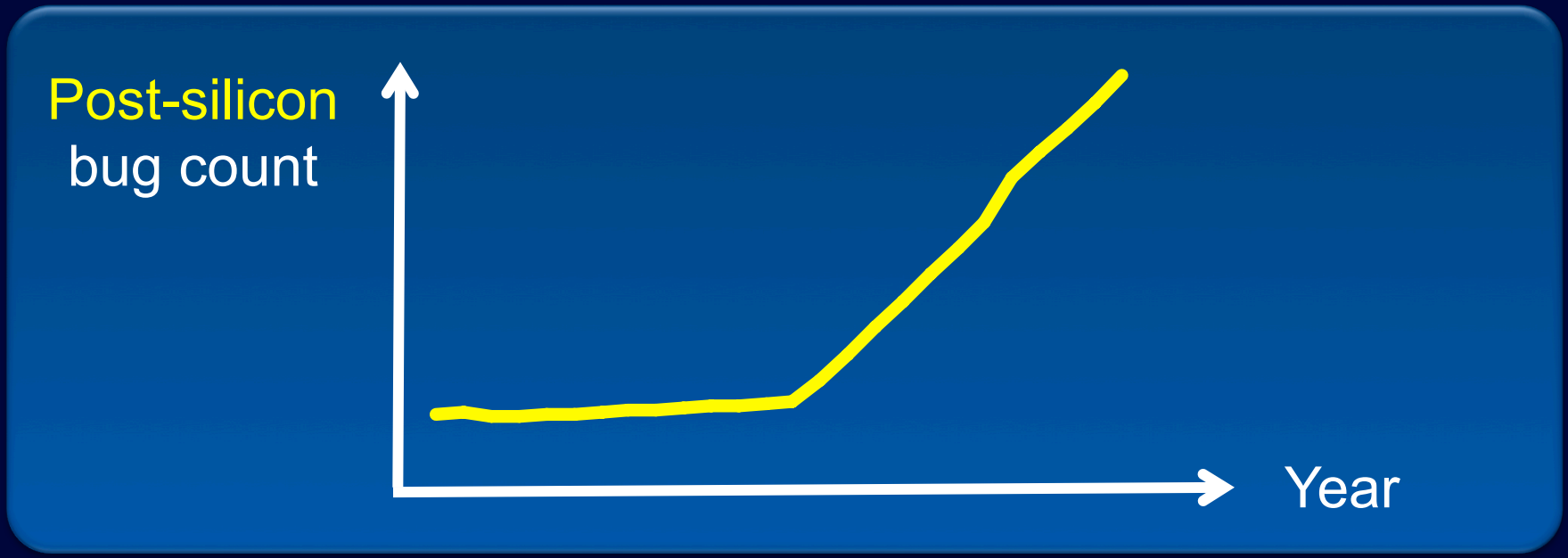
QED & Symbolic QED

Subhasish Mitra

Department of EE & Department of CS
Stanford University

Acknowledgment: Students & Collaborators

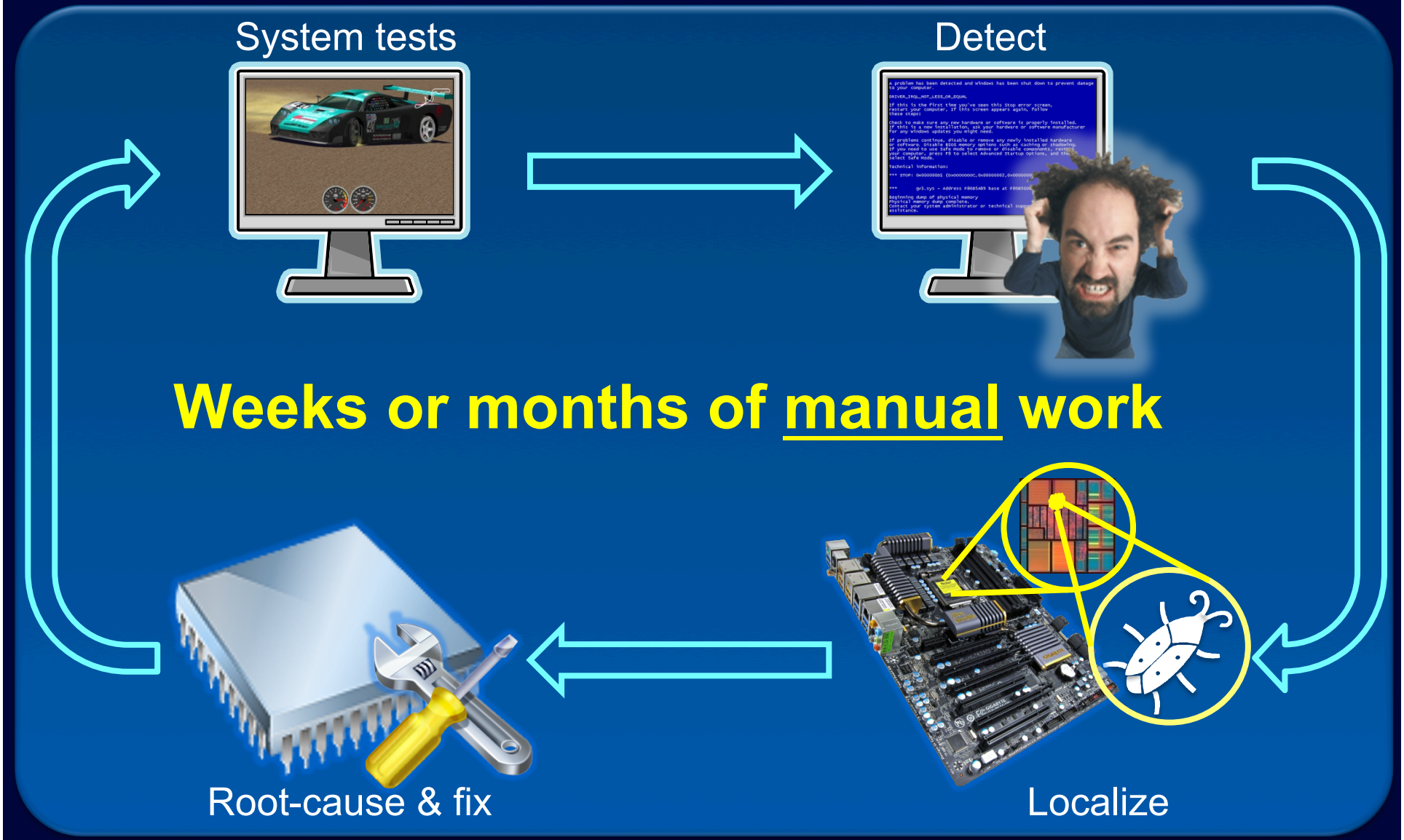
Pre-Silicon Verification Inadequate



