

QED & Symbolic QED

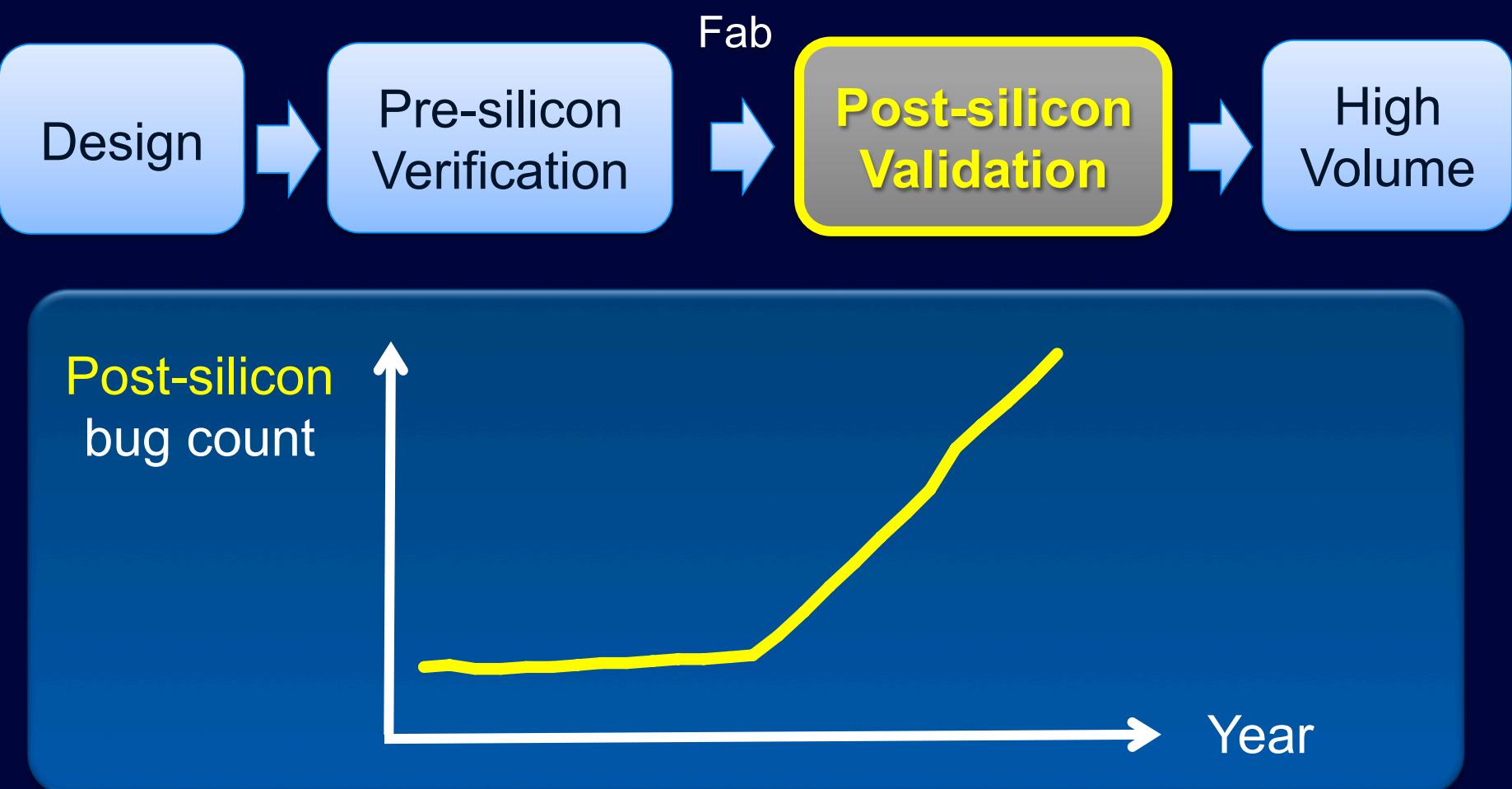
Subhasish Mitra



Department of EE & Department of CS
Stanford University

Acknowledgment: Students & Collaborators

Pre-Silicon Verification Inadequate



New architectural features limited by validation

Post-Silicon Validation Difficult

System tests



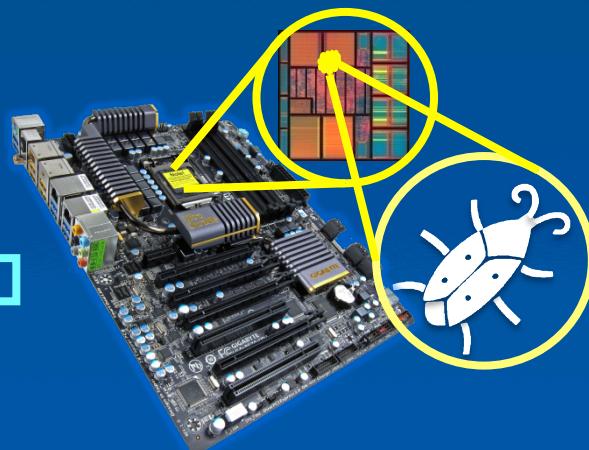
Detect



Weeks or months of manual work



Root-cause & fix



Localize

Scalability Barriers

- System-level failure reproduction
- Full system simulation

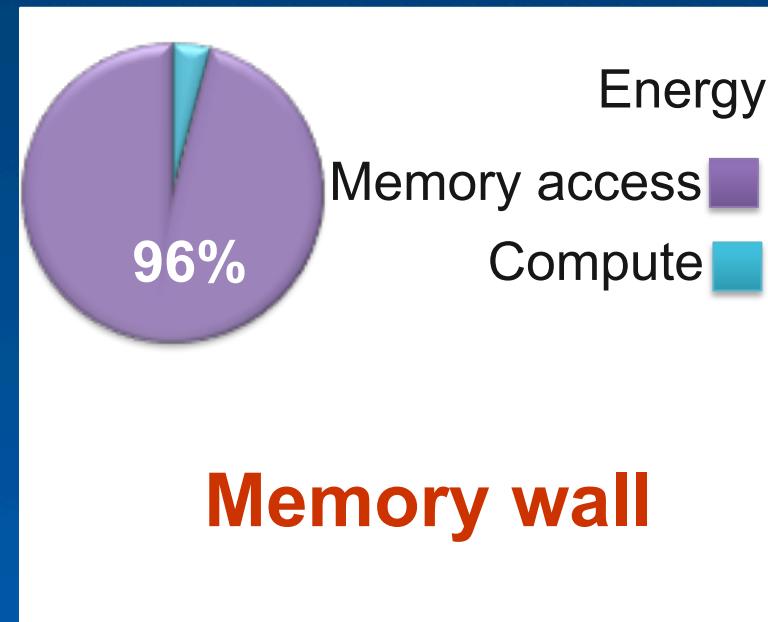
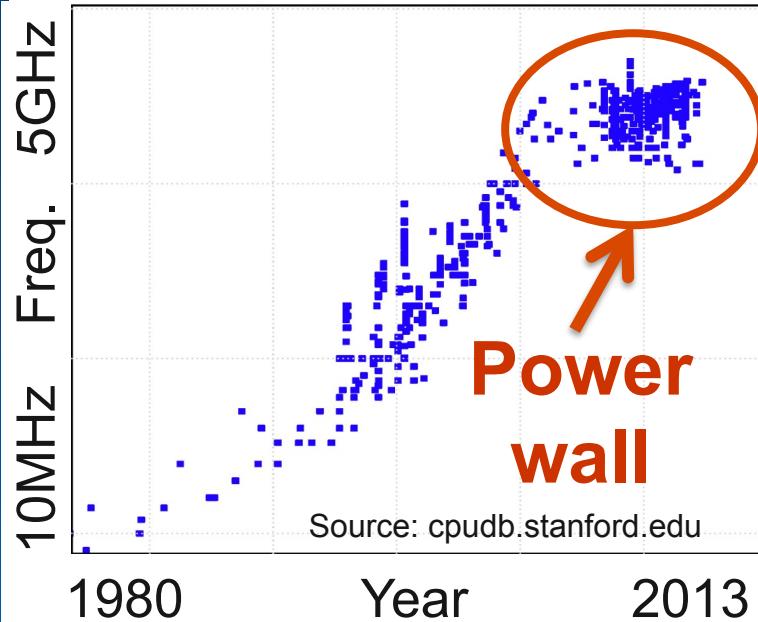


J. Stinson (ex-Intel)

Post-silicon costs rising faster than design cost

It's Only Getting Worse

Many walls simultaneously

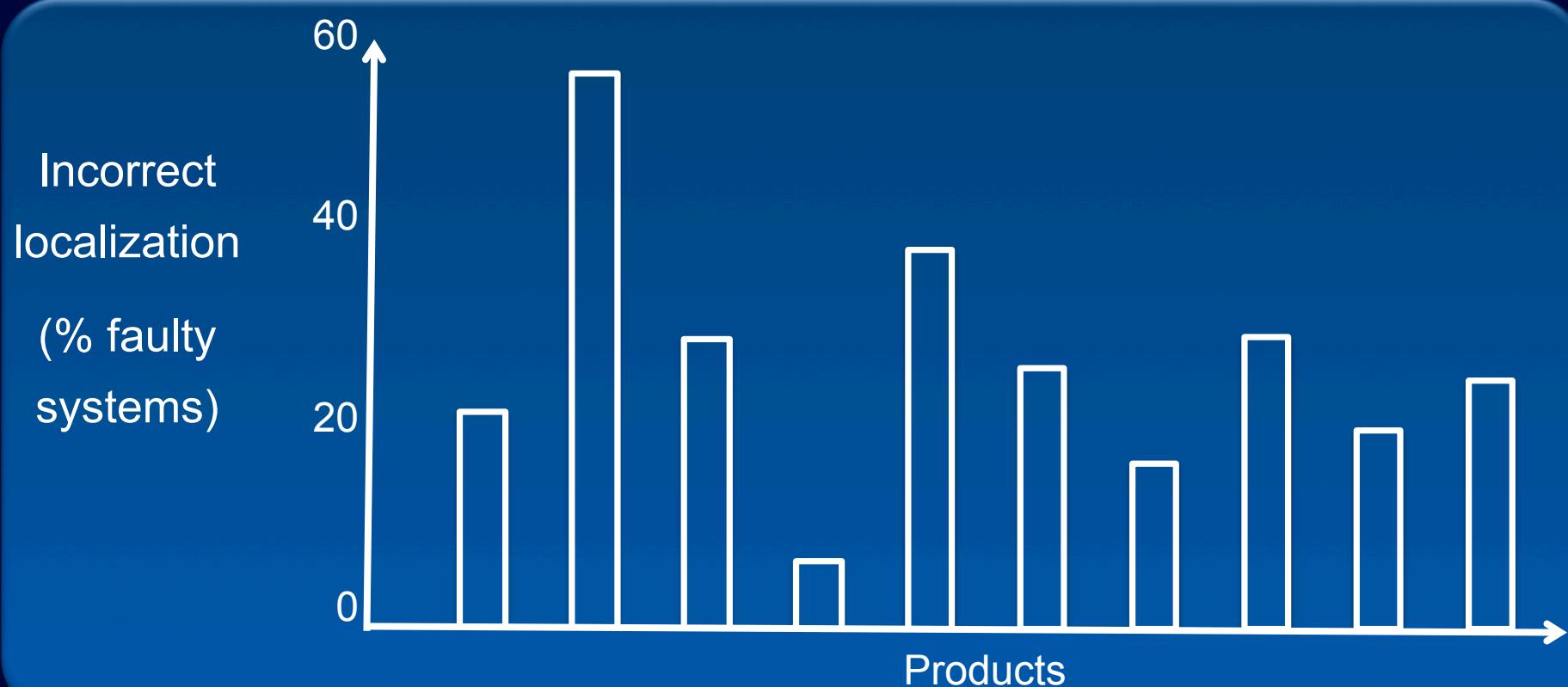


Also: interconnect wall, cooling wall, resilience wall, ...

