multi-scalar systems [TVLSI'12]

# Architectures for Many-core Resilience

---

- Dynamic Constitution and In-network Error Tolerance
- Fluid NMR

# Future Work

---

- Application Robustification
  - Investigating other minimization strategies
  - Sensitivity analysis of parameters (i.e. penalty, step size, conditioning)
  - Evaluate other benchmark transformations
- Low overhead fault detection for Sparse linear algebra
  - Modular Resilience
    - Understand how the approximate nature of checks have impact In the context of other applications.
- Algorithmic Partial Recomputation and Error Localization
  - Understand generality in context of different classes of applications

# Architectures and Compilers for Exploiting Application-level Error Tolerance

---

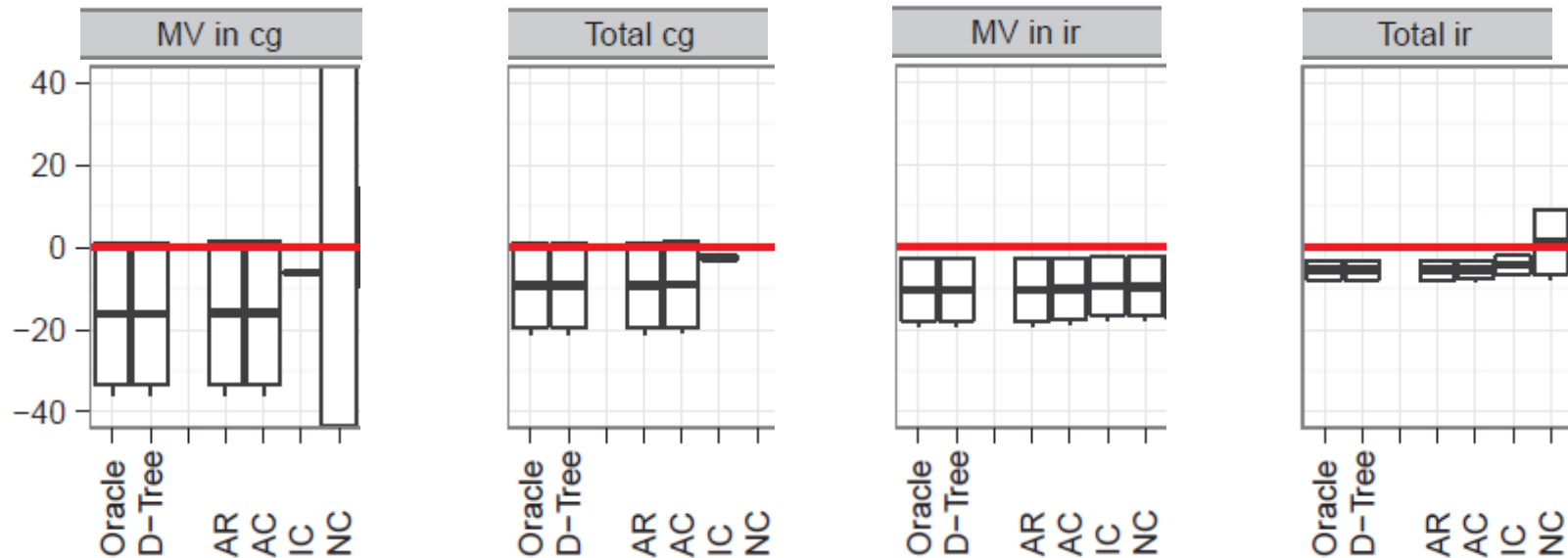
# QR-based Algorithm

---

- Inputs:  $A, b$ ; Output:  $x$ 
  1.  $[Q, R] = \text{qr}(A)$     *// Compute QR factorization (Q orthonormal, R upper triangular)*  
    *//      $A=QR$*   
        *//      $Q^T A x = Q^T b$*   
        *//      $Q^T Q = I$*   
        *//      $R x = Q^T b$* 
    1.  $z = Q^T b$
    2.  $x = \text{backsubstitution}(R, z)$

# Linear Solver Results

---



The sparse techniques in the context of a full system implementation, allow CG to complete 10-20% faster compared to the traditional dense check.

Matrix reuse amortizes the setup costs for conditioning and clustering techniques.

# Householder Factorization (QR)

---

```
function [U,R] = householder(A)

[m, n] = size(A);
R = A;

for k = 1:n,
    x = R(k:m,k);
    e = zeros(length(x),1); e(1) = 1;
    u = sign(x(1))*norm(x)*e + x;
    u = u./norm(u);

    R(k:m, k:n) = R(k:m, k:n) - 2*u*u'*R(k:m, k:n);
    U(k:m,k) = u;
end
```

# Back substitution

---

- `x=backsubstitute(U,b)`

```
n = length( b );
```

```
x = zeros( n, 1 );
```

```
for i=n:-1:1
```

```
    x(i) = ( b(i) - U(i, :)*x )/U(i, i);
```

```
end
```

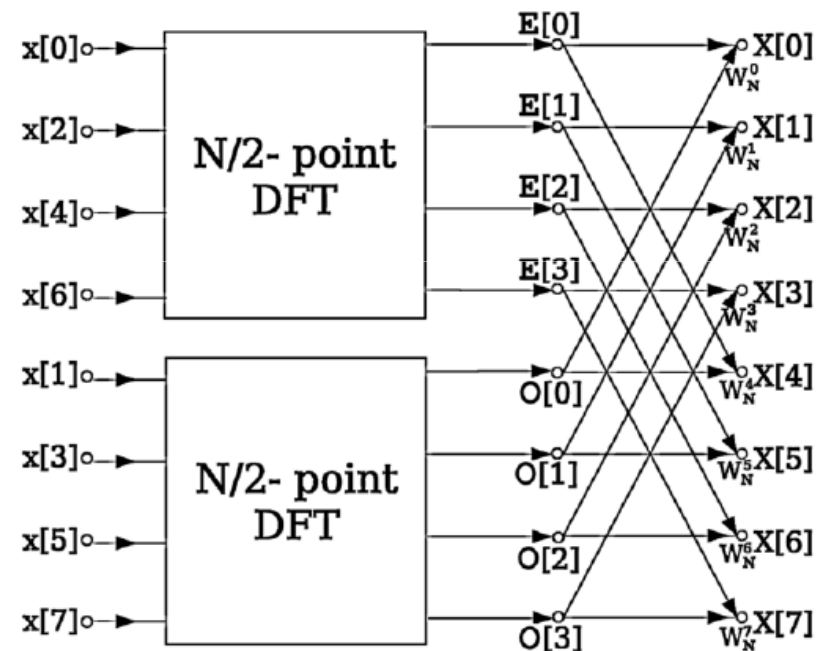


# FFT

- Staged computation
- Recursively apply energy check (Parseval's thm.):

$$\sum_{n=0}^{N-1} x[n]^2 = \frac{1}{N} \sum_{k=0}^{N-1} X[k]^2$$

- Recompute sub-DFT if error detected



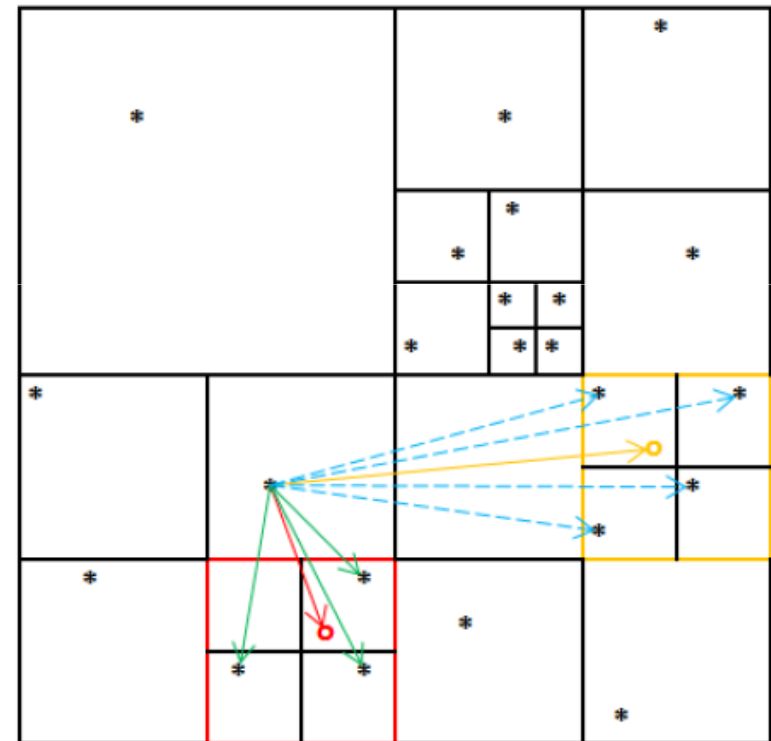
# Barnes-Hut

---

- Detect faults by conservation of energy on subset of bodies:

$$E(t) = V(t) + T(t)$$

- For faulty subset:
  - Re-build Hierarchy
  - recompute forces
- Foreach body: potentially update position and velocity



# Methodology

---

- Fault Models
  - Symmetric
    - Distribution w/ single and two Gaussian modes
  - Memory
    - An exponential distribution representing bit-flip model
  - Non-symmetric
    - Distribution w/ Gaussian centered at large positive ( $1e5$  or  $1e10$ ) representing unsigned representation faults.
- Benchmarks
  - University of Florida Sparse Matrix Collection
  - Linear Solvers (CG and Richardson iteration)

## Methodology (2)

- Detection Accuracy for MVM measured by F1-Score
  - TP= True Positives, FP= False Positives, FN= False Negatives

$$F_1 = \frac{2TP}{2TP + FP + FN}$$

- 20 Millions runs for each configuration

MVM and Solver Parameters	Values
Techniques	Dense,AR,AC,NC,IC, ICAR,ICAC,NCAR,NCAC
Fault rates	0, 1e-6, 1e-5, 1e-4,1e-3,1e-2,1e-1
LSQ tolerance (IC)	1e-10,1e-6,1e-3,1e-1,1, 1e1,1e3,1e6,1e10
LSQ input condition num. (IC)	1e15
Eigen solver tolerance (NC)	1e-10,1e-6, 1e-1 1
Sample rate (AR,AC)	0.001, 0.01, 0.05,0.1,0.2,0.3, ... 1.0
Other Solver Parameters	Values
Linear Solver	CG, IR, CG-pre, IR-pre
Detection Threshold Factor( $\tau_0$ )	1e-5, .1,1,10,1e3,1e5,1e7

TABLE I  
MVM AND LINEAR SOLVER PARAMETERS

## Example 2: Robustified GM (contd)

---

- Input Matrix:  $W = \begin{bmatrix} 5 & 2 & 8 \\ 1 & 6 & 2 \\ 9 & 3 & 7 \end{bmatrix}$
- $-\langle W, X \rangle$  for one example of a matched graph

$$\left\langle \begin{bmatrix} 5 & 2 & 8 \\ 1 & 6 & 2 \\ 9 & 3 & 7 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \right\rangle = -18$$

- $-\langle W, X \rangle$  for the correctly matched graph:

$$\left\langle \begin{bmatrix} 5 & 2 & 8 \\ 1 & 6 & 2 \\ 9 & 3 & 7 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \right\rangle = -23$$