New architectural features limited by validation

Source: Intel

Post-Silicon Validation Difficult



Scalability Barriers

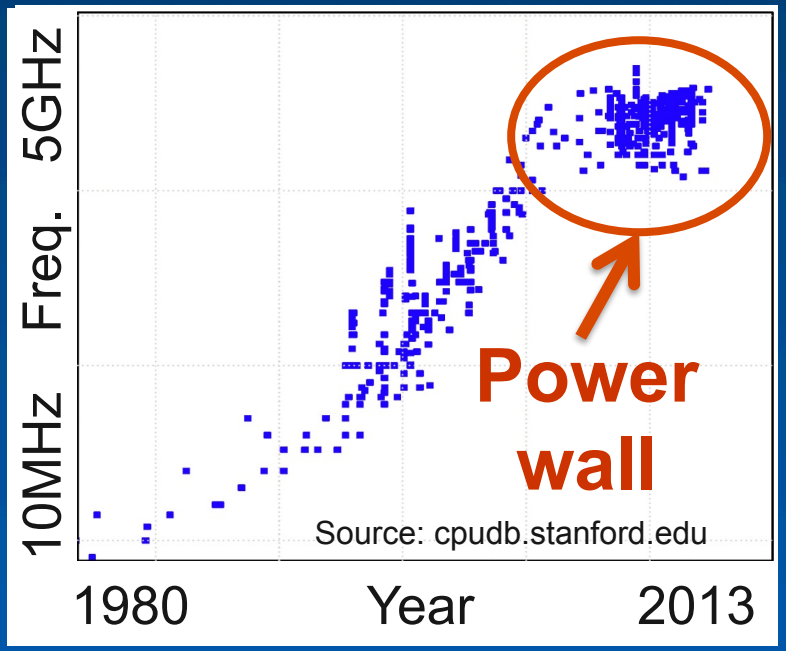
- System-level failure reproduction
- Full system simulation