Bigger Obstacles at System Level

1. Localize faulty IC
2. No Trouble Found

Incorrect localization: up to 60%



QED & Symbolic QED Results

- Pre-silicon formal verification

Few minutes to 7 hours

Automatic: no manual properties, constraints

Billion-transistor designs:

Processor cores, uncore, accelerators

QED & Symbolic QED Results

- Post-silicon validation & debug

20 flip-flops (out of 1 Million)

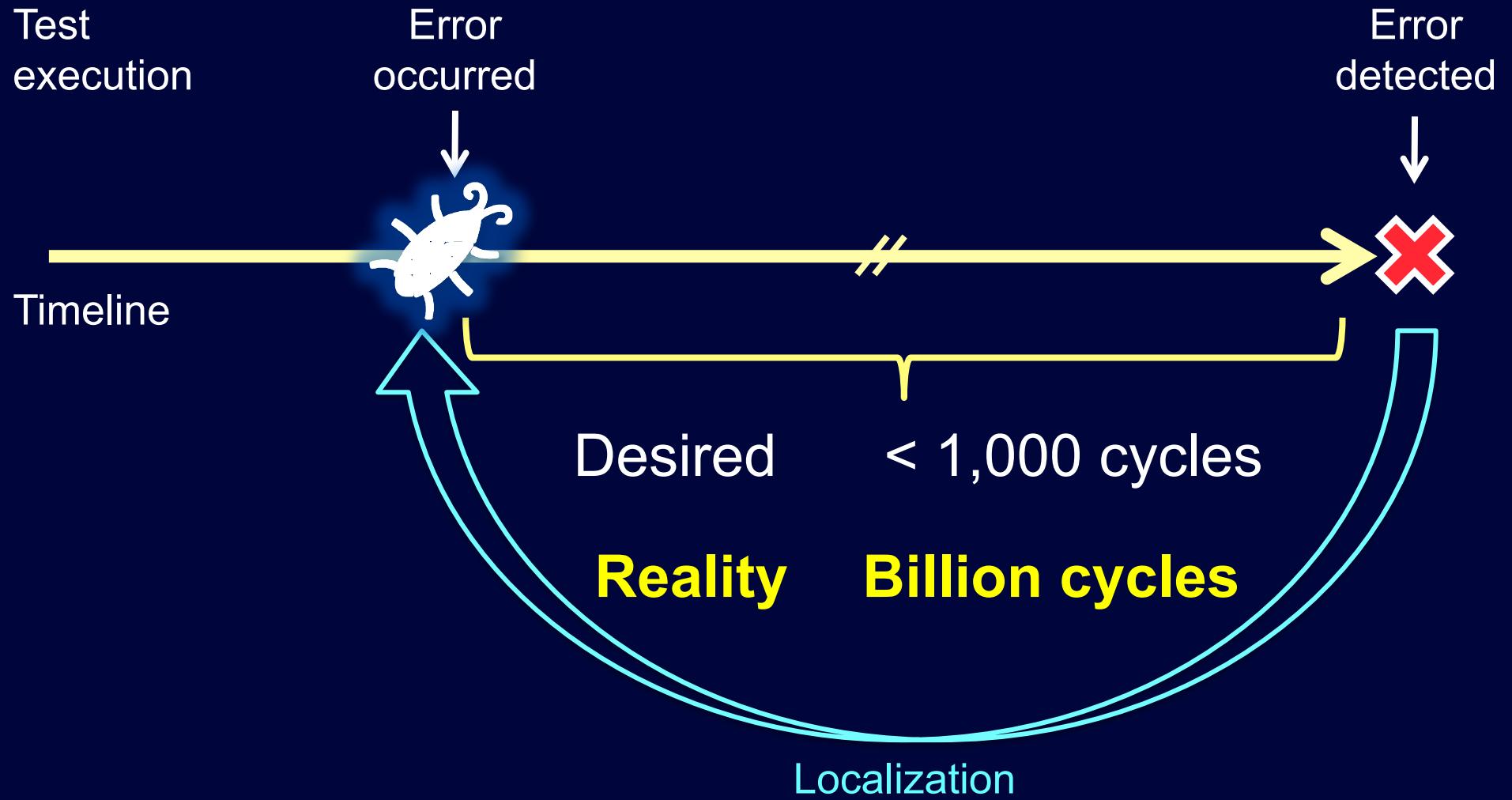
9 instructions (out of billions)

Automatic

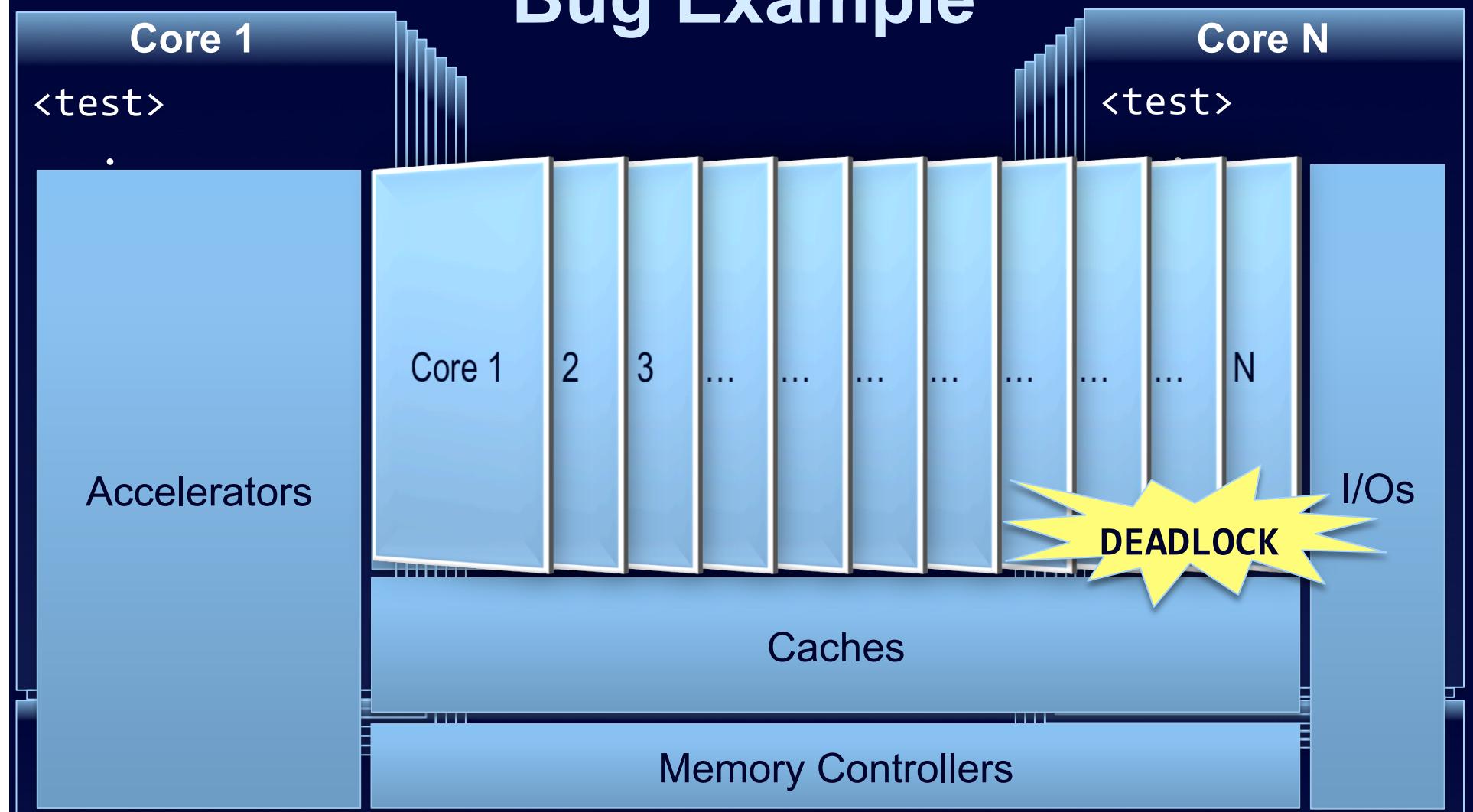
Few seconds to 9 hours

Billion-transistor designs

Error Detection Latency



Bug Example



Bug Example



Existing Techniques Inadequate

- Very long error detection latencies

Deadlock detection

Design-specific assertions

Self-checking tests

Store readback tests

Checkpointing

etc...

Quick Error Detection



- Error detection latency: guaranteed short
- Coverage: improved
- Software-only: readily applicable

Quick Error Detection

Snippet 1

Snippet 2

Snippet 3

Snippet 4

Snippet 5

Snippet 6

Snippet 7

...

**QED
transforms**

1. Wide variety
2. Diversity



Error detection latency target
Intrusiveness constraints

Snippet 1

QED: CFTSS-V 1

QED: EDDI-V
with diversity 1

QED: PLC 1

Snippet 2

QED: CFTSS-V 2

QED: EDDI-V
with diversity 2

QED: PLC 2

Snippet 3

...

Quick Error Detection

Snippet 1

Snippet 2

Snippet 3

Snippet 4

Snippet 5

Snippet 6

Snippet 7

...