**(lower than  
incorrectly matched  
Graph)**

# Other Work

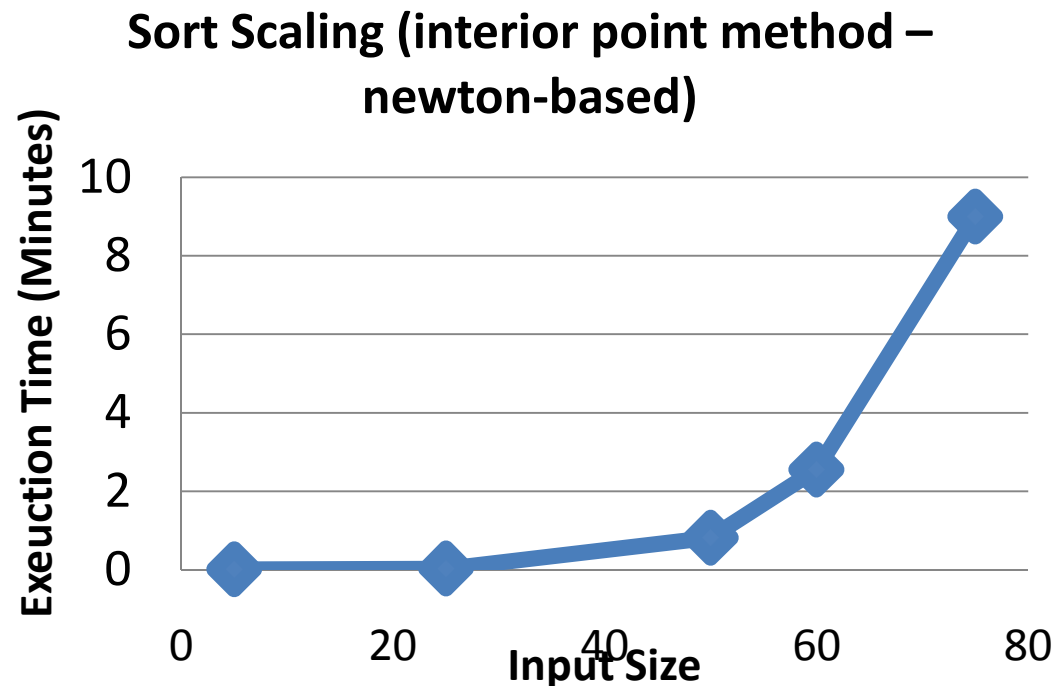
---

- Past
  - *Hardware/System Support for Four Economic Models for Manycore Computing* [UIUC-CRHC-TR 2007]
  - *Towards Scalable Reliability Frameworks for Error Prone CMPs* [CASES 2009]
- Future
  - Viewing all computation as Statistical Inference
  - System-level Optimizations for Exploiting Application Error Tolerance
  - Performance/Robustness Tradeoffs of ODE Solvers in face of Error-prone Hardware

# Limitations

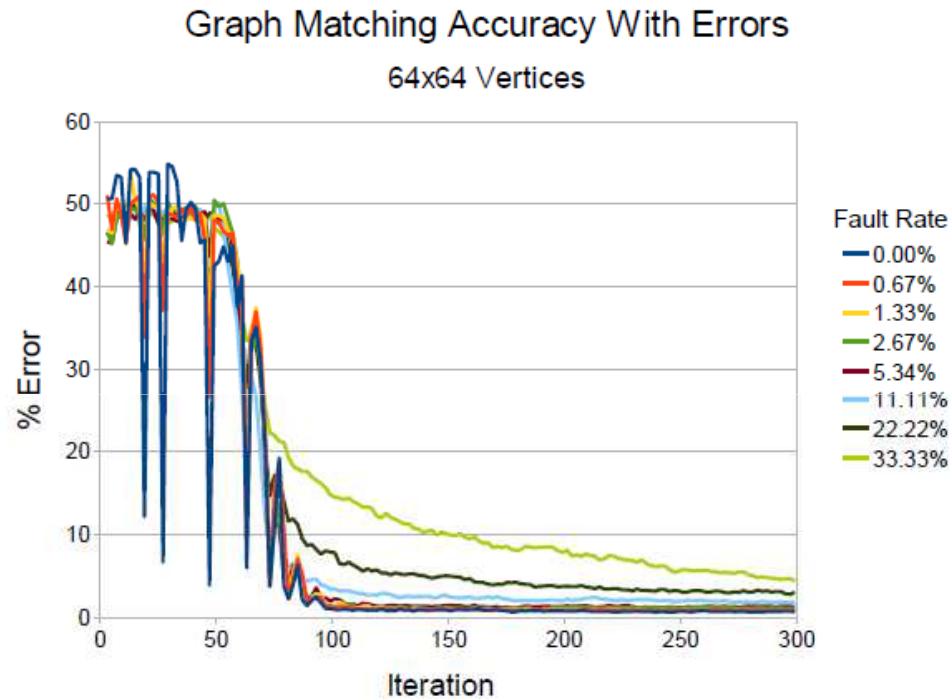
---

- Gradient Descent (1<sup>st</sup> order)
  - learning rates can be difficult to select and slow
    - E.g. choosing sequence of penalty parameters
- Newton-based approaches (2<sup>nd</sup> order)
  - Expensive per iteration cost for Hessian calculation



# Graph Matching (64x64) using Gradient Descent (300 Iterations)

---





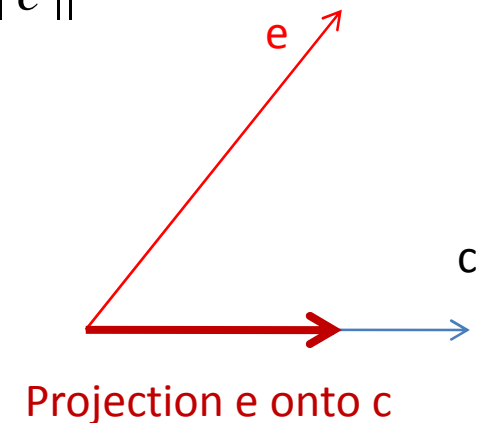
# Approximate Correction

---

- Faulty MV product output ( $v'$ ):  $v' = Au + e$
- Traditional ABFT corrects up to  $\left\lfloor \frac{k}{2} \right\rfloor$  faults where  $k$  is number of check vectors.
- Applications may only be concerned reducing error (RMS Accuracy:  $\|v' - v\|^2$ )
- 
- Subtract projection of error on code space:  $v'' = v' - \frac{(c^T e)c}{\|c\|^2}$
- Guaranteed to improve accuracy

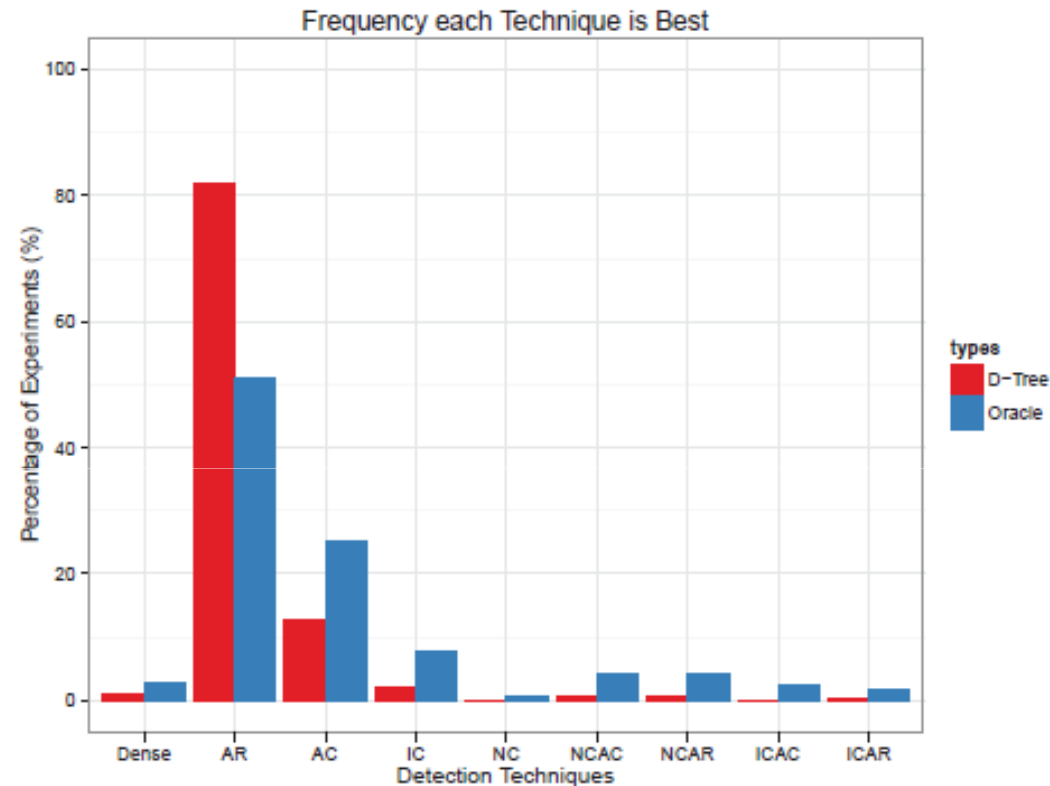
$$\|v'' - v\|^2 = \|v' - v\|^2 - \frac{(c^T e)^2}{\|c\|^2}$$

$$\|v'' - v\|^2 \leq \|v' - v\|^2$$



# Decision Tree Based Results

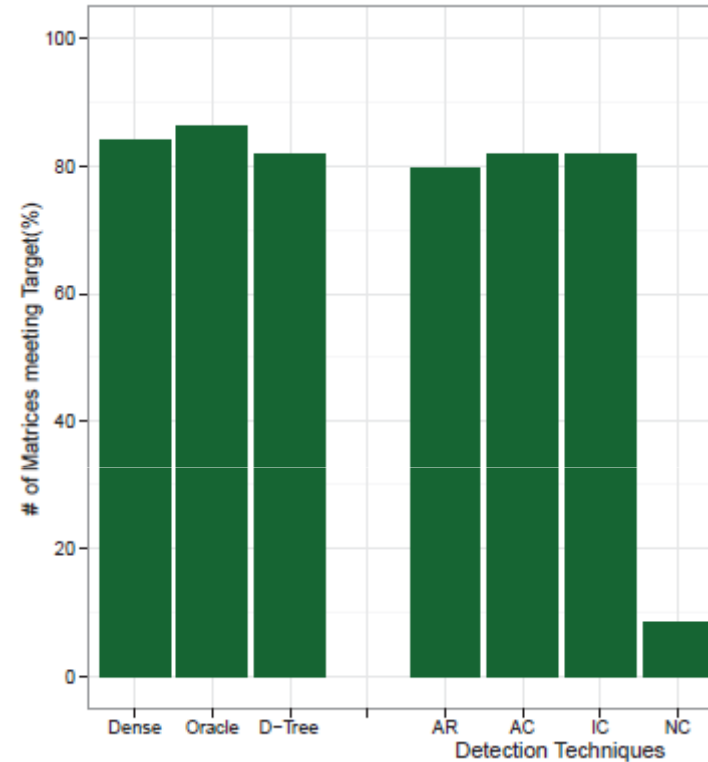
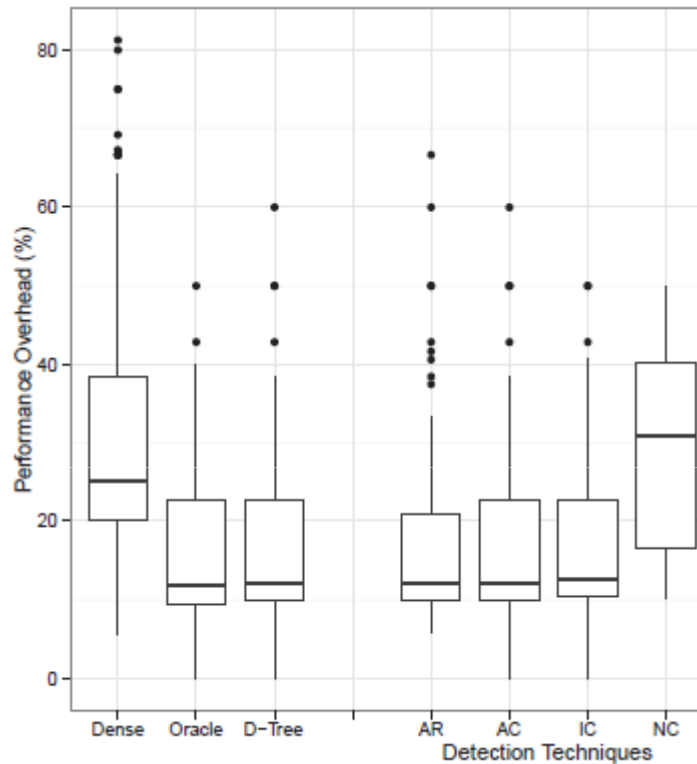
---



Approximate Random (sampling) was frequently chosen by the decision tree. The D-Tree configurations were comparable the oracle configurations >90% of time.