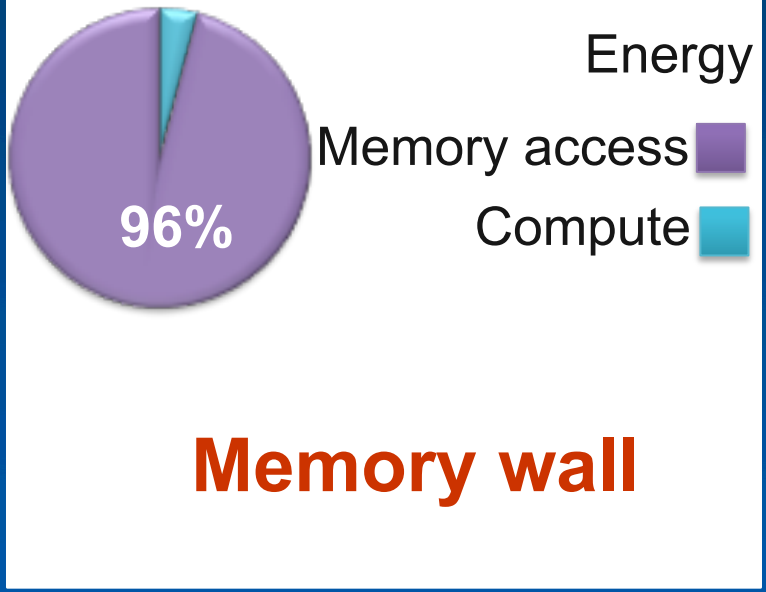
J. Stinson (ex-Intel)

Post-silicon costs rising faster than design cost

It's Only Getting Worse

Many walls simultaneously



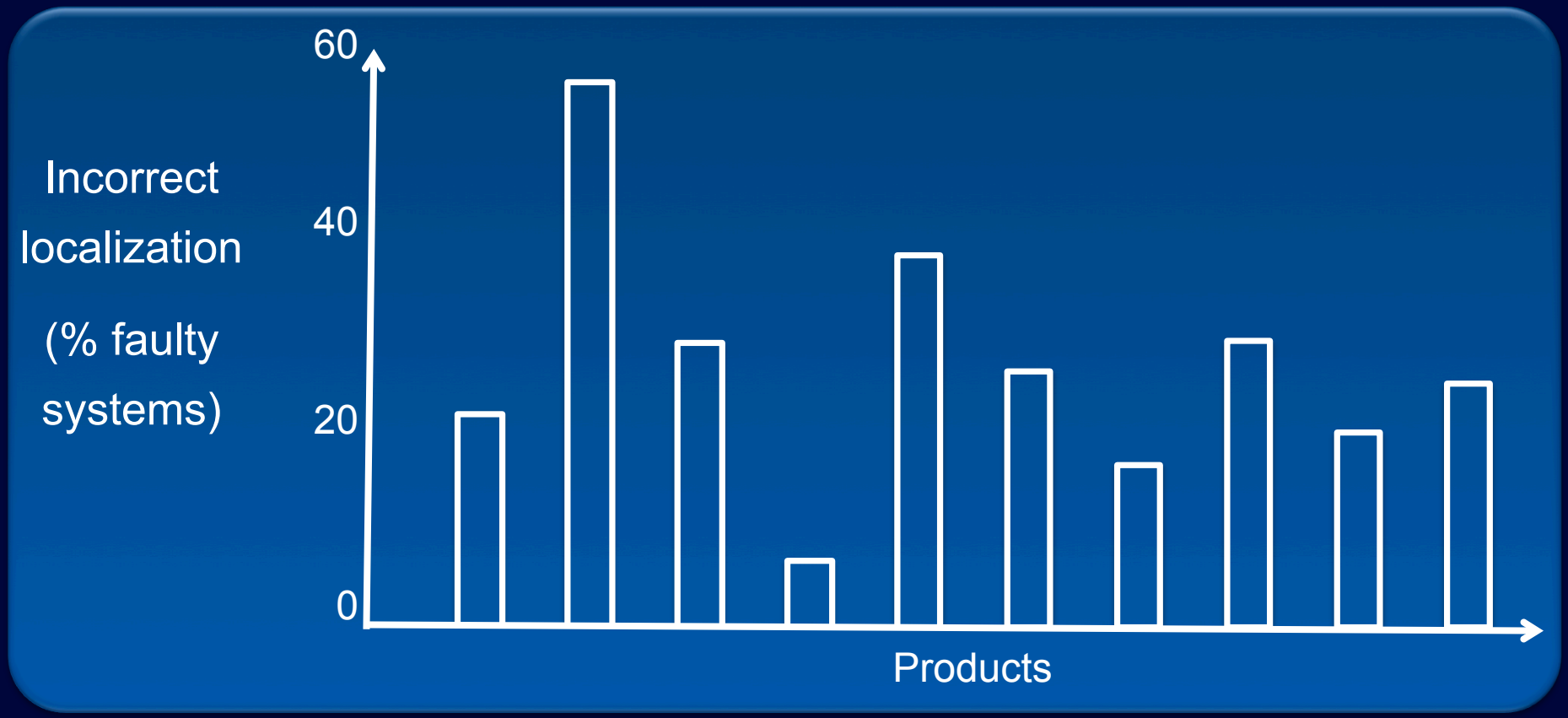


Also: interconnect wall, cooling wall, resilience wall, …

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

Collaborators: C. Barrett (Stanford), D. Chen (UIUC), W. Kunz, D. Stoffel (TU Kaiserslautern)