**QED
transforms**

1. Wide variety
2. Diversity



Error detection latency target

Intrusiveness constraints



Snippet 1

QED: CFTSS-V 1

QED: EDDI-V
with diversity 1

QED: PLC 1

Snippet 2

QED: CFTSS-V 2

QED: EDDI-V
with diversity 2

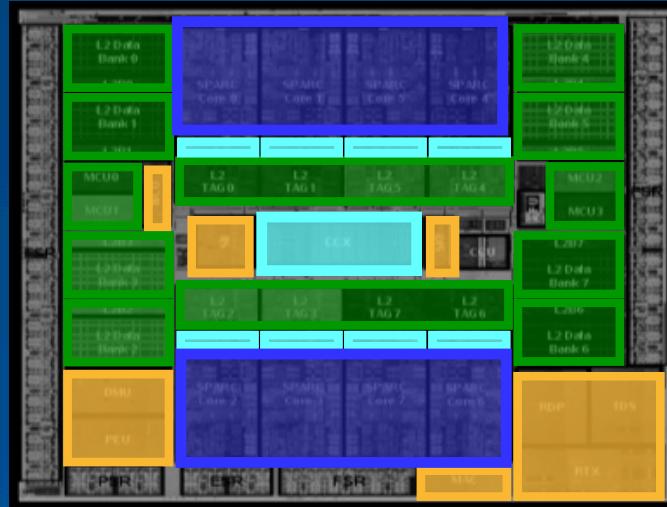
QED: PLC 2

Snippet 3

...

QED Transforms

System on Chip



Processor cores

Uncore, accelerators

EDDI-V

PLC

CFTSS-V

Fast QED

CFCSS-V

Hybrid QED

QED Example: Duplicate & Check

Validation program

Trace

```

void matgen (REAL a[], int lda, int n, REAL b[], REAL *norma)
{
    int init, i, j;
    init = 1325;
    /*norma = 0.0;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            if ((i + j) % 3 == 0) {
                init = 3125*init % 1000000000;
                a[i*lda+j] = init;
                a[i*lda+j+1] = init*32768/16384.0;
                *norma = (a[lda*j+i] > *norma) ? a[lda*j+i] : *norma;
            }
        }
    }
    for (i = 0; i < n; i++) {
        b[i] = 0.0;
    }
    for (j = 0; j < n; j++) {
        for (i = 0; i < n; i++) {
            b[i] = b[i] + a[lda*j+i];
        }
    }
}
return;
}

void dgefa(REAL a[], int lda, int n, int ipvt[], int *info){
/* internal variables */
REAL t;
int j,k,kpl,1,mn1;
/* gaussian elimination with partial pivoting */
*info = 0;
mn1 = n - 1;
if (mn1 < 0) {
    for (k = 0; k < mn1; k++) {
        kpl = k + 1;
        /* find l = pivot index */
        l = idamax(n-k,a[lda*k+k],1) + k;
        ipvt[k] = l;
        if (a[lda*k+k] == 0.0) {
            /* zero pivot implies this column already
            triangularized */
            if (a[lda*k+k] != 0.0) {
                /* interchange if necessary */
                if (l != k) {
                    t = a[lda*k+k];
                    a[lda*k+k] = a[lda*k+l];
                    a[lda*k+l] = t;
                }
            }
            /* compute multipliers */
            t = -ONE/a[lda*k+k];
            dscln(n-(k+1),t,sa[lda*k+k+1],1);
            /* row elimination with column indexing */
            for (j = kpl; j < mn1; j++) {
                t = a[lda*k+j];
                if (l != k) {
                    a[lda*k+j+1] = a[lda*k+j+k];
                    a[lda*k+j+k] = t;
                }
                dsaxy(n-(k+1),t,sa[lda*k+k+1],1,
                      sa[lda*k+j+k],1);
            }
            else {
                *info = k;
            }
        }
    }
}
if (mn1 == n-1) {
    if (a[lda*(n-1)*(n-1)] == ZERO) *info = n-1;
    checkExpected(a, ipvt, info);
    return;
}

void dgels(REAL a[],int lda,int n,int ipvt[],REAL b[],int job )
{
    REAL t;
    int k,kb,1,mn1;
    mn1 = n - 1;
    if (job == 0) {
        /* solve A*x = b */
        first_solve: l*y = b
        if (mn1 >= 1) {
            for (k = 0; k < mn1; k++) {
                l = ipvt[k];
                t = b[l];
                if (l != k) {
                    b[l] = b[k];
                    b[k] = t;
                }
                dsaxy(n-(k+1),t,sa[lda*k+k],1,b[b[k],1]);
            }
        }
        /* now solve A^T*x = y */
        for (kb = 0; kb < n; kb++)
        {
            l = ipvt[kb];
            b[kb] = b[l]/a[lda*kb+k];
            t = b[kb];
            dsaxy(n,k,t,sa[lda*kb+k],1,bb[0,1]);
        }
    }
    else {
}
}
```

$R1 \leftarrow R1 + 5$
 $R2 \leftarrow R2 - R1$
 $R3 \leftarrow R1 * R2$

• • •



Very
Long!



• • •

Check end_result



QED Example: Duplicate & Check

Validation program

QED Trace

```

void matgen (REAL a[], int lda, int n, REAL b[], REAL *norma)
{
    int init, i, j;
    init = 1325;
    /*norma = 0.0;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            int k = 3125*init % 1000000000;
            a[lda*j+i] = (k+32768.0)/16384.0;
            *norma = (a[lda*j+i] > *norma) ? a[lda*j+i] : *norma;
        }
    }
    for (i = 0; i < n; i++) {
        b[i] = 0.0;
    }
    for (j = 0; j < n; j++) {
        for (i = 0; i < n; i++) {
            b[i] = b[i] + a[lda*j+i];
        }
    }
}
return;
}

void dgefa(REAL a[], int lda, int n, int ipvt[], int *info){
/* internal variables */
REAL t;
int j,k,kpl,1,nn1;
/* gaussian elimination with partial pivoting */
*info = 0;
nn1 = n - 1;
if (nn1 < 0) {
    for (k = 0; k < nn1; k++) {
        kpl = k + 1;
        /* find l = pivot index */
        l = idamax(n-k,&a[lda*k+k],1) + k;
        ipvt[k] = l;
        if (a[l+k] == 0.0) {
            /* zero pivot implies this column already
            triangularized */
            if (a[l+k+1] != 0.0) {
                /* interchange if necessary */
                if (l != k) {
                    t = a[l+k+1];
                    a[l+k+1] = a[l+k];
                    a[l+k] = t;
                }
            }
            /* compute multipliers */
            t = -ONE/a[l+k+k];
            dscln(n-(k+1),t,&a[lda*k+k+1],1);
            /* row elimination with column indexing */
            for (j = kpl; j < nn1; j++) {
                t = a[l+k+j];
                if (l != k) {
                    a[l+k+j] = a[l+k+j] - t;
                    a[l+k+j+k] = a[l+k+j+k];
                }
                daxpy(n-(k+1),t,&a[lda*k+k+1],1,
                      &a[lda*j+k+1],1);
            }
            else{
                *info = k;
            }
        }
    }
}
if (nn1 == n-1) {
    if (a[lda*n-1]<=ZERO) *info = n-1;
    checkExpected(a, ipvt, info);
    return;
}

void dgesl(REAL a[],int lda,int n,int ipvt[],REAL b[],int job )
{
    REAL t;
    int k,kb,1,nn1;
    nn1 = n - 1;
    if (job == 0) {
        /* solve a * x = b */
        first_solve: t = b[0];
        if (nn1 >= 1) {
            for (k = 0; k < nn1; k++) {
                l = ipvt[k];
                t = b[l];
                if (l != k) {
                    b[l] = b[k];
                    b[k] = t;
                }
                daxpy(n-(k+1),t,&a[lda*k+k],1,&b[k+1],1 );
            }
        }
        /* now solve a^T * x = y */
        for (kb = 0; kb < n; kb++) {
            t = b[kb];
            b[kb] = b[kb]/a[lda*kb+k];
            t = -b[kb];
            daxpy(k,t,&a[lda*k+k],1,&b[0],1 );
        }
    }
    else {
}
}

```

R16 \leftarrow R1; R18 \leftarrow R2; R19 \leftarrow R3

• • •

R1 \leftarrow R1 + 5

R16 \leftarrow R16 + 5

R1 == R16

R2 \leftarrow R2 - R1

R18 \leftarrow R18 - R16

R2 == R18

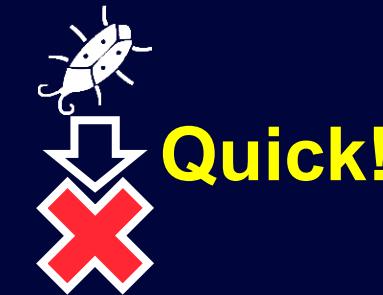
R3 \leftarrow R1 * R2

R19 \leftarrow R16 * R18

R3 == R19

• • •

Check end_result



Quick!

QED Improves Coverage

Validation program

Trace

R1

$R1 \leftarrow R1 + 1$



3 2

• • •

• • •

$R1 \leftarrow 0$

• • •

Error

Check $R1 == 0$ masked!

```

void matgen (REAL a[], int lda, int n, REAL b[], REAL *norma)
{
    int init, i, j;
    init = 1325;
    *norma = 0.0;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            init = 3125*init % 1000000000;
            a[i*lda+j] = init;
            if (i == j) a[i*lda+j] = 32768.0/16384.0;
            *norma = (a[lda*j+i] > *norma) ? a[lda*j+i] : *norma;
        }
    }
    return;
}

void dgesv(REAL a[], int lda, int n, int ipvt[], int *info){
    /* internal variables */
    REAL t;
    int j,k,kpl,1,nn1;
    /* gaussian elimination with partial pivoting */
    *info = 0;
    nn1 = n - 1;
    if (nn1 < 0) {
        for (k = 0; k < nn1; k++) {
            kpl = k + 1;
            /* find l = pivot index */
            l = idamax(n-k,&a[lda*k+k],1) + k;
            ipvt[k] = l;
            if (a[lda*k+k] == ZERO) {
                /* interchange if necessary */
                if (l != k) {
                    t = a[lda*k+l];
                    a[lda*k+l] = a[lda*k+k];
                    a[lda*k+k] = t;
                }
                /* compute multipliers */
                t = -ONE/a[lda*k+k];
                dscln(n-(k+1),t,&a[lda*k+k+1],1);
                /* row elimination with column indexing */
                for (j = kpl; j < nn1; j++) {
                    t = a[lda*k+l];
                    if (l != k) {
                        a[lda*j+k+1] = a[lda*j+k];
                        a[lda*j+k] = t;
                    }
                    daxpy(n-(k+1),t,&a[lda*k+k+1],1,
                          &a[lda*j+k+1],1);
                }
            }
            else {
                *info = k;
            }
        }
    }
    if (nn1 == n-1) {
        if (a[lda*(n-1)*(n-1)] == ZERO) *info = n-1;
        checkExpected(a, ipvt, info);
        return;
    }

void dgesl(REAL a[],int lda,int n,int ipvt[],REAL b[],int job )
{
    REAL t;
    int k,kb,1,nn1;
    nn1 = n - 1;
    if (job == 0) {
        /* solve a * x = b */
        first_solve: t = y;
        if (nn1 >= 1) {
            for (k = 0; k < nn1; k++) {
                l = ipvt[k];
                t = b[l];
                if (l != k) {
                    b[k] = t;
                    b[l] = b[l]/a[lda*k+k];
                    b[k] = t;
                }
                daxpy(n-(k+1),t,&a[lda*k+k+1],1,&b[k+1],1 );
            }
        }
        /* now solve a^T*x = y */
        for (kb = 0; kb < n; kb++)
        {
            l = ipvt[kb];
            b[kb] = b[l]/a[lda*k+k];
            t = -b[kb];
            daxpy(k,t,&a[lda*k+k],1,&b[0],1 );
        }
    }
    else {
}
}