# MVM Results

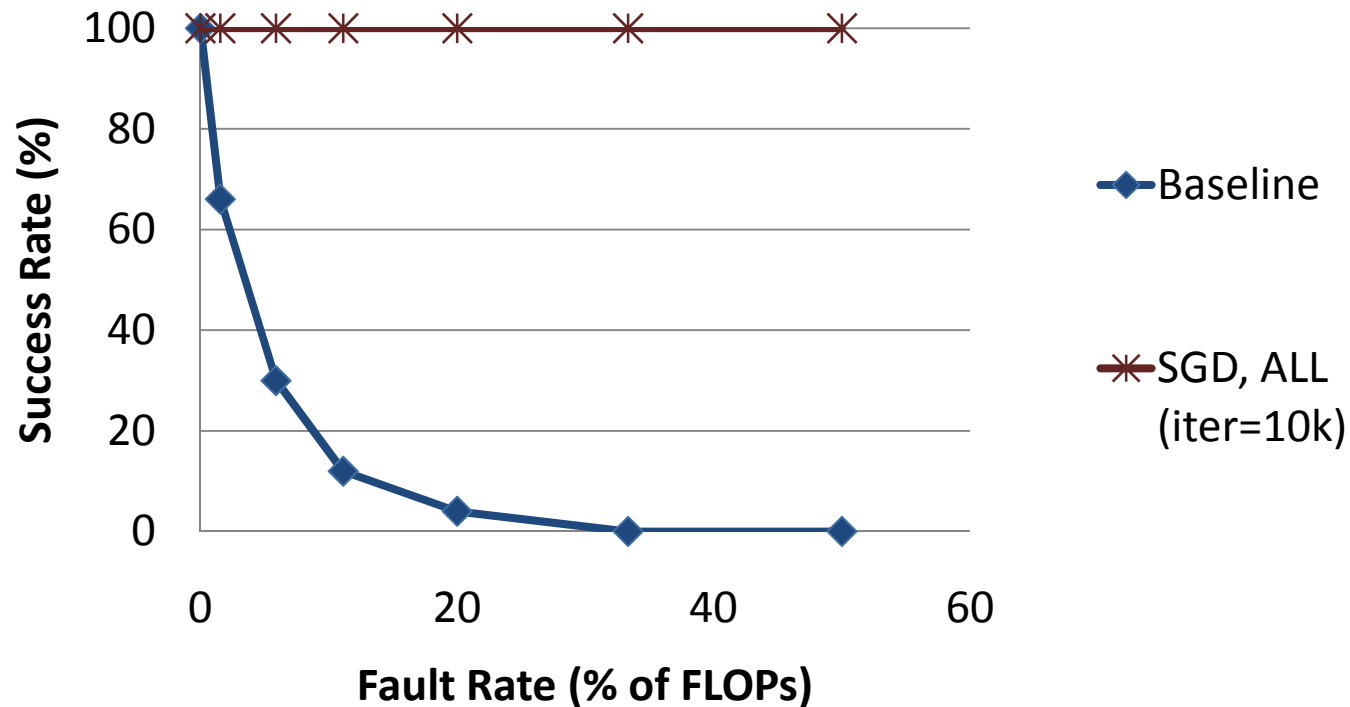
---



50-60% Performance reductions are typical for large and sparse problems for the same detection accuracy as traditional dense checks.

# Graph Matching(5x6) using Gradient Descent (10k Iterations)

---



100% Accuracy with Graph Matching using SGD even in face of large error rates.

# Future work

---

- Investigating other minimization strategies
- Sensitivity analysis - to parameter, compiler etc.  
exact parameter want to explore
- Other benchmarks
- Comparison NMR – something to say about
- Other Solvers for numerical Optimization formulations
- NMR approaches
  
- Detection Limitations
  - Dsn limitations
  - Modular resilience – note that our detection work –
  - In context of other applications approximateness of detection

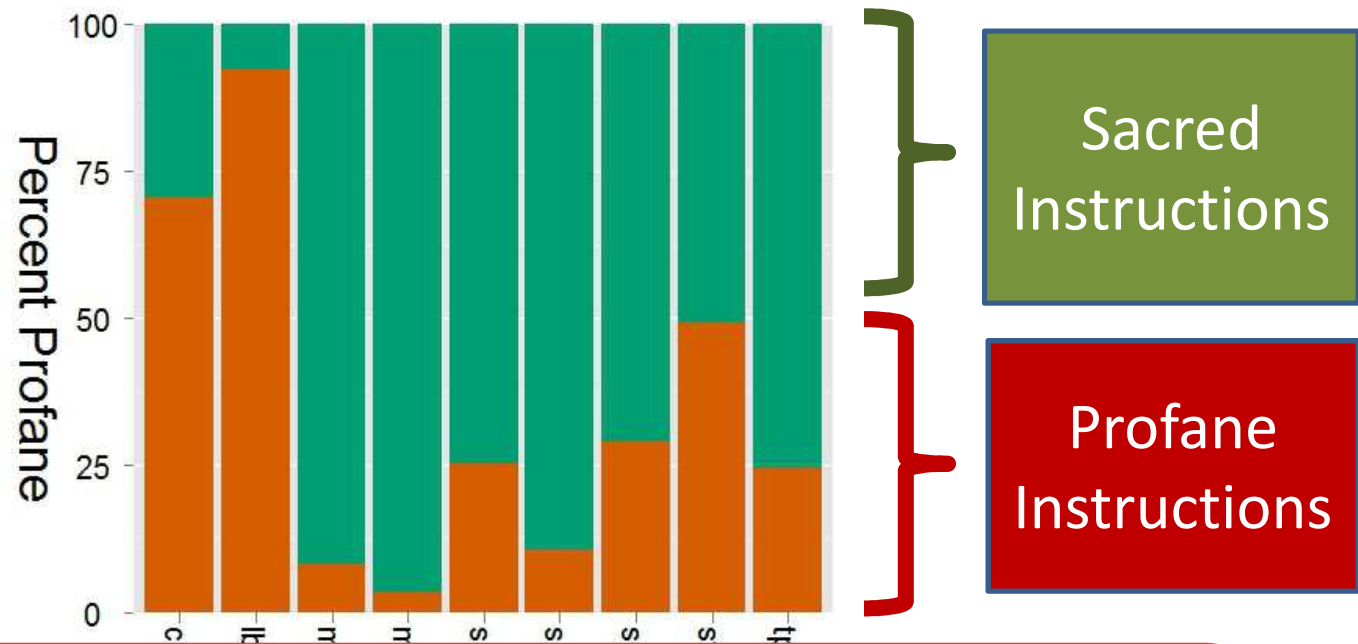
# Additional Work

---

- Statistical Inference
- Partial Recomputation and Error Localization
- Modular Reliability
- PDE Reliability

# Impact of Errors on Software

---



Conclusion:

Points to categorization of instructions

Sacred instructions = error **intolerant**

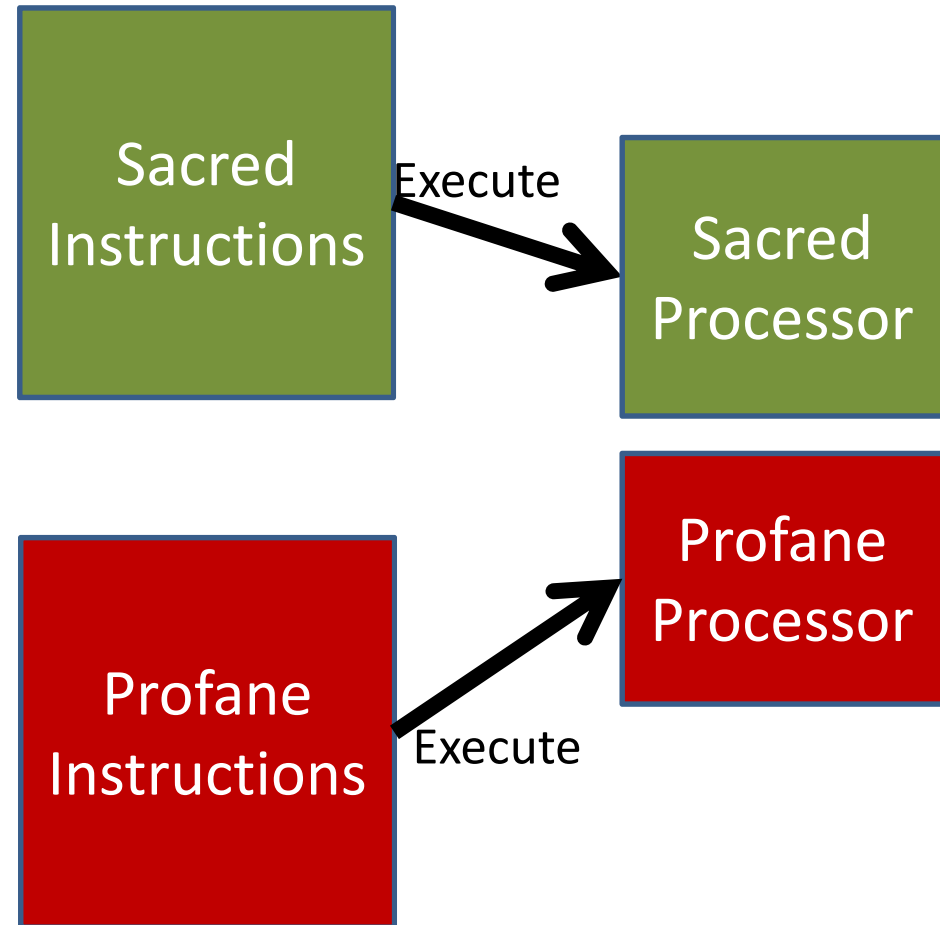
Profane instructions = error **tolerant**

# First Step

---

## Key Idea:

- Run **profane** instructions in low power/unreliability mode
- Run **sacred** instructions in high power/reliability mode





# Instruction Interleaving

```
movss (% r s i ) ,%xmm0
l e a (%r15,%r9 , 1 ) ,%r a x
a d d s s (%r c x ) ,%xmm0
movaps %xmm3,%xmm1
add $0x1,%r10d
add %r d i ,% r s i
add %r d i ,%r9
mul s s (%rax,%r8 , 4 ) ,%xmm1
add %r d i ,%r c x
a d d s s (%rdx ) ,%xmm0
add %r d i ,%rdx
24 a d d s s (%rax,%r12 , 4 ) ,%xmm0
26 a d d s s (%rax,%rbp , 4 ) ,%xmm0
28 a d d s s (%rax,%rbx , 4 ) ,%xmm0
30 mul s s %xmm2,%xmm0
32 s u b s s %xmm1,%xmm0
34 movss %xmm0,(% r11 )
add %r d i ,%r11
cmp %r10d,%r14d
j g .L1
movss (% r s i ) ,%xmm0
l e a (%r15,%r9 , 1 ) ,%r a x
a d d s s (%r c x ) ,%xmm0
movaps %xmm3,%xmm1
add $0x1,%r10d
add %r d i ,% r s i
add %r d i ,%r9
mul s s (%rax,%r8 , 4 ) ,%xmm1
add %r d i ,%r c x
a d d s s (%rdx ) ,%xmm0
add %r d i ,%rdx
24 a d d s s (%rax,%r12 , 4 ) ,%xmm0
26 a d d s s (%rax,%rbp , 4 ) ,%xmm0
28 a d d s s (%rax,%rbx , 4 ) ,%xmm0
add %r d i ,%r11
cmp %r10d,%r14d
j g .L1
```