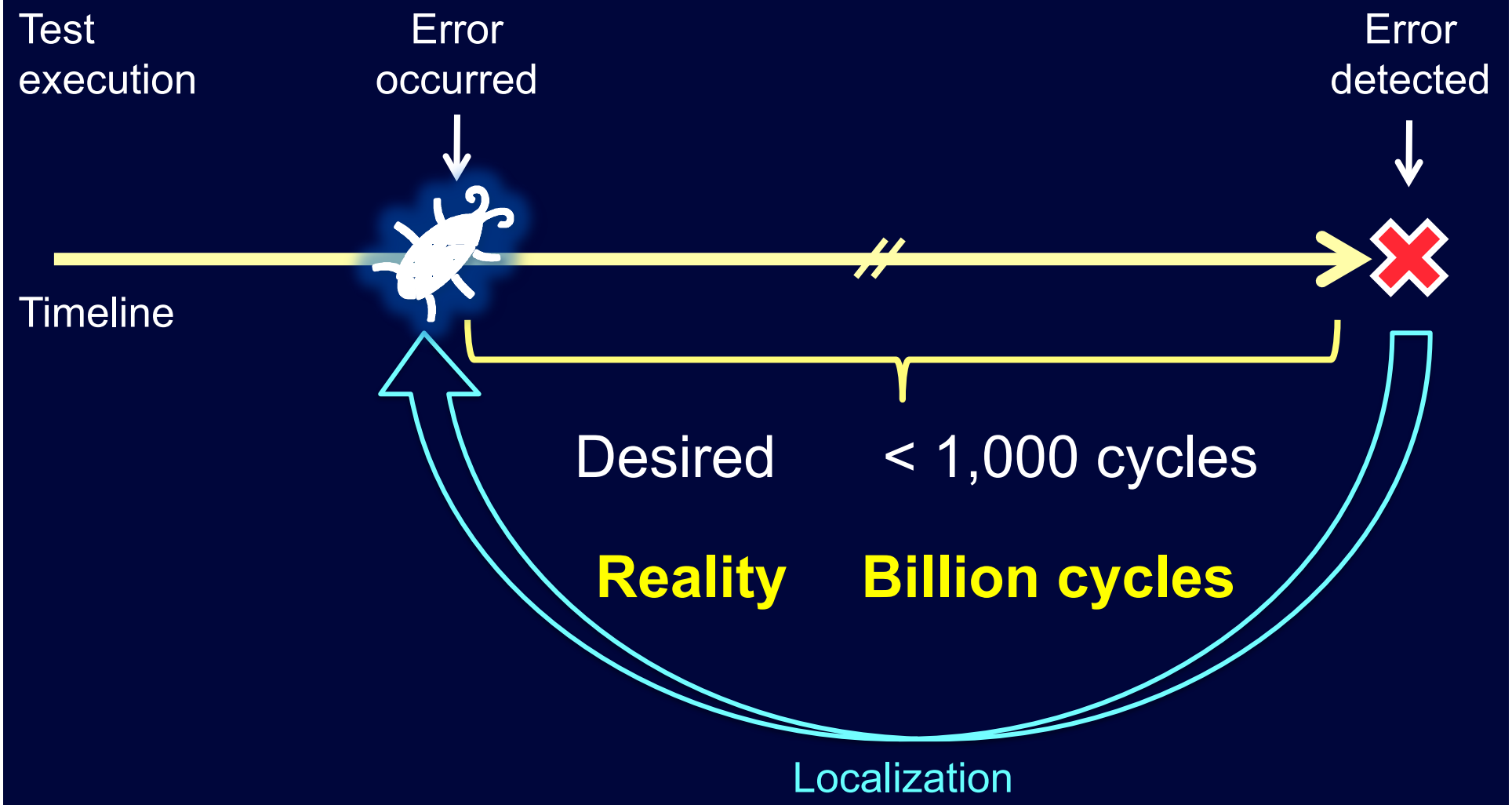
QED & Symbolic QED Results

- Post-silicon validation & debug