```

QED Improves Coverage

Validation program

```

void matgen (REAL a[], int lda, int n, REAL b[], REAL *norma)
{
    int init, i, j;
    init = 1325;
    *norma = 0.0;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            if (i == j) {
                init = 3125*init % 1000000000;
                a[i*lda+j] = init;
                b[i] = 32768*init % 1000000000;
                if (a[i*lda+j] > *norma) a[i*lda+j] = *norma;
            }
        }
    }
    return;
}

void dgesv(REAL a[], int lda, int n, int ipvt[], int *info){
    /* internal variables */
    REAL t;
    int j,k,kpl,1,mn;
    /* gaussian elimination with partial pivoting */
    *info = 0;
    mn = n - 1;
    if (mn < 0) {
        for (k = 0; k < mn; k++) {
            kpl = k + 1;
            /* find l = pivot index */
            l = idamax(n-k,a[lda*k+k],1) + k;
            ipvt[k] = l;
            if (a[l*lda+k] == 0.0) {
                /* interchange if necessary */
                if (l != k) {
                    t = a[l*lda+k];
                    a[l*lda+k] = a[lda*k+k];
                    a[lda*k+k] = t;
                }
                /* compute multipliers */
                t = -ONE/a[l*lda+k];
                dscln(n-(k+1),t,sa[lda*k+k+1],1);
                /* row elimination with column indexing */
                for (j = kpl; j < mn; j++) {
                    t = a[j*lda+k];
                    if (l != k) {
                        a[j*lda+k+1] = a[j*lda+k];
                        a[j*lda+k] = t;
                    }
                    daxpy(n-(k+1),t,sa[lda*k+k+1],1,
                          sa[lda*j+k+1],1);
                }
            }
            else{
                *info = k;
            }
        }
    }
    if (mn == n-1) {
        if (a[(mn)*mn+(mn)] == ZERO) *info = n-1;
        checkExpected(a, ipvt, info);
        return;
    }

    void dgesl(REAL a[],int lda,int n,int ipvt[],REAL b[],int job )
    {
        REAL t;
        int k,kb,1,mn;
        mn = n - 1;
        if (job == 0) {
            /* solve a * x = b */
            first_solve: l*y = b;
            if (mn >= 1) {
                for (k = 0; k < mn; k++) {
                    l = ipvt[k];
                    t = b[l];
                    if (l != k) {
                        b[l] = b[k];
                        b[k] = t;
                    }
                    daxpy(n-(k+1),t,sa[lda*k+k+1],1,bb[k+1],1 );
                }
            }
            /* now solve a^T*x = y */
            for (kb = 0; kb < n; kb++) {
                l = ipvt[kb];
                b[kb] = b[l]/sa[lda*k+k];
                t = -bb[kb];
                daxpy(k,t,sa[lda*k+k],1,bb[0],1 );
            }
        }
    }
}

```

QED Trace

• • •

R1 \leftarrow R1 + 1
R18 \leftarrow R18 + 1
R1 == R18



• • •

R1 \leftarrow 0

• • •

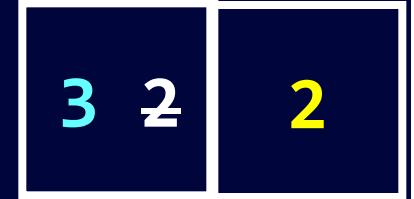
Error

Check R1 == 0 masked!

No QED: bug escape
QED: quick detection

R1

R18



Diversity-Enhanced QED

Validation program

QED Trace

```

void matgen (REAL a[], int lda, int n, REAL b[], REAL *norma)
{
    int init, i, j;
    init = 1325;
    *norma = 0.0;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            int s = 3125*init % 1000000000;
            if (i == 0) a[i*n+j] = s;
            else a[i*n+j] = 32768.0/16384.0;
            *norma = (a[lda*j+i] > *norma) ? a[lda*j+i] : *norma;
        }
    }
    for (i = 0; i < n; i++) {
        b[i] = 0.0;
    }
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            b[i*n+j] = a[lda*j+i];
        }
    }
    return;
}

void dgesv(REAL a[], int lda, int n, int ipvt[], int *info){
    /* internal variables */
    REAL t;
    int j,k,kpl,1,nn1;
    /* gaussian elimination with partial pivoting */
    *info = 0;
    nn1 = n - 1;
    if (nn1 <= 0) {
        for (k = 0; k < nn1; k++) {
            kpl = k + 1;
            /* find l = pivot index */
            l = idamax(n-k,&a[lda*k+k],1) + k;
            ipvt[k] = l;
            /* zero pivot implies this column already
               triangularized */
            if (a[l*lda+k] == ZERO) {
                /* interchange if necessary */
                if (l != k) {
                    t = a[l*lda+k];
                    a[l*lda+k] = a[k*lda+k];
                    a[k*lda+k] = t;
                }
                /* compute multipliers */
                t = -ONE/a[l*lda+k];
                dscln(n-(k+1),t,&a[lda*k+k+1],1);
            }
            /* row elimination with column indexing */
            for (j = kpl; j < nn1; j++) {
                t = a[j*lda+k];
                if (l != k) {
                    a[j*lda+k+1] = a[l*lda+j+k];
                    a[l*lda+j+k] = t;
                }
                daxpy(n-(k+1),t,&a[lda*k+k+1],1,
                      &a[lda*j+k+1],1);
            }
            else {
                *info = k;
            }
        }
    }
    if (nn1 == n-1) {
        if (a[nn1*lda+n-1] == ZERO) *info = n-1;
        checkExpected(a, ipvt, info);
        return;
    }

void dgesl(REAL a[],int lda,int n,int ipvt[],REAL b[],int job )
{
    REAL t;
    int k,kb,1,nn1;
    nn1 = n - 1;
    if (job == 0) {
        /* solve A*x = b */
        first_solve: t = x * b;
        if (nn1 >= 1) {
            for (k = 0; k < nn1; k++) {
                l = ipvt[k];
                t = b[l];
                if (l != k) {
                    b[l] = b[k];
                    b[k] = t;
                }
                daxpy(n-(k+1),t,&a[lda*k+k],1,&b[k+1],1 );
            }
        }
        /* now solve A^T*x = y */
        for (kb = 0; kb < n; kb++) {
            t = b[kb];
            b[kb] = b[ipvt[kb]];
            b[ipvt[kb]] = t;
            daxpy(k,t,&a[lda*k+kb],1,&b[0],1 );
        }
    }
}

```

$$a \leftarrow 1; a' \leftarrow a; b \leftarrow 1, b' \leftarrow b$$

$$R1 \leftarrow a + b$$

$$R18 \leftarrow 5a' + 5b'$$



6 2

R1

14 10

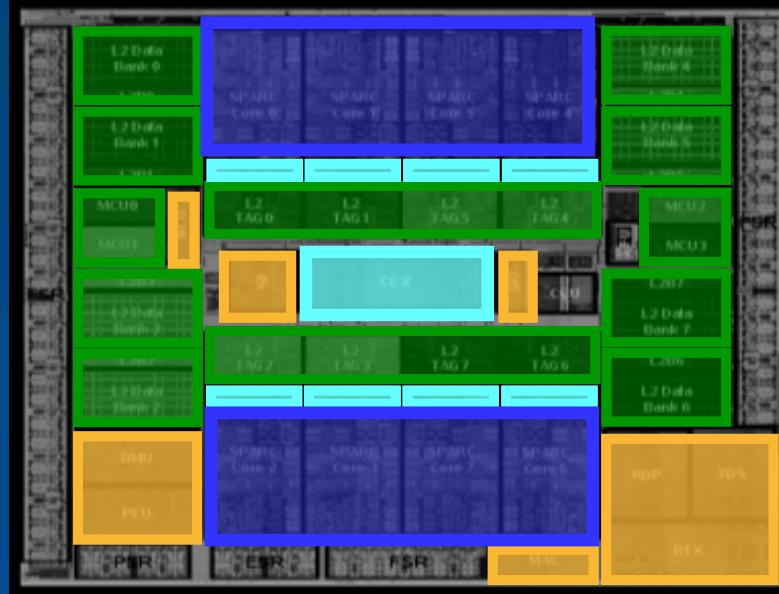
R18

$$5 \times R1 == R18 \quad \text{X}$$

Many diversity techniques

e.g., ED⁴I [Oh IEEE Trans. Comp. 02]

QED Transforms



Processor cores

EDDI-V

CFTSS-V

CFCSS-V

Uncore, accelerators

PLC

Fast QED

Hybrid QED

Original Test

Core 1

A = B * 2

Store mem[1] ← C

Core 2

E = F * G

H = D + E

Store mem[5] ← H

Core N

I = E / 2

Load J ← mem[7]

K = J + 1

• • •

QED: Duplicate & Check

Core 1

$$A' = A \cdot B + B' \cdot C' \quad C' = C$$

$$A = B * 2 \quad * \quad 2$$

$$A' = B * B' * 2$$

Check(A==A')

Store mem[1] $\leftarrow C'$

Store mem[1'] $\leftarrow C'$

Core 2

$$D' = D \cdot E + E' \cdot F \quad F' = F$$

$$G' = G \cdot H + H' \cdot I$$

$$E = E * G \quad * \quad G$$

$$E' = E * F' * G'$$

Check(E==E')

$$H = D + E + E$$

$$H' = D' + E' + E'$$

Check(H==H')

Store mem[5] $\leftarrow H$

Store mem[5'] $\leftarrow H'$

• • •

Core N

$$I' = E / 2 \quad / \quad 2$$

$$J' = E' / 2$$

Check(I==I')