Fine grained interleaving of  
profane/sacred instructions  
(7-10 instructions on average)

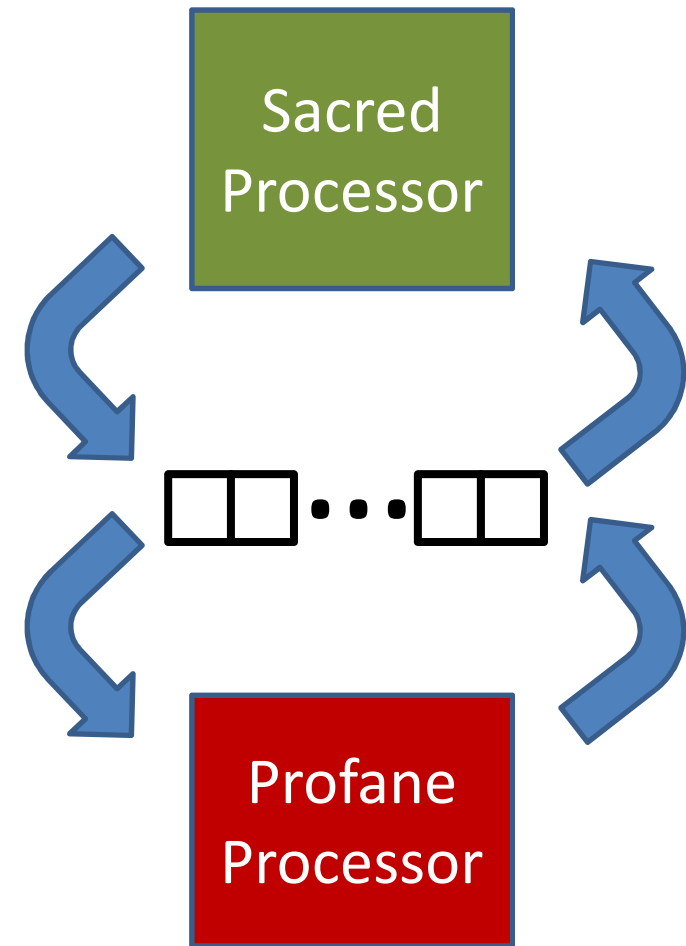
**Mode switches can take  
100-10000 cyles!**

**May kill any potential  
benefit**

# Instruction Interleaving

---

- Decoupling fine grained interleaving /dependencies not easy
- Use queue to communicate between modes



# Proposed Execution Model

add	\$c, \$a, \$b
mul	\$d, \$c, \$a
mul	\$e, \$c, \$b
add	\$d, \$d, \$e

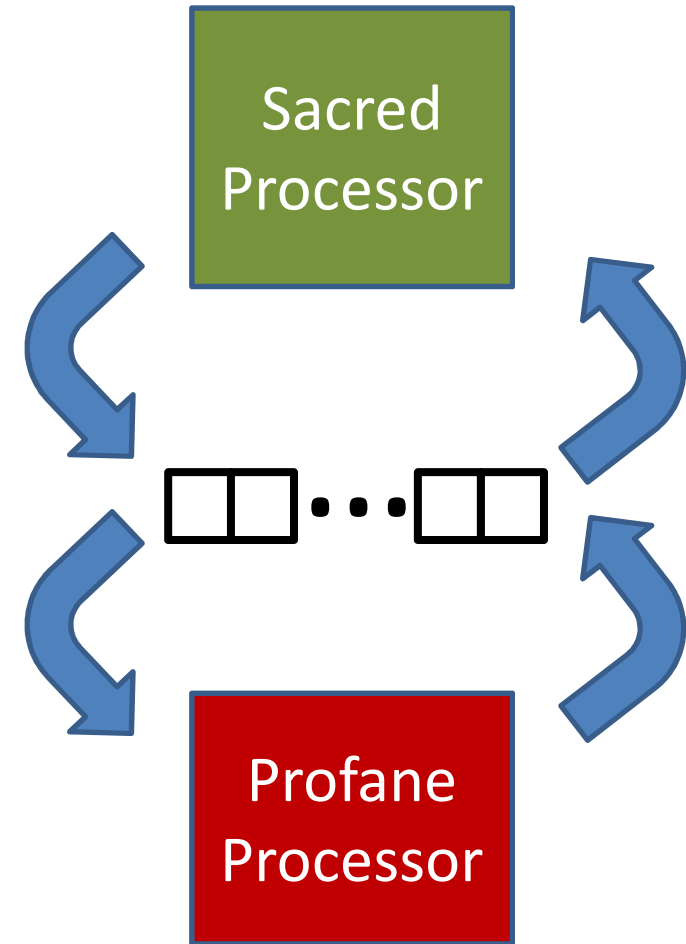
**Typical  
Communication:  
Sacred-> profane**

.  
. .  
. .

add	\$c, \$a, \$b
mul	\$e, \$c, \$b
push \$(PBlockAddr)	
push \$c	
push \$e	

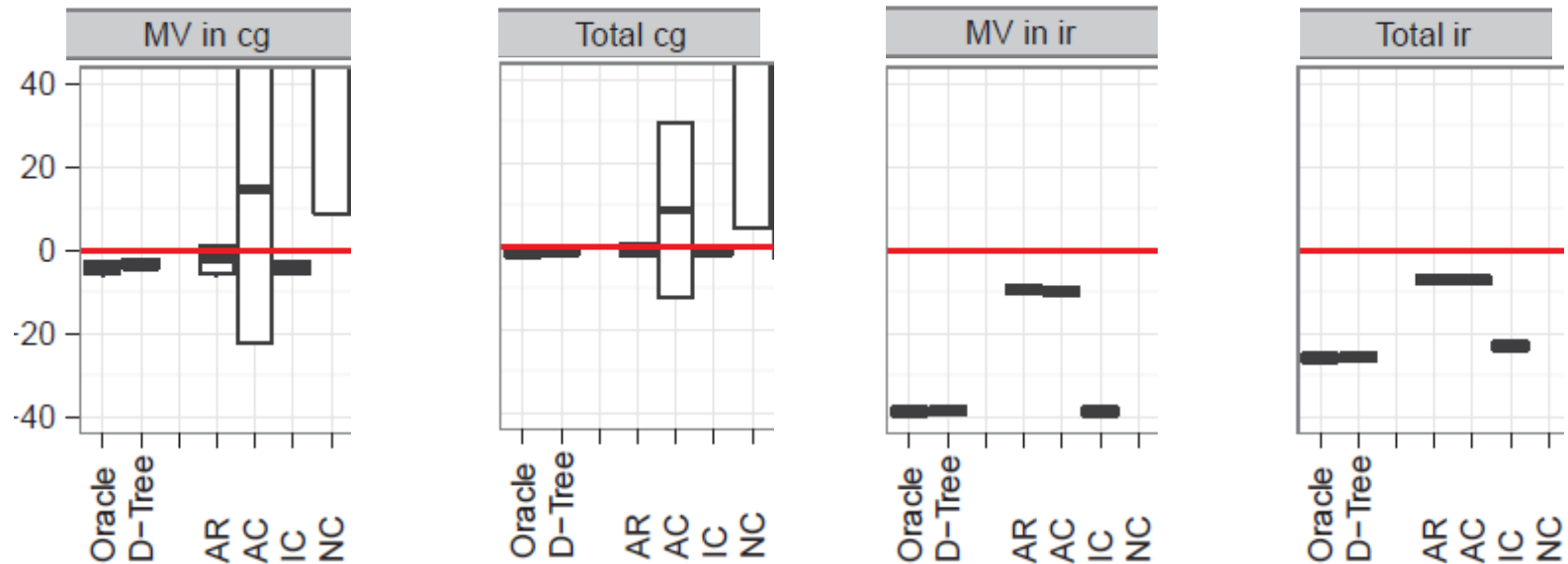
pop (update PC)  
pop \$c

mul	\$d, \$c, \$a
add	\$d, \$d, \$e



# Preconditioned Linear Solvers

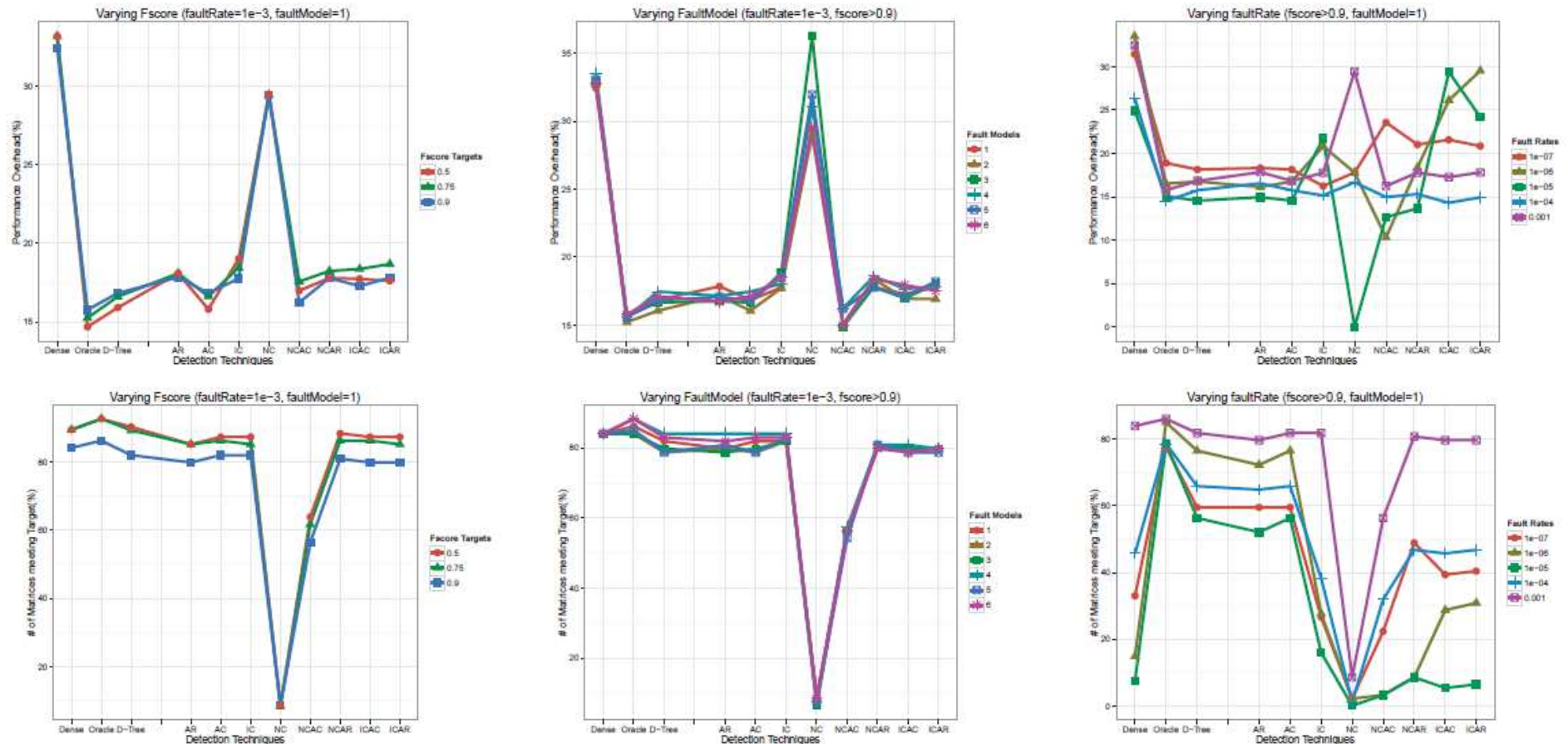
---



Preconditioned-CG showed less improvement due to less tolerance of intermediate errors on the solver.