Load J \leftarrow mem[7] \leftarrow mem[7]

Load J' \leftarrow mem[7'] \leftarrow mem[7']

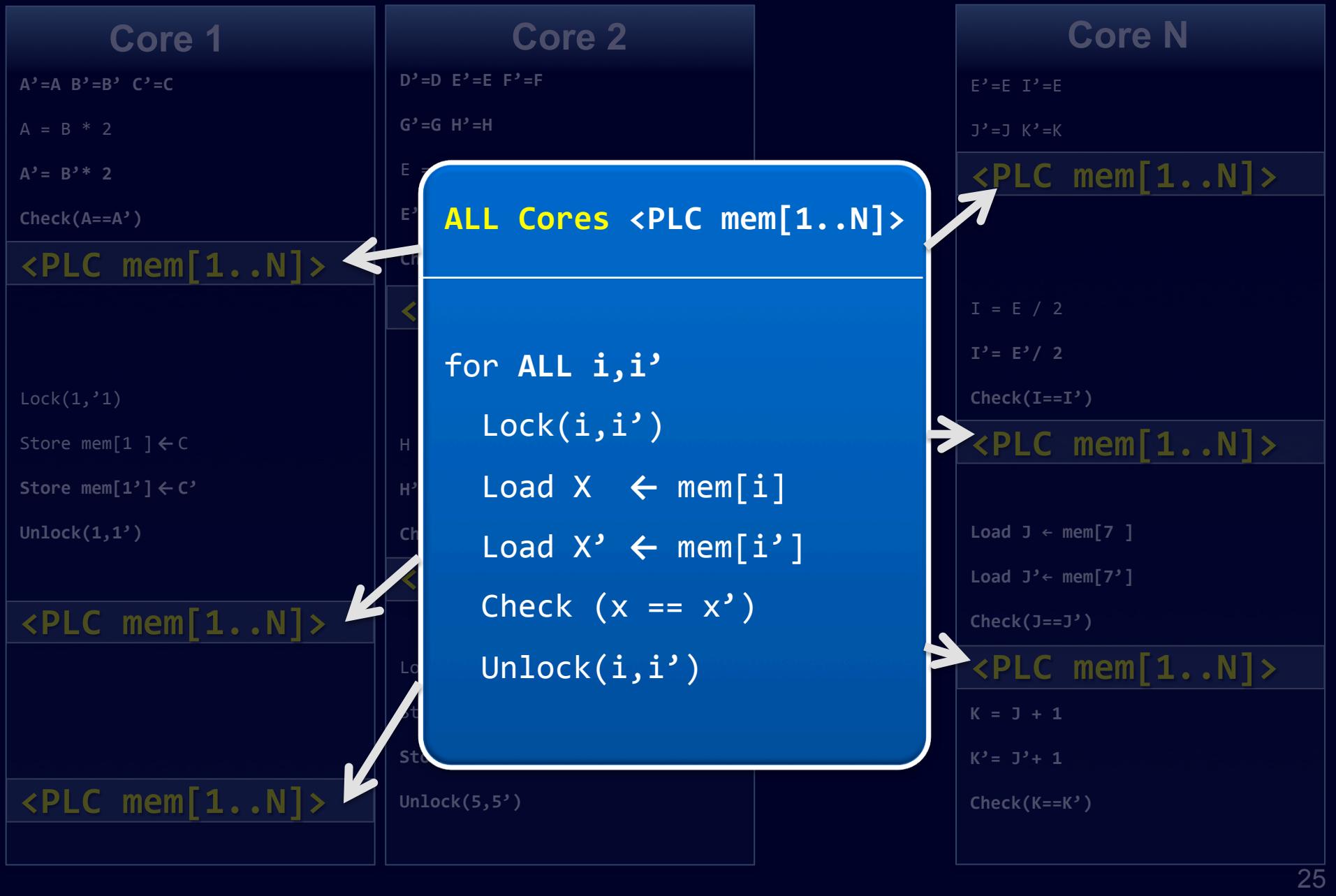
Check(J==J')

$$K = J + 1 \quad + \quad 1$$

$$K' = J' + 1 \quad + \quad 1$$

Check(K==K')

QED: Proactive Load & Check



QED Coverage Considerations

- Challenge
 - Intrusiveness: coverage impact ?
- Systematic solutions
 - QED family tests
 - Hardware-enhanced QED

Error Detection Latency vs. Intrusiveness

😊 Improved error detection latency

😢 Increased intrusiveness?

```

void daxpy (int n1, REAL y[], int n2, int ldm, REAL x[], REAL m[], int n3p, REAL yp[], int n2p, int ldm, REAL
yp[], REAL mp[])
{
    int j,i,jmin;
    int jp,ip,jminp;
    ...
    for (j = jmin-1, jp=jminp; j < n2; j = j + 21,jp+=21) {
        for (i = 0, ip=0; i < n1; i++, ip++) {
            y[i] = (((((((((((( y[i]
                + x[j-20]*m[ldm*(j-20)+i])
                + x[j-19]*m[ldm*(j-19)+i])
                + x[j-18]*m[ldm*(j-18)+i])
                + x[j-17]*m[ldm*(j-17)+i])
                + x[j-16]*m[ldm*(j-16)+i])
                + x[j-15]*m[ldm*(j-15)+i])
                + x[j-14]*m[ldm*(j-14)+i])
                + x[j-13]*m[ldm*(j-13)+i])
                + x[j-12]*m[ldm*(j-12)+i])
                + x[j-11]*m[ldm*(j-11)+i])
                + x[j-10]*m[ldm*(j-10)+i])
                + x[j- 9]*m[ldm*(j- 9)+i])
                + x[j- 8]*m[ldm*(j- 8)+i])
                + x[j- 7]*m[ldm*(j- 7)+i])
                + x[j- 6]*m[ldm*(j- 6)+i])
                + x[j- 5]*m[ldm*(j- 5)+i])
                + x[j- 4]*m[ldm*(j- 4)+i])
                + x[j- 3]*m[ldm*(j- 3)+i])
                + x[j- 2]*m[ldm*(j- 2)+i])
                + x[j- 1]*m[ldm*(j- 1)+i])
                + x[j]*m[ldm*j+i];
        }
    }
    ...
    y[lp] = (((((((((( y[lp]
        + xp[j-20]*mp[ldm*(lp-20)+lp])
        + xp[j-19]*mp[ldm*(lp-19)+lp])
        + xp[j-18]*mp[ldm*(lp-18)+lp])
        + xp[j-17]*mp[ldm*(lp-17)+lp])
        + xp[j-16]*mp[ldm*(lp-16)+lp])
        + xp[j-15]*mp[ldm*(lp-15)+lp])
        + xp[j-14]*mp[ldm*(lp-14)+lp])
        + xp[j-13]*mp[ldm*(lp-13)+lp])
        + xp[j-12]*mp[ldm*(lp-12)+lp])
        + xp[j-11]*mp[ldm*(lp-11)+lp])
        + xp[j-10]*mp[ldm*(lp-10)+lp])
        + xp[j- 9]*mp[ldm*(lp- 9)+lp])
        + xp[j- 8]*mp[ldm*(lp- 8)+lp])
        + xp[j- 7]*mp[ldm*(lp- 7)+lp])
        + xp[j- 6]*mp[ldm*(lp- 6)+lp])
        + xp[j- 5]*mp[ldm*(lp- 5)+lp])
        + xp[j- 4]*mp[ldm*(lp- 4)+lp])
        + xp[j- 3]*mp[ldm*(lp- 3)+lp])
        + xp[j- 2]*mp[ldm*(lp- 2)+lp])
        + xp[j- 1]*mp[ldm*(lp- 1)+lp])
        + xp[j]*mp[ldm*lp+i]);
    }
    ...
    check("daxpy bba", y[lp], y[lp]);
    check("daxpy bbb", 1, lp);
    ("daxpy bcc");
    ...
}
return;
}

void daxpy(int n, REAL da, REAL dx[], int incx, REAL dy[], int incy, int np, REAL ddp, REAL dpx[], int incxp,
REAL dpy[], int incyp)
{
    int i,ix,iy,m,mp1;
    int ip,ixp,iyp,np1p;
    ...
}

```

Snippet 1

Snippet 2

Diverse Execution

QED_check

```

void daxpy (int n1, REAL y[], int n2, int ldm, REAL x[], REAL m[], int n3p, REAL yp[], int n2p, int ldm, REAL
yp[], REAL mp[])
{
    int j,i,jmin;
    int jp,ip,jminp;
    ...
    for (j = jmin-1, jp=jminp; j < n2; j = j + 21,jp+=21) {
        for (i = 0, ip=0; i < n1; i++, ip++) {
            y[i] = (((((((((((( y[i]
                + x[j-20]*m[ldm*(j-20)+i])
                + x[j-19]*m[ldm*(j-19)+i])
                + x[j-18]*m[ldm*(j-18)+i])
                + x[j-17]*m[ldm*(j-17)+i])
                + x[j-16]*m[ldm*(j-16)+i])
                + x[j-15]*m[ldm*(j-15)+i])
                + x[j-14]*m[ldm*(j-14)+i])
                + x[j-13]*m[ldm*(j-13)+i])
                + x[j-12]*m[ldm*(j-12)+i])
                + x[j-11]*m[ldm*(j-11)+i])
                + x[j-10]*m[ldm*(j-10)+i])
                + x[j- 9]*m[ldm*(j- 9)+i])
                + x[j- 8]*m[ldm*(j- 8)+i])
                + x[j- 7]*m[ldm*(j- 7)+i])
                + x[j- 6]*m[ldm*(j- 6)+i])
                + x[j- 5]*m[ldm*(j- 5)+i])
                + x[j- 4]*m[ldm*(j- 4)+i])
                + x[j- 3]*m[ldm*(j- 3)+i])
                + x[j- 2]*m[ldm*(j- 2)+i])
                + x[j- 1]*m[ldm*(j- 1)+i])
                + x[j]*m[ldm*j+i];
        }
    }
    ...
    y[lp] = (((((((((( y[lp]
        + xp[j-20]*mp[ldm*(lp-20)+lp])
        + xp[j-19]*mp[ldm*(lp-19)+lp])
        + xp[j-18]*mp[ldm*(lp-18)+lp])
        + xp[j-17]*mp[ldm*(lp-17)+lp])
        + xp[j-16]*mp[ldm*(lp-16)+lp])
        + xp[j-15]*mp[ldm*(lp-15)+lp])
        + xp[j-14]*mp[ldm*(lp-14)+lp])
        + xp[j-13]*mp[ldm*(lp-13)+lp])
        + xp[j-12]*mp[ldm*(lp-12)+lp])
        + xp[j-11]*mp[ldm*(lp-11)+lp])
        + xp[j-10]*mp[ldm*(lp-10)+lp])
        + xp[j- 9]*mp[ldm*(lp- 9)+lp])
        + xp[j- 8]*mp[ldm*(lp- 8)+lp])
        + xp[j- 7]*mp[ldm*(lp- 7)+lp])
        + xp[j- 6]*mp[ldm*(lp- 6)+lp])
        + xp[j- 5]*mp[ldm*(lp- 5)+lp])
        + xp[j- 4]*mp[ldm*(lp- 4)+lp])
        + xp[j- 3]*mp[ldm*(lp- 3)+lp])
        + xp[j- 2]*mp[ldm*(lp- 2)+lp])
        + xp[j- 1]*mp[ldm*(lp- 1)+lp])
        + xp[j]*mp[ldm*lp+i]);
    }
    ...
    check("daxpy bba", y[lp], y[lp]);
    check("daxpy bbb", 1, lp);
    ("daxpy bcc");
    ...
}
return;
}

void daxpy(int n, REAL da, REAL dx[], int incx, REAL dy[], int incy, int np, REAL ddp, REAL dpx[], int incxp,
REAL dpy[], int incyp)
{
    int i,ix,iy,m,mp1;
    int ip,ixp,iyp,np1p;
    ...
}

```

Snippet 1

Diverse Execution

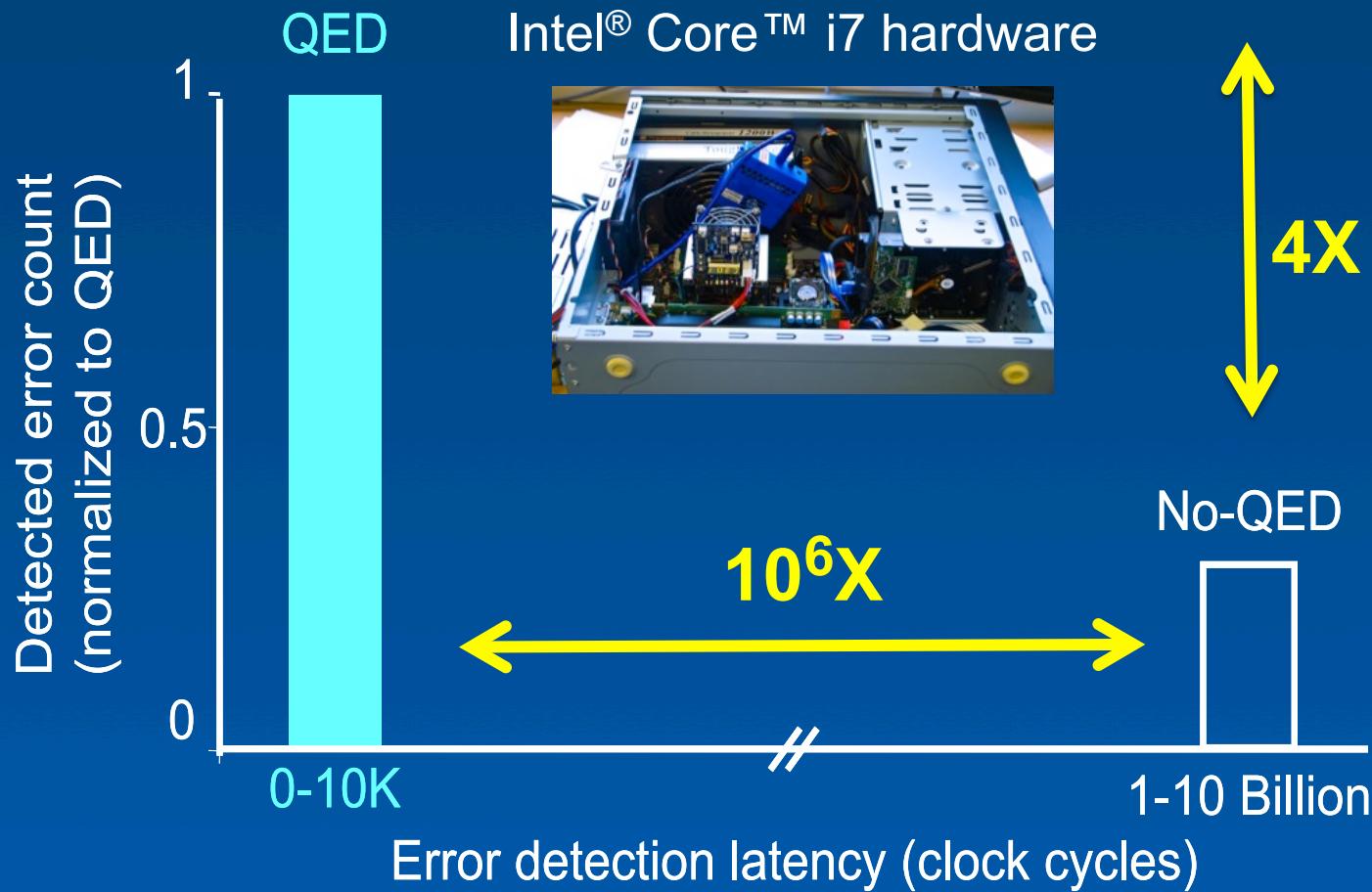
QED_check

Snippet 2

Diverse Execution

QED_check

QED Effective for Electrical Bugs



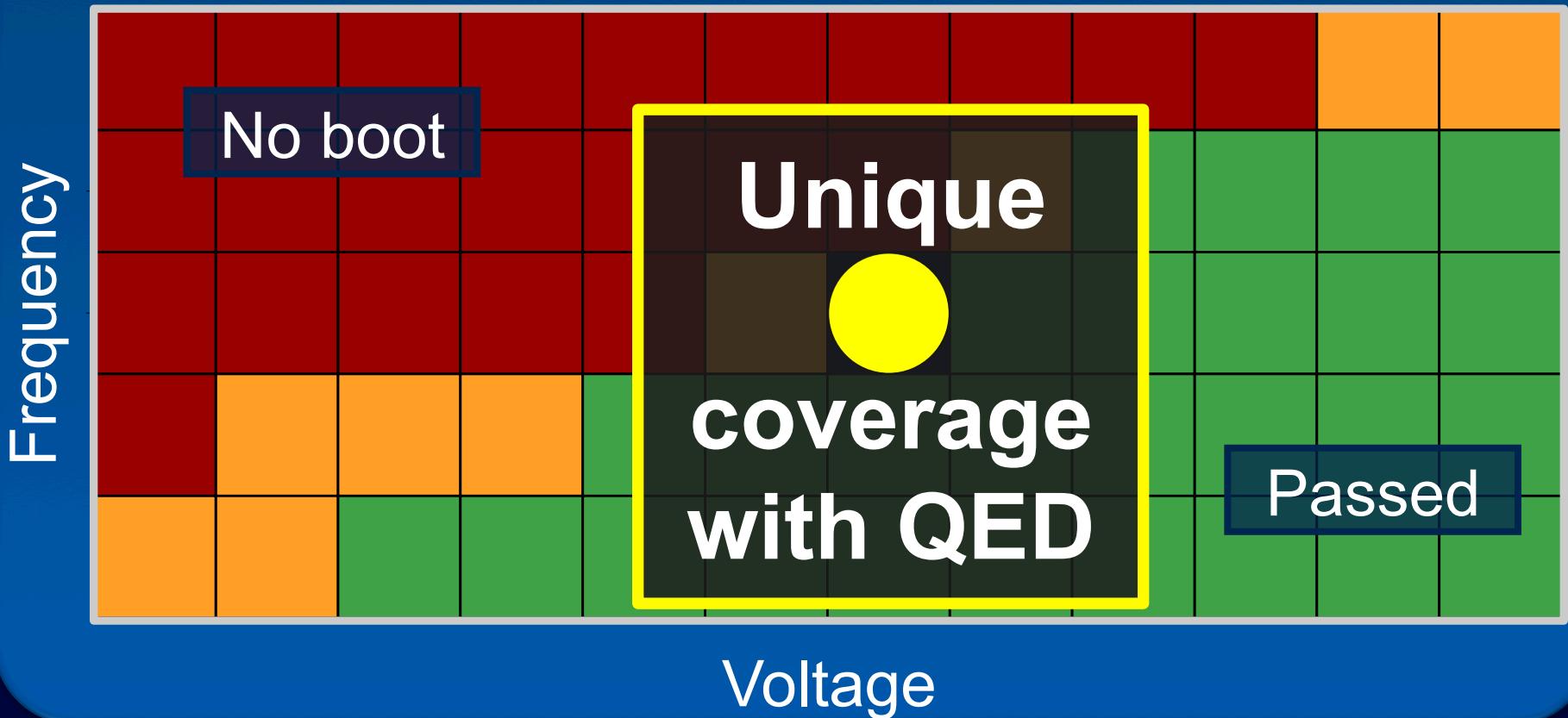
Intel® Core™ i7 Hardware



Error detected (QED and no-QED)



Error detected (QED only)