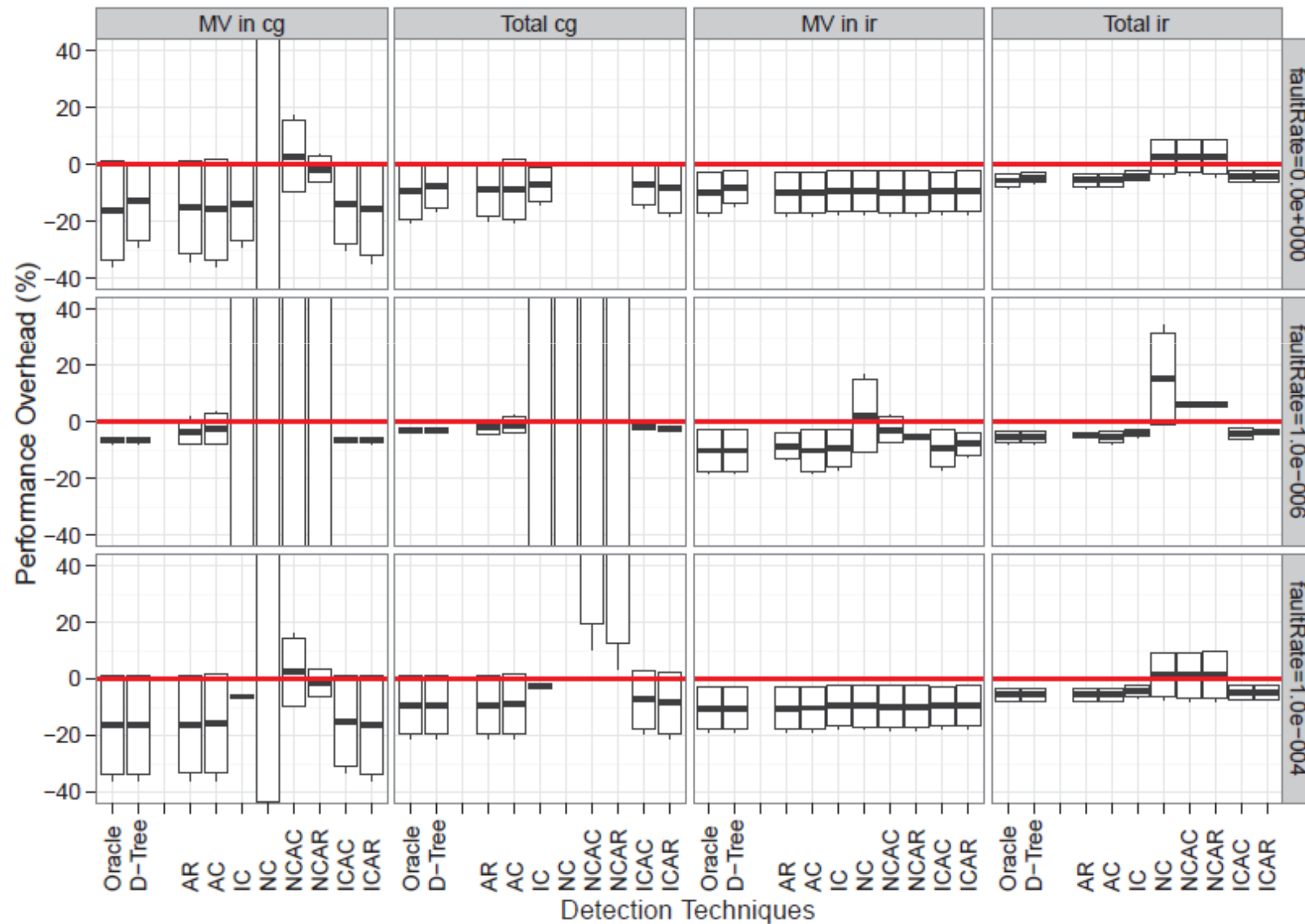
IR (Richardson Iteration) showed improvements up to 40% due to greater error tolerance and slower convergence.

# Fault Model Sensitivity



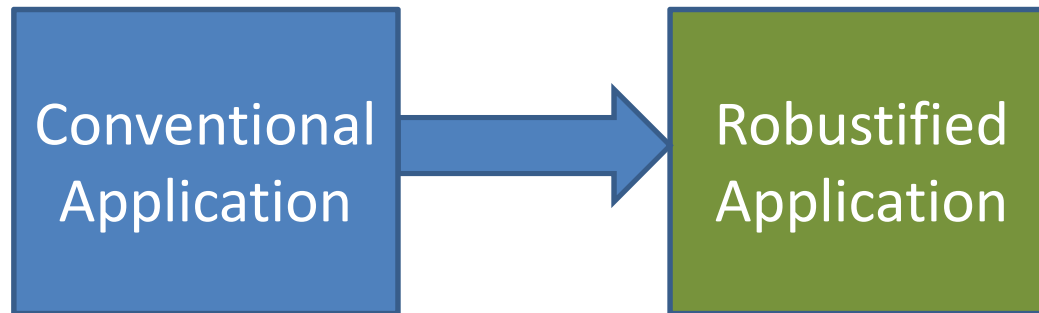
The fault model had little impact on the observed trends across the techniques. Fault rate was a much more significant system parameter.

# Fault Rate sensitivity for Linear Solvers



# Application Robustification

---

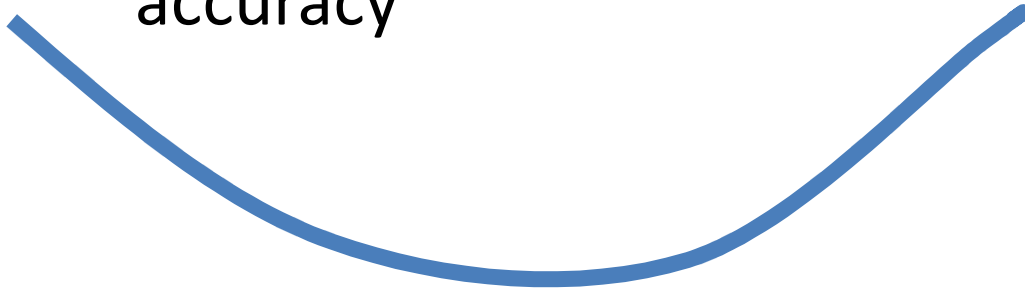


- **Goal**
  - Redesign applications that produce acceptable output in presence of errors
    - Same as output without errors for most applications [*in spite of intermediate stochastic behavior*]
    - Within certain tolerance for other applications
  - Lower costs than simple re-execution based techniques

# GD Solver Variations

---

- Shape of objective function impacts performance and accuracy



Solver 'friendly' objective

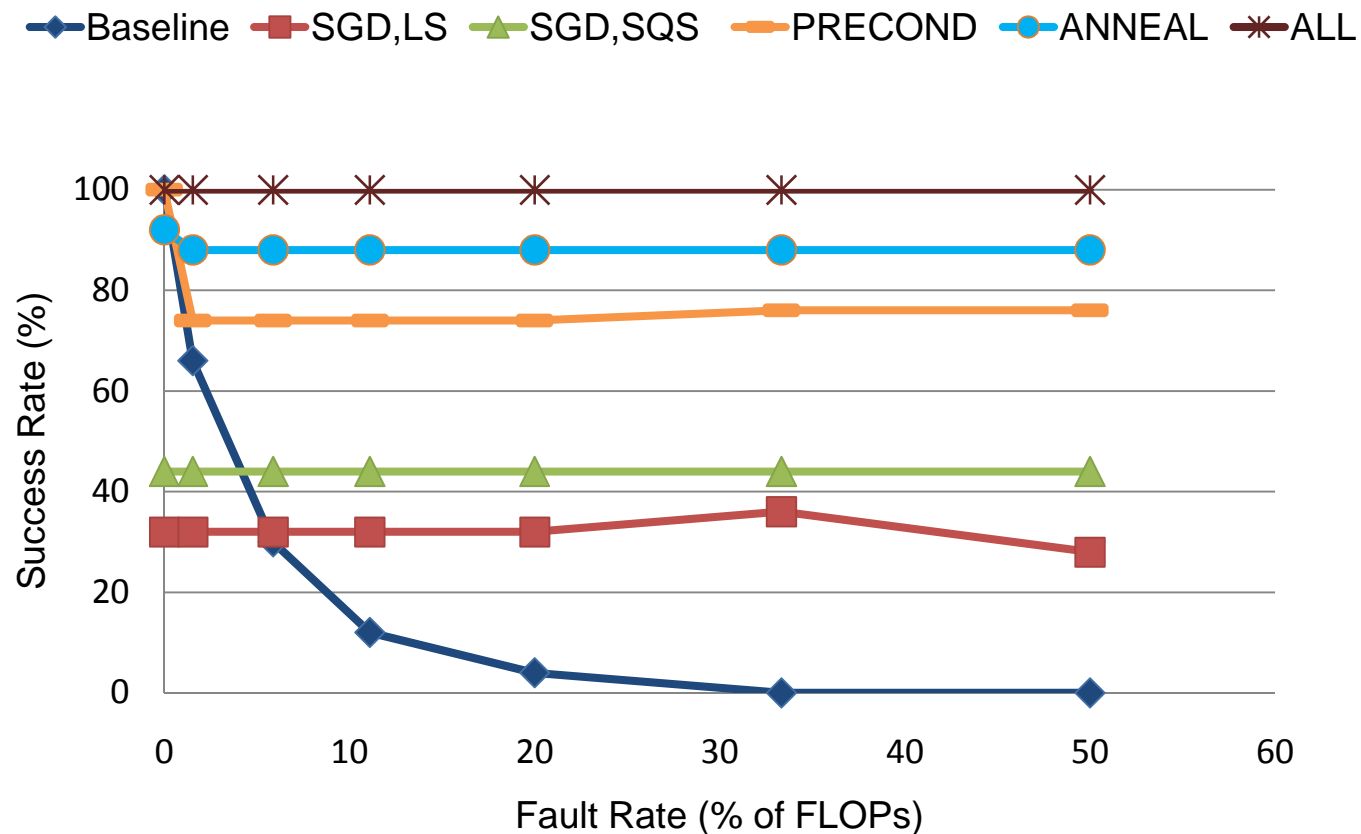


'unfriendly' objective

- Techniques for making objective more friendly
  - Preconditioning
- Techniques for improving performance with unfriendly objectives
  - Projected Gradient/Rounding
  - Fixed Rate/Adaptive Step Sizing
  - Exploit sparsity in input matrices



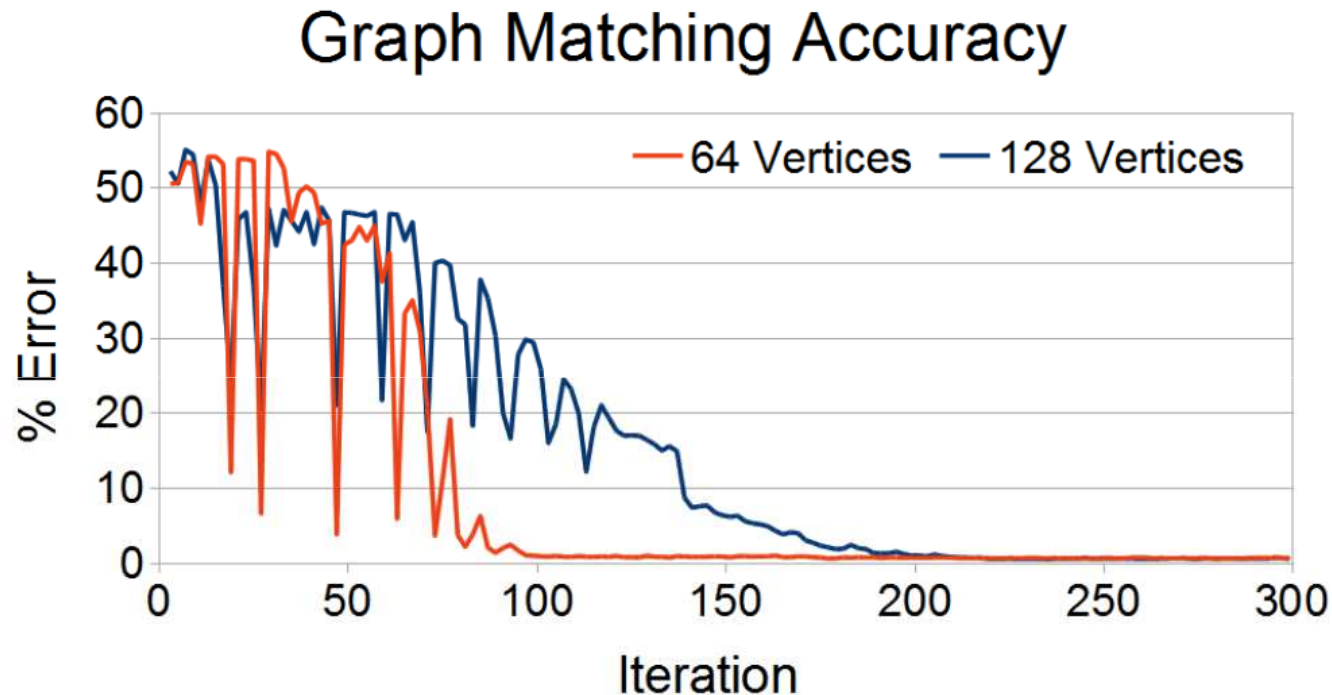
# Graph Matching(5x6) using Gradient Descent (10k Iterations)