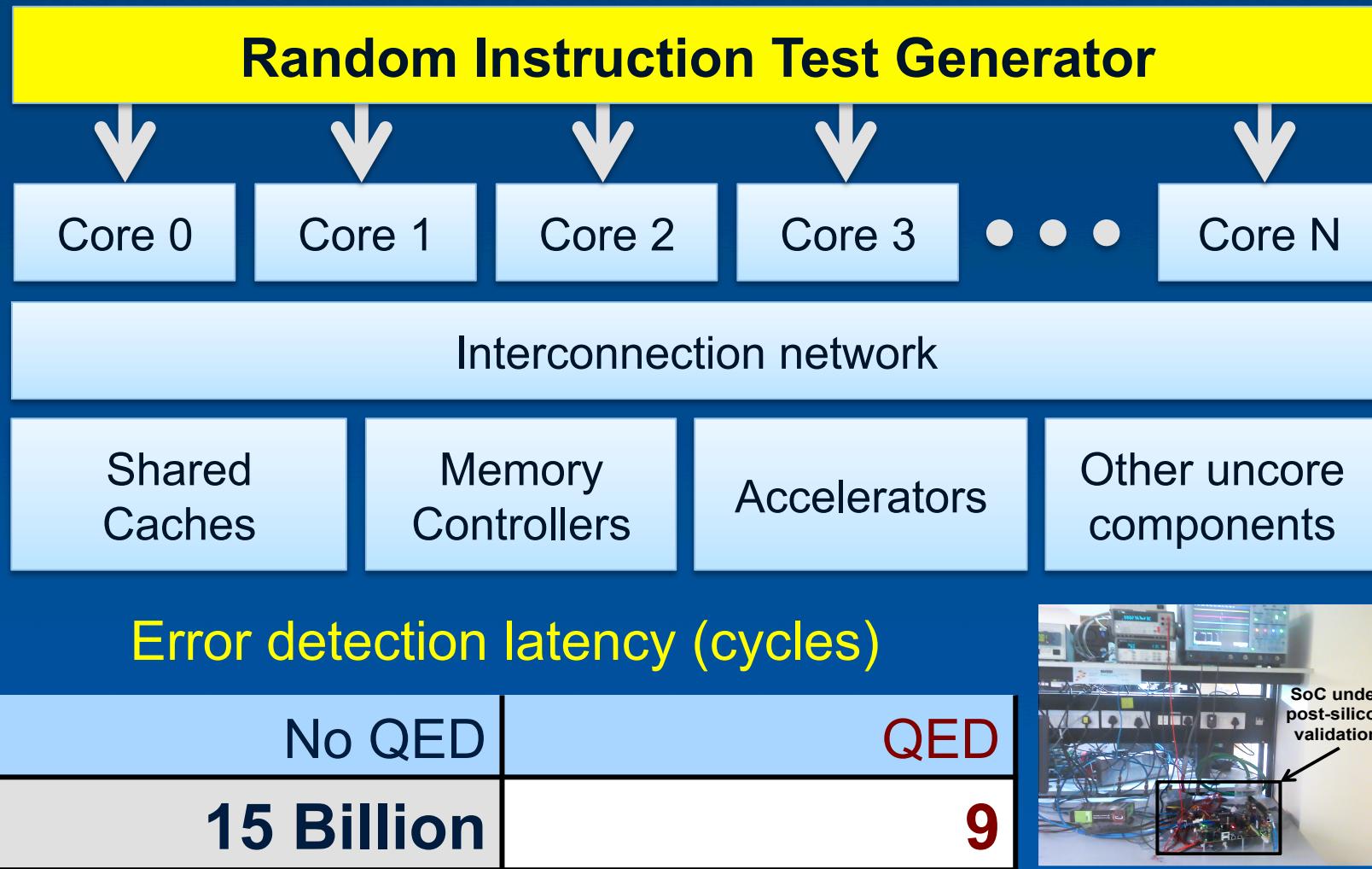
QED Effective for Logic Bugs

- “Difficult” logic bugs
 - Industrial bug databases
- Processor cores, accelerators, uncore,
power management



QED Logic Bugs

Freescale SoC hardware



Symbolic QED

- Logic bugs: pre-silicon & post-silicon

	Traditional	Symbolic QED
Pre-silicon	Doesn't scale	20 mins to 7 hrs, automatic
Post-silicon	Weeks, months (Manual)	Overnight, automatic
	Million cycle traces	3 to 22-cycle traces

OpenSPARC T2 SoC (500M transistors): difficult bugs inserted

Traditional Bounded Model Checking

Property

Model



If property violated



Counter-example = bug trace

Traditional BMC vs. Symbolic QED

	Traditional BMC	Symbolic QED
Properties	Manual	Automatic QED checks
Design size	Small blocks	Large SoCs
False positives	Possible	None
Input constraints	Manual	QED module

Traditional BMC Challenges

1. What property ?

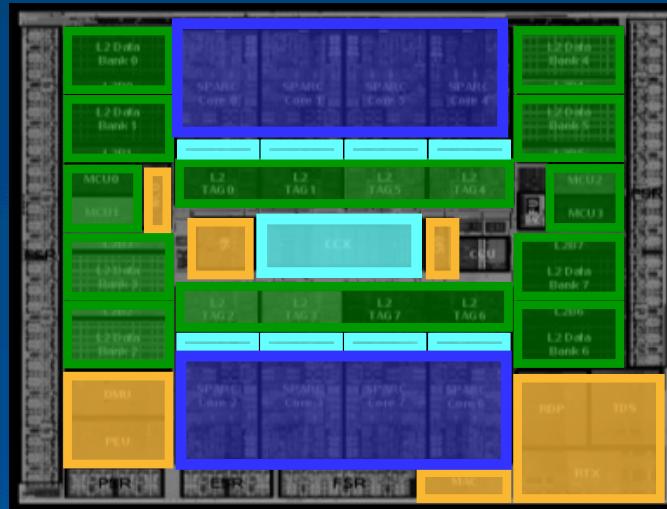
- Bugs not known *a priori*

2. Design size ?

- Big designs, cannot load

QED Transforms

System on Chip



Processor cores

Uncore, accelerators

EDDI-V

PLC

CFTSS-V

Fast QED

CFCSS-V

Hybrid QED

QED: Duplicate & Check

Input

```
R16 ← R1; R18 ← R2; R19 ← R3
```

• • •

```
R1 ← R1 + 5
```

```
R2 ← R2 - R1
```

```
R3 ← R1 * R2
```

```
R16 ← R16 + 5
```

```
R18 ← R18 - R16
```

```
R19 ← R16 * R18
```

```
CMP R1 == R16
```

```
CMP R2 == R18
```

```
CMP R3 == R19
```

• • •



Quick!



“Universal” Property: QED Check

CMP Ra == Ra'

- Ra – original register
- Ra' – corresponding duplicated register
- Ra ≠ Ra' – error detected

BMC Symbolic QED

“Universal” Property

Model



If property violated



Counter-example = bug trace

Need Input Constraints

- If unconstrained, BMC may choose any inputs

$Ra \leftarrow 1$

$Ra' \leftarrow 2$

CMP $Ra == Ra'$

- False fail – not a bug

Solution: QED Module

Only during BMC, not in fabricated chip

Input
(Chosen by BMC)

```
ST [0x10000], Ra
ST [0x10040], Rb
LD Rc, [0x10000]
```

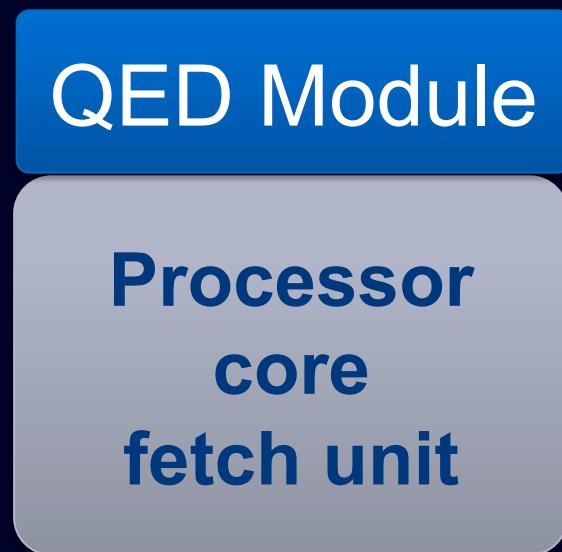
QED
Module

Output

```
ST [0x10000], Ra
ST [0x10040], Rb
LD Rc, [0x10000]
ST [0x20000], Ra'
ST [0x20040], Rb'
LD Rc', [0x20000]
<COMPARES>
```

Solution: QED Module

Only during BMC, not in fabricated chip



- Automatically duplicates instructions
- Determines when QED checks happen

BMC Symbolic QED

“Universal” Property

Model + QED Module
(no hardware overhead)



BMC Tool



If property violated



Counter-example = bug trace

Initial State Important

- If unconstrained, BMC may choose any initial state

$Ra = 1 \quad Ra' = 2 \quad // \text{initial state}$

$Ra = Ra + 1; Ra' = Ra' + 1; \text{CMP } Ra == Ra'$

- QED-consistent initial state
 - Run “simple” QED test
- More sophisticated [Fadiheh DATE 2018]

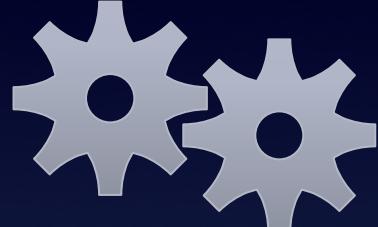
BMC Symbolic QED

“Universal” Property +
QED-consistent initial state

Model + QED Module
(no hardware overhead)



BMC Tool



If property violated



Counter-example = bug trace

Symbolic QED Results

RIDECORE: RISC-V super-scalar core

3 new bugs: previously unknown

Difficult & complex corner cases

~ 1 minute Symbolic QED run

Automatic: no manual properties, constraints, etc.

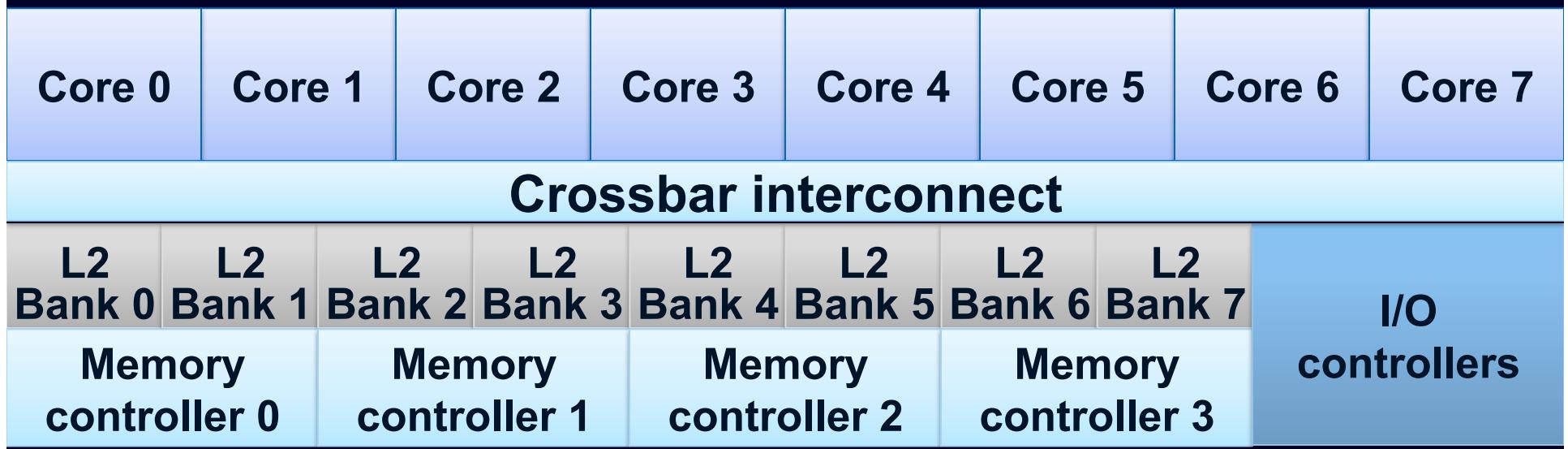
Confirmed & fixed by designers

(<https://github.com/ridecore/ridecore/issues/4>)

BUT...

- Big designs ?
- **Solution: QED checks compositional**
 - Preserved across partial instances
 - Not design- / implementation-specific

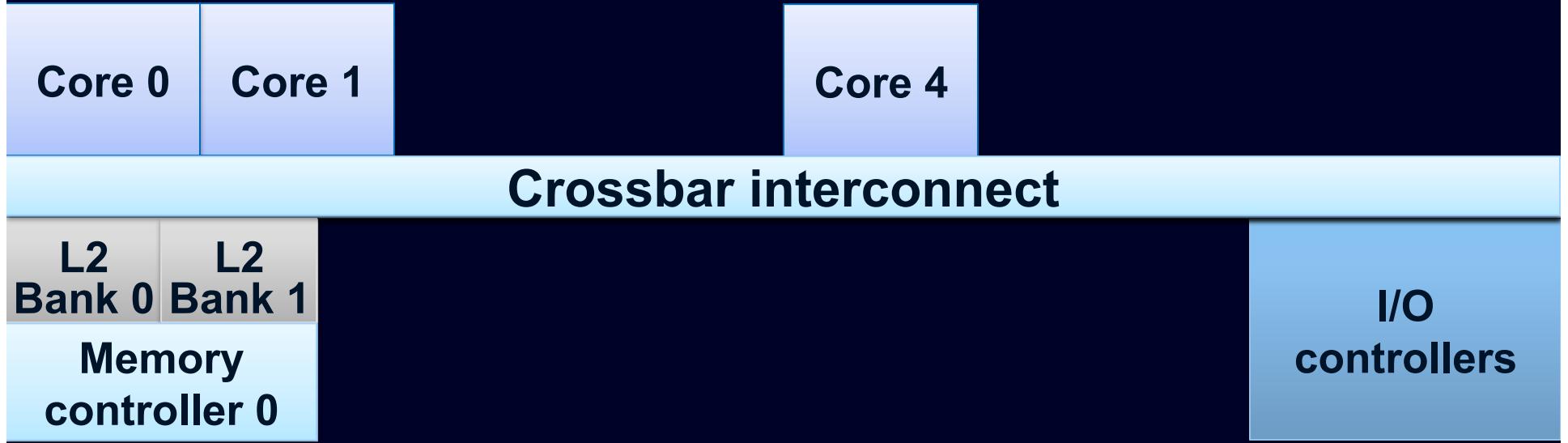
Solution: Partial Instantiation



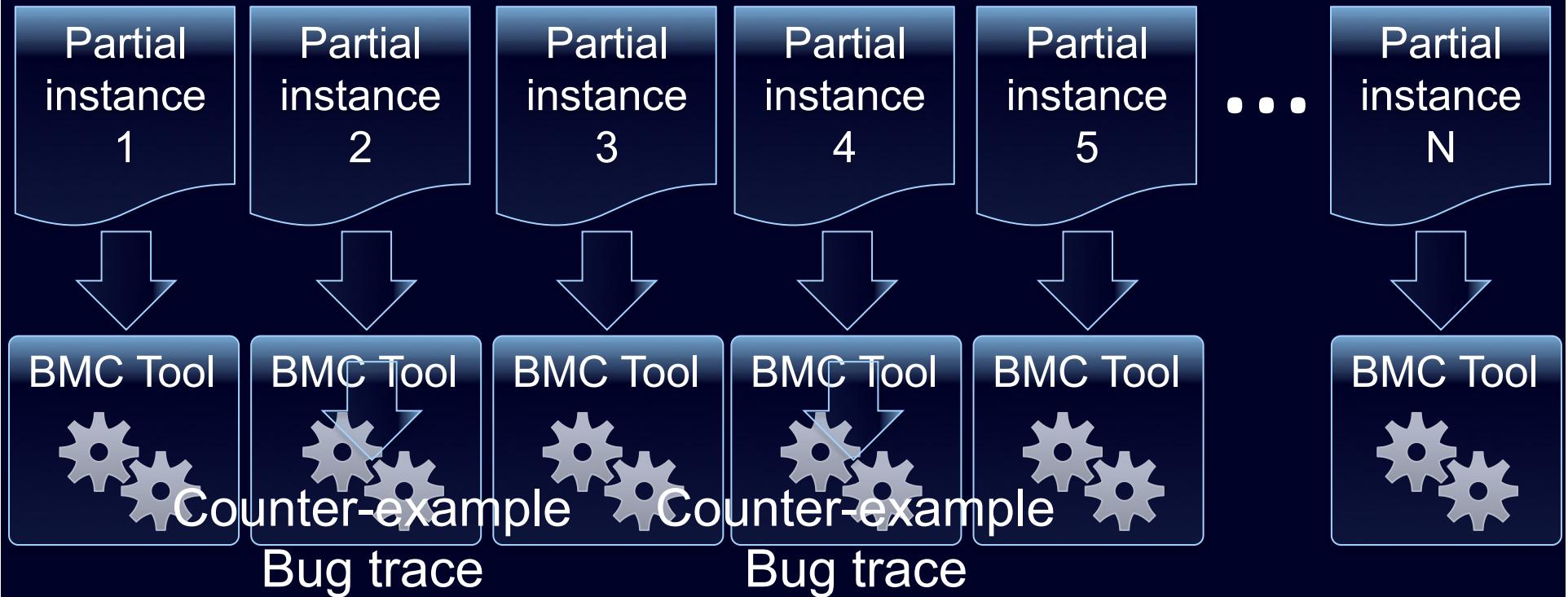
Solution: Partial Instantiation

Partial
instances

- At least 1 processor core per partial instance



BMC on Partial Instances



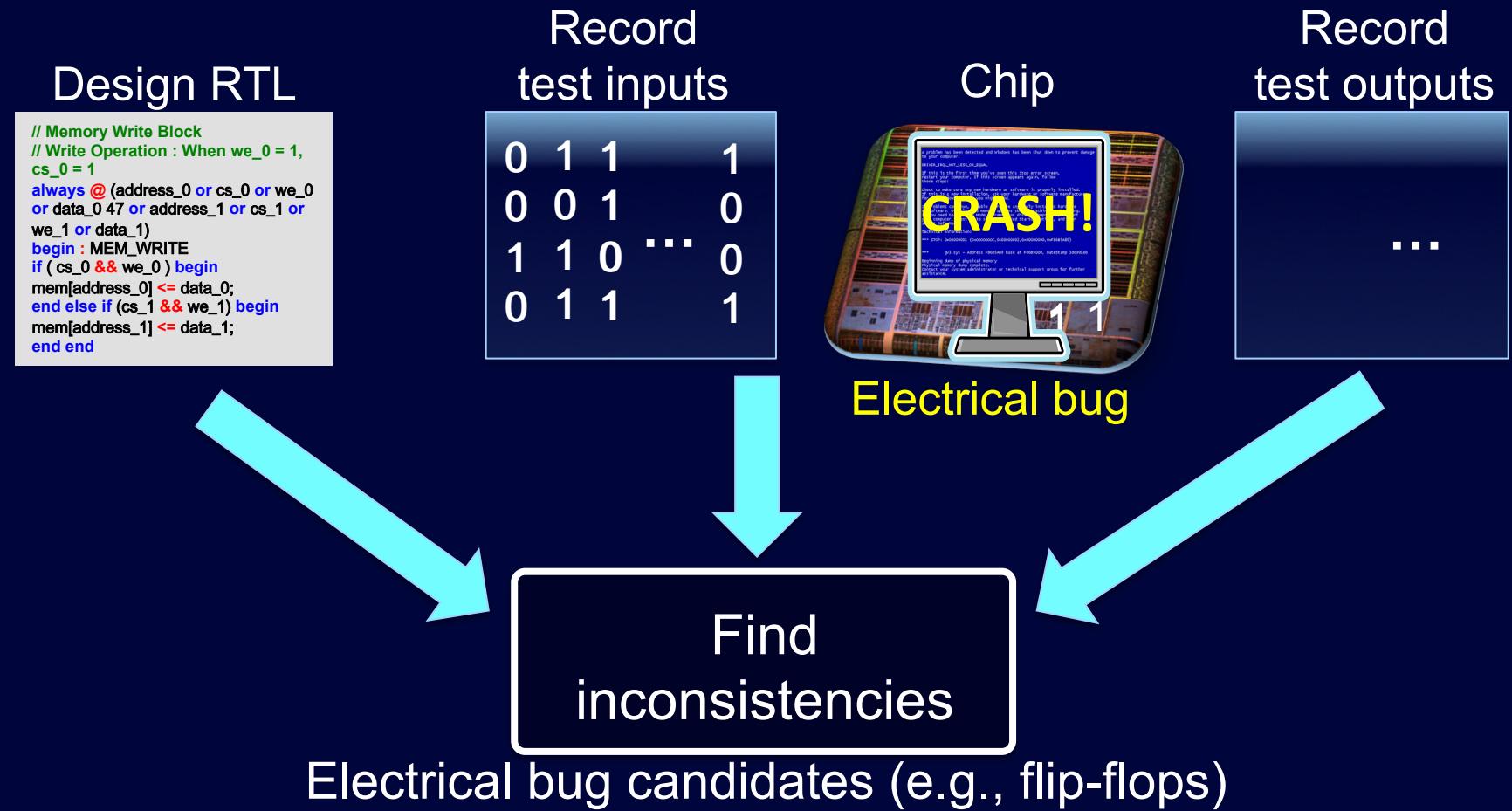
Symbolic QED Results

	Traditional	Symbolic QED
Pre-silicon	BMC doesn't scale	20 mins to 7 hrs, automatic
Post-silicon	Weeks, months (Manual)	Overnight, automatic
	Million cycle traces	3 to 22-cycle traces

OpenSPARC T2 SoC (500M transistors): difficult bugs inserted

E-QED

- Electrical bugs: post-silicon debug



E-QED

Design phase: E-QED signature blocks



QED post-silicon validation tests



Localize error: formal methods

E-QED

- Electrical bugs: post-silicon debug

	E-QED
Flip-flop candidates (~1 million total)	18 flip-flops (50,000× localization)
Area impact	2.5% (actually 0%)
Debug effort	Automatic
Runtime	~ 9 hours

OpenSPARC T2 SoC (500M transistors)

Thanks to Students & Collaborators



Thanks to Sponsors



QED & Symbolic QED

- Pre-silicon verification, post-silicon debug
- Billion-transistor SoCs
- Overnight (vs. weeks or months)
- Automatic (vs. manual)
- Broadly applicable
 - Core, uncore, accelerator, logic & electrical bugs