100% Accuracy with proper subset of techniques for  
arbitrary inputs

# Convergence of other Optimization Formulations

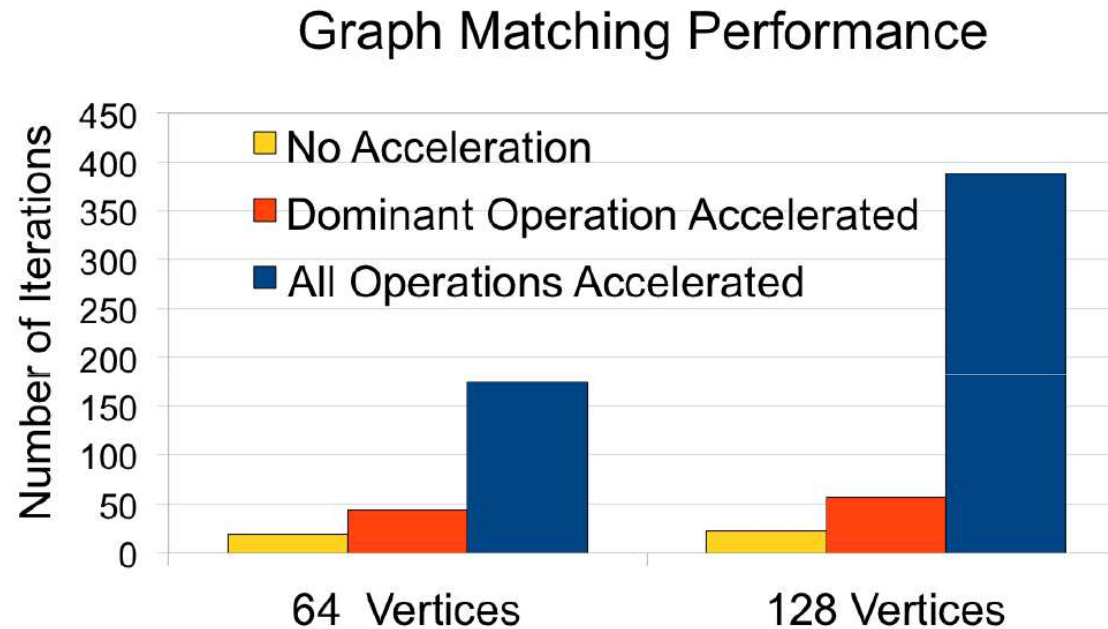
---



On traditional architectures, some low complexity applications (polynomial) can incur large execution overheads compared to the baselines.

# Parallelism Opportunities

---



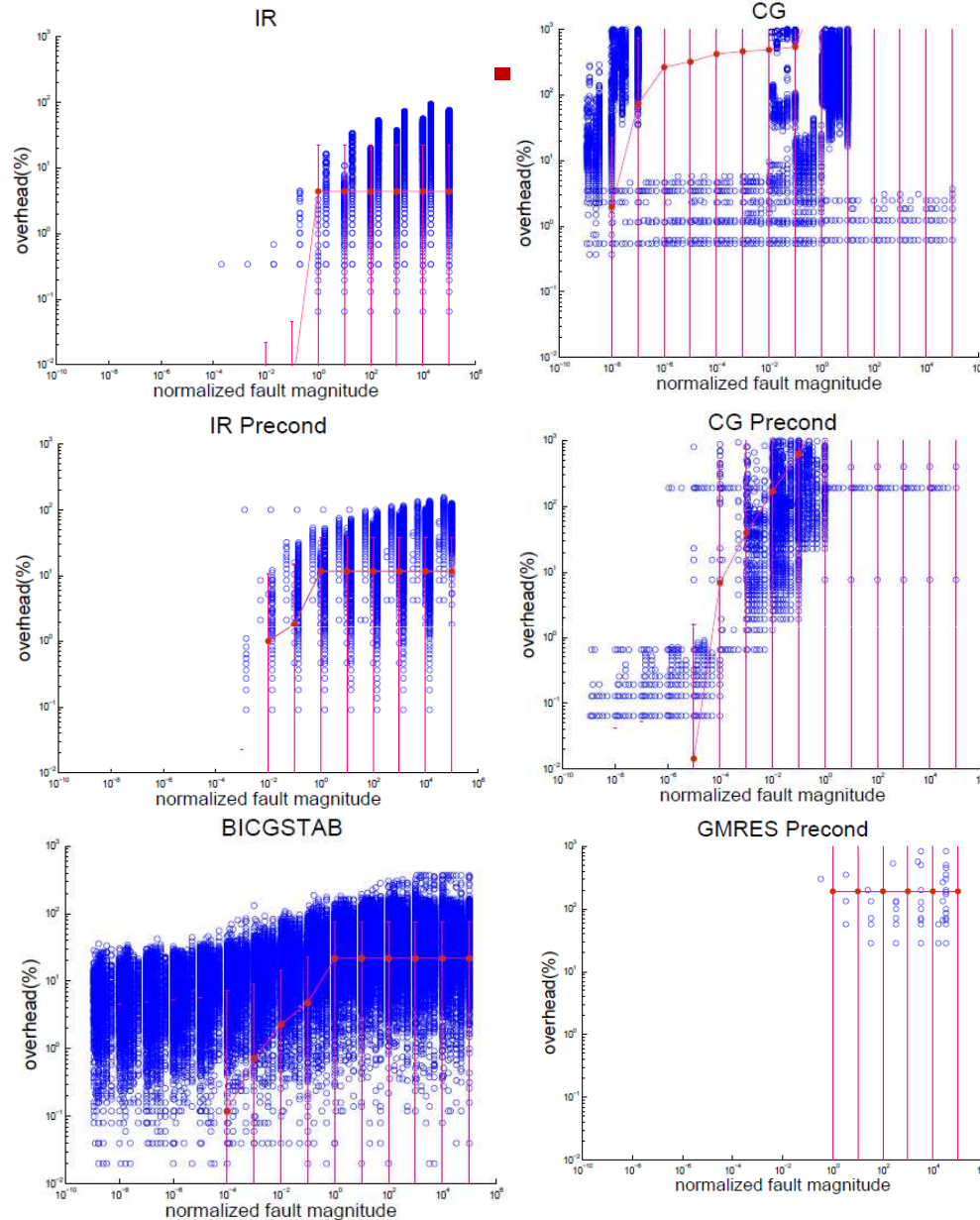
- By formulating applications as numerical optimization problems, the solution is inherently parallelized.

Exploit with parallel architectures and accelerators.  
(Solver-Engine) [Kesler, et al, 2010]

# Linear Solver Fault Injections

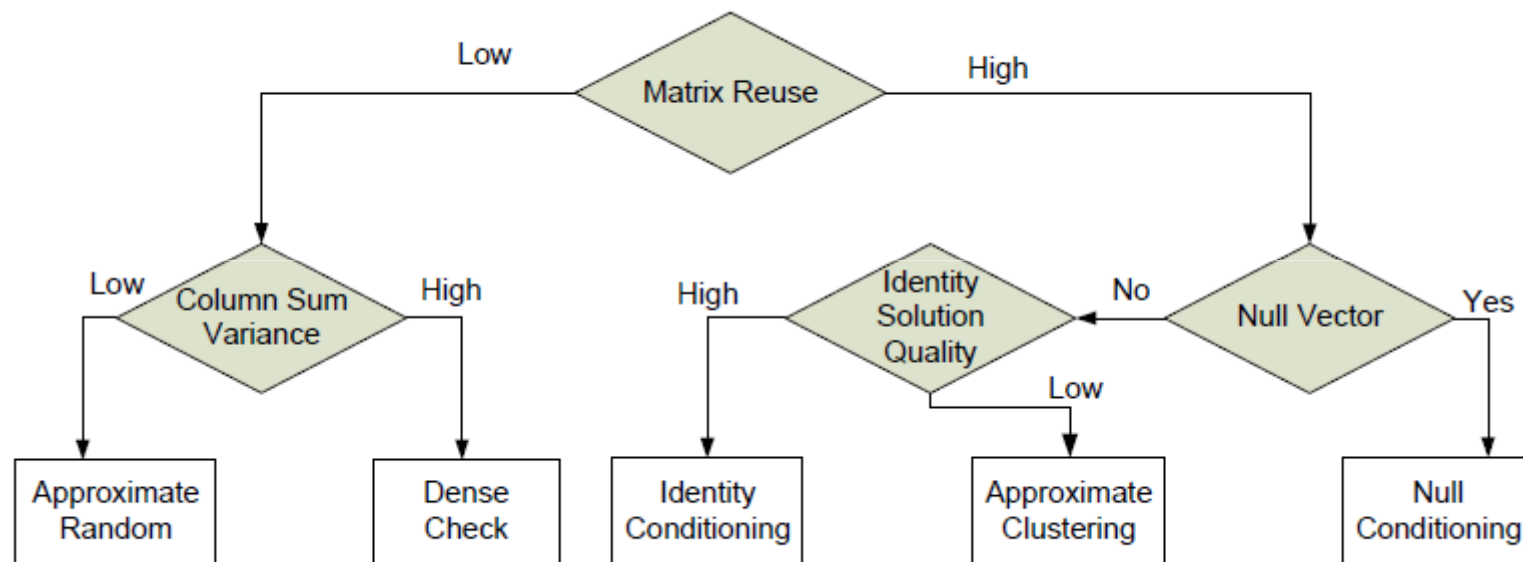
All of the solvers, on average, show inherent error tolerance.

The sparse techniques detect fault magnitudes which tend to result in critical linear solver errors, with high accuracy.



# Algorithmic Decision Tree

---



# Algorithmic Fault Correction

---

- Detection + Rollback recovery techniques really help when errors are rare.
- Under high error rates, techniques for forward error correction are needed.
  - *Application Robustification* is an example of forward error correction.
- Our current work is aimed at **algorithmic fault correction**, by relying on inherent application fault tolerance.
- General problem formulation:
  - given an application with unknown correct output  $y^*$ , ensure that the app, even in the presence of faults, produces an output  $y$  within a certain threshold of  $y^*$ .

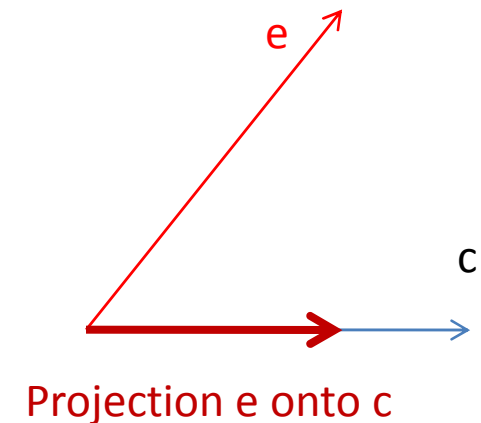
# Algorithmic Fault Correction for Linear Algebra (MV Product)

---

- Faulty MV product output ( $v'$ ):  $v' = Au$
- Traditional ABFT corrects up to  $\left\lfloor \frac{k}{2} \right\rfloor$  faults where  $k$  is number of check vectors.
- Instead, application may only care only about approximately correcting vector error ( $e = v' - v$ ) and improving accuracy. (RMS Accuracy:  $\|v' - v\|^2$ )
- Approx correction by subtracting the projection of error onto the code space (check vector= $c$ ) The partially corrected MV product output ( $v''$ ):

$$v'' = v' - \frac{(c^T e)c}{\|c\|^2}$$

$$v = Au$$



# Algorithmic Fault Correction Benefits

---

- Guaranteed to improve accuracy

$$\|v'' - v\|^2 = \|v' - v\|^2 - \frac{(c^T e)^2}{\|c\|^2}$$
$$\|v'' - v\|^2 \leq \|v' - v\|^2$$

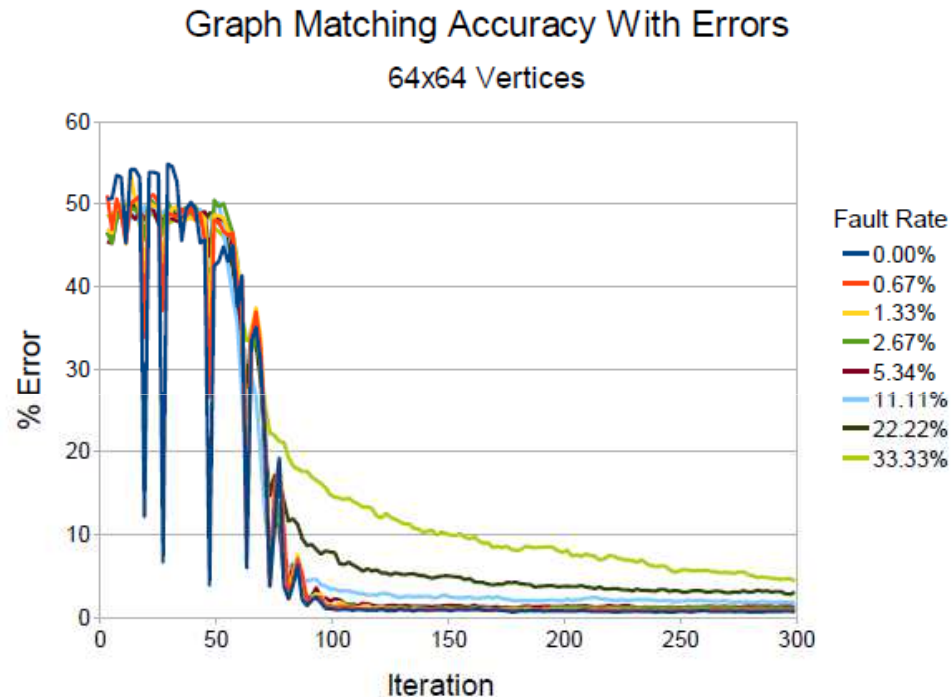
- Include multiple codes in check to meet necessary accuracy targets.
- A given application sees faults manifested in different ways (performance and accuracy).
- Approximate Error Correction efficiently provisions the correction technique to account for the most important faults, in terms of performance and accuracy.

Ongoing work



# Replace prior graph with the following

---

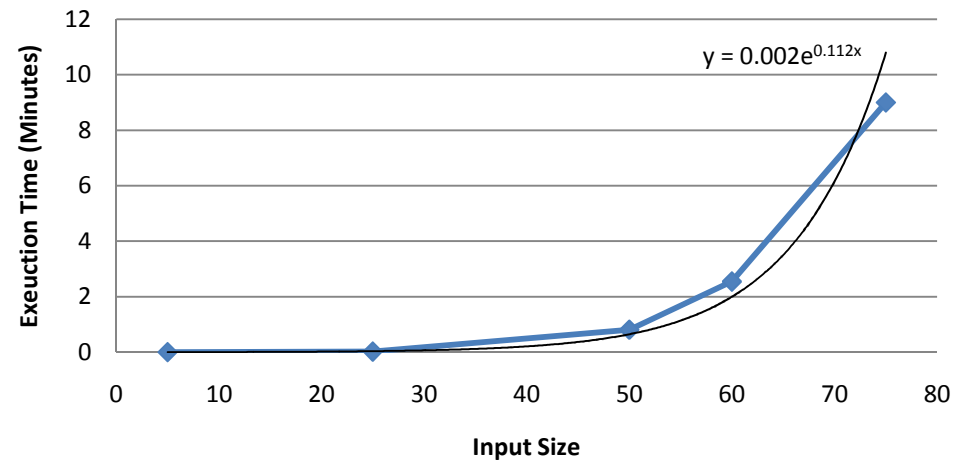


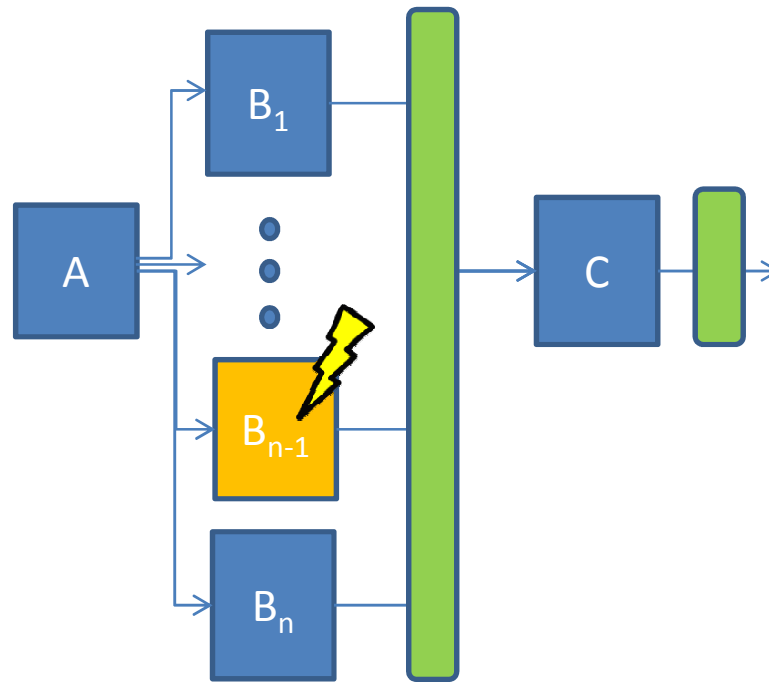
- GM (32 x 32)
  - Fault rate vs **error rate**
    - Interior point method
    - Simplex
    - SGD
    - Baseline
    - Baseline DMR
    - Baseline TMR

# Performance Scaling

---

linear program scaling (interior point  
method - cvxopt)





# Discussion

---

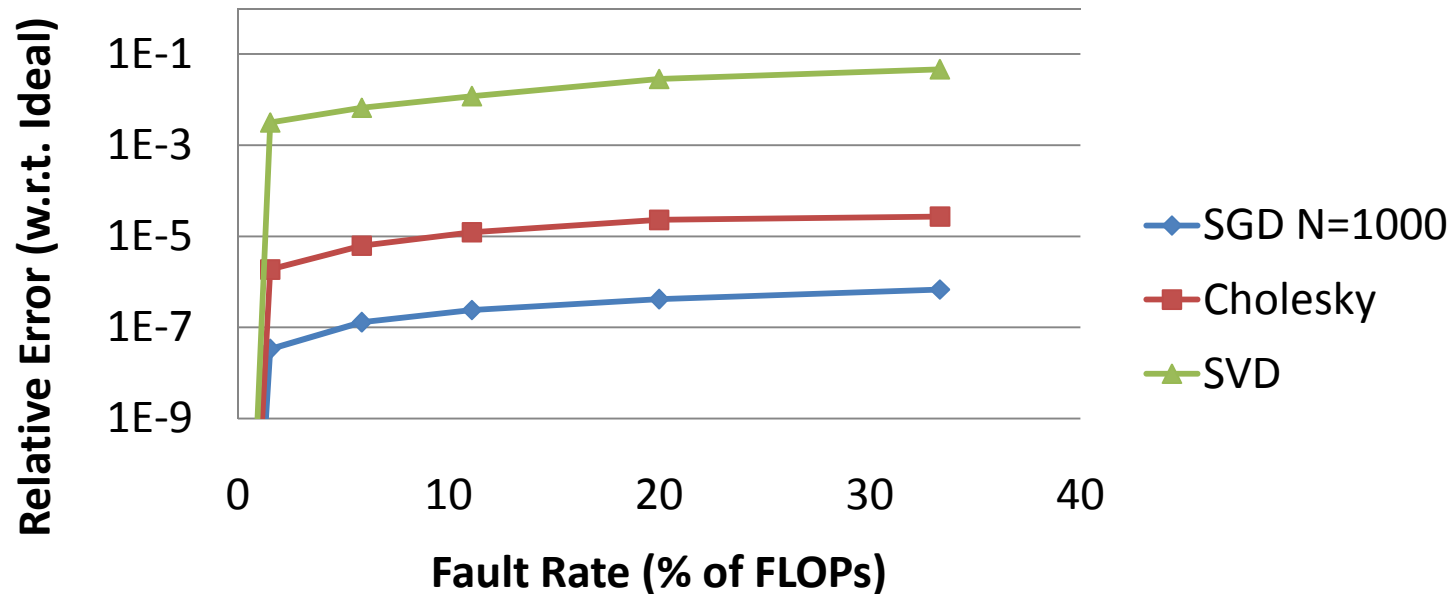
1. An iteration of an optimization-based formulation may have higher complexity than the baseline for some apps (e.g. sort).
  - Robustification still useful when the computational substrate is inherently stochastic.
2. For other applications, the complexity of a single iteration may be lower compared to the baseline (e.g. graph matching).
  - Robustification may be useful for such applications even for voltage over scaling related systems which exploit reliability/power tradeoff.
  - By formulating applications as numerical optimization problems, the problems are more parallelized. (And can be exploited with an accelerator)

# Thanks ! 😊 Questions

---

# Least Squares (100x10) using Gradient Descent

---

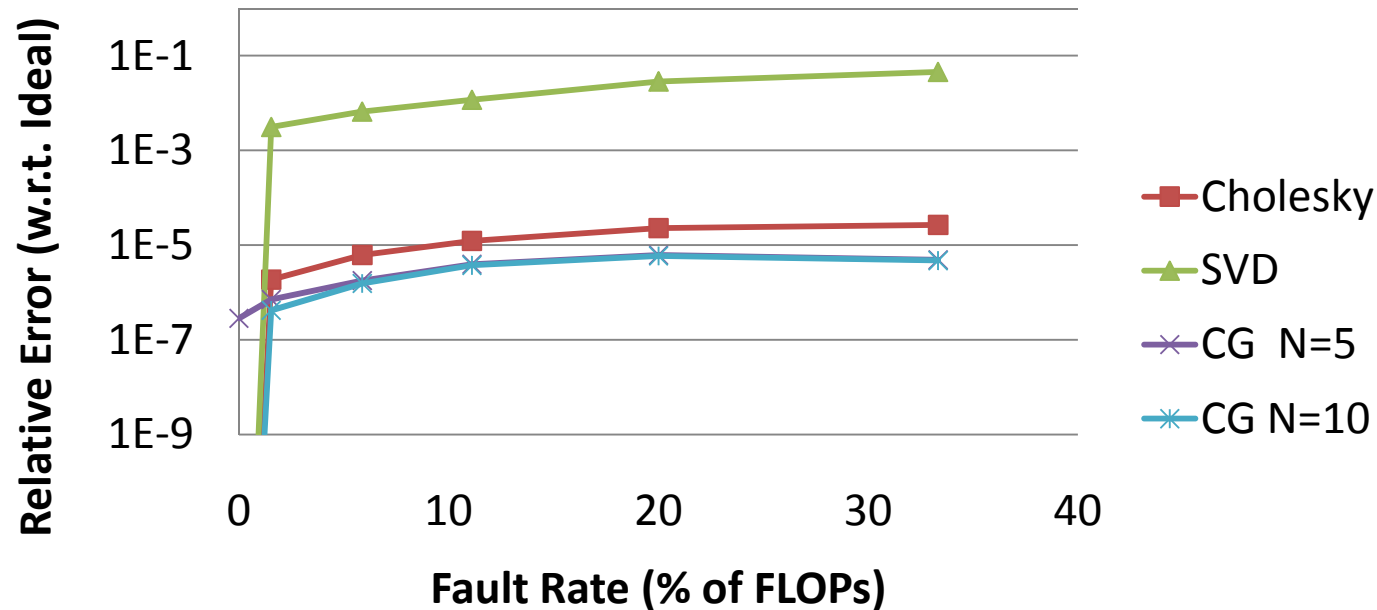


- SVD & QR are not robust to even small fault rates.

LSQ using GD several magnitudes less error than  
baseline with faults

# Least Squares (100x10) using Conjugate Gradient

---

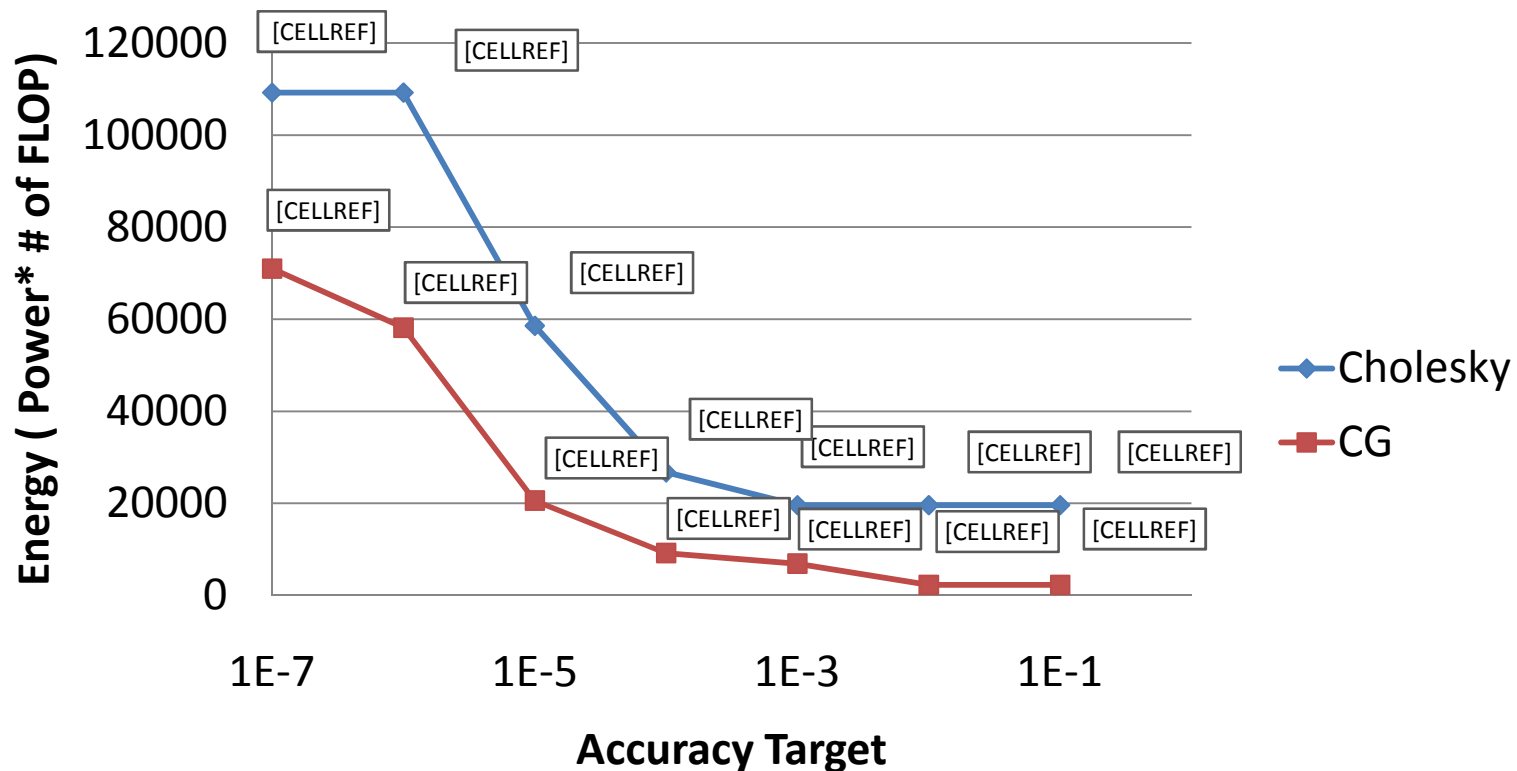


- Floor on accuracy for given fault rate.

CG converges faster (1000 iterations vs. 5-10 iterations)

# Least Squares(100x10)

## Energy / Robustness Tradeoffs



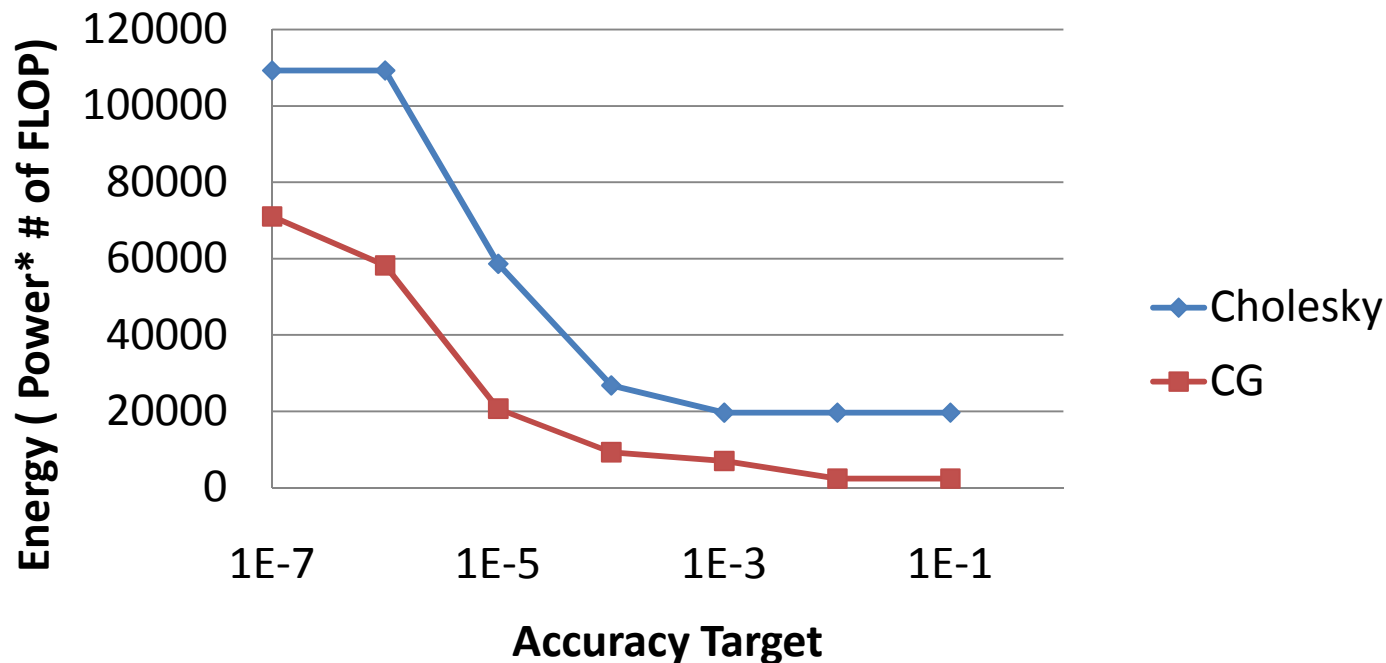
With symmetric positive definite inputs and relaxed accuracy targets, **more than an order of magnitude energy savings** over the best baseline (Cholesky).



# Least Squares(100x10)

## Energy / Robustness Tradeoffs

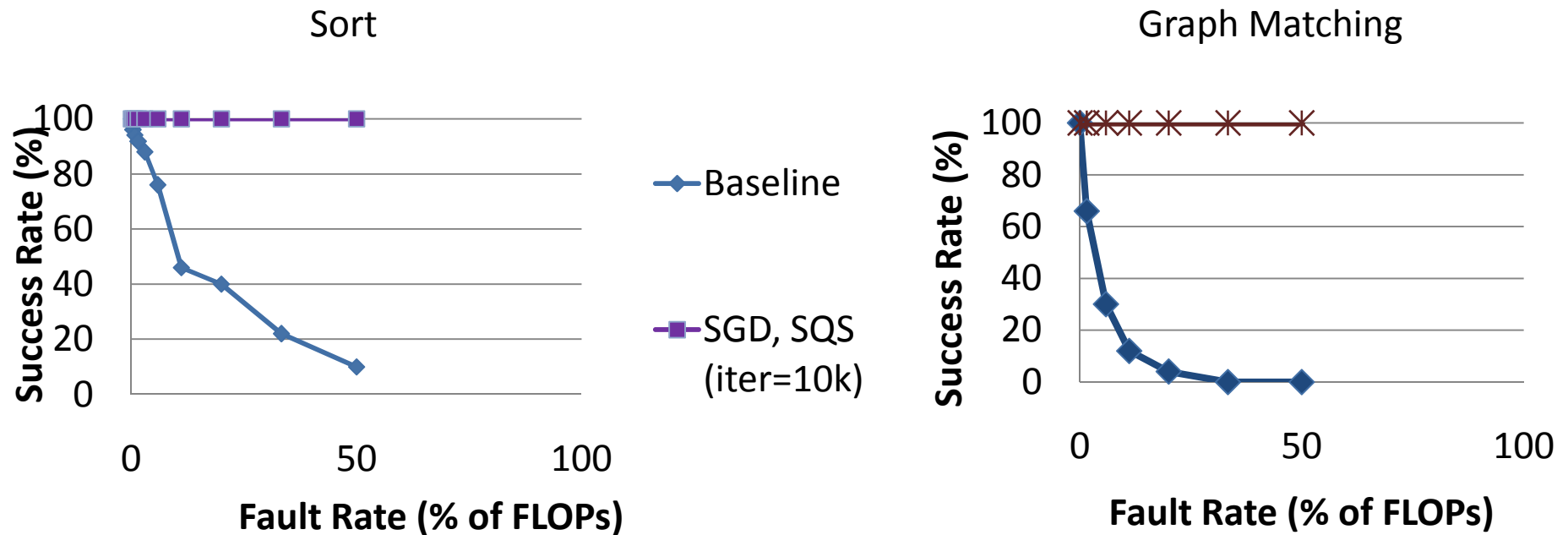
---



With symmetric positive definite inputs and relaxed accuracy targets, **more than an order of magnitude energy savings** over the best baseline (Cholesky).

# Sorting (size=10) and Graph Matching(5x6) using Gradient Descent

---



100% Accuracy with Sort and GM using SGD even in face of large error rates.

# Why optimization Solvers

---

- Iterative algorithms -> successive approximations to obtain more accurate solutions
- Optimizations problem - > Find best available solution among several alternatives
- Multiple acceptable possible answers
- Iterative -> repetitive -> redundant
- Approach
  - Get at solution fast
  - Evaluate goodness
  - Repeat
  - Succesively better outputs
- More data flow vs Control -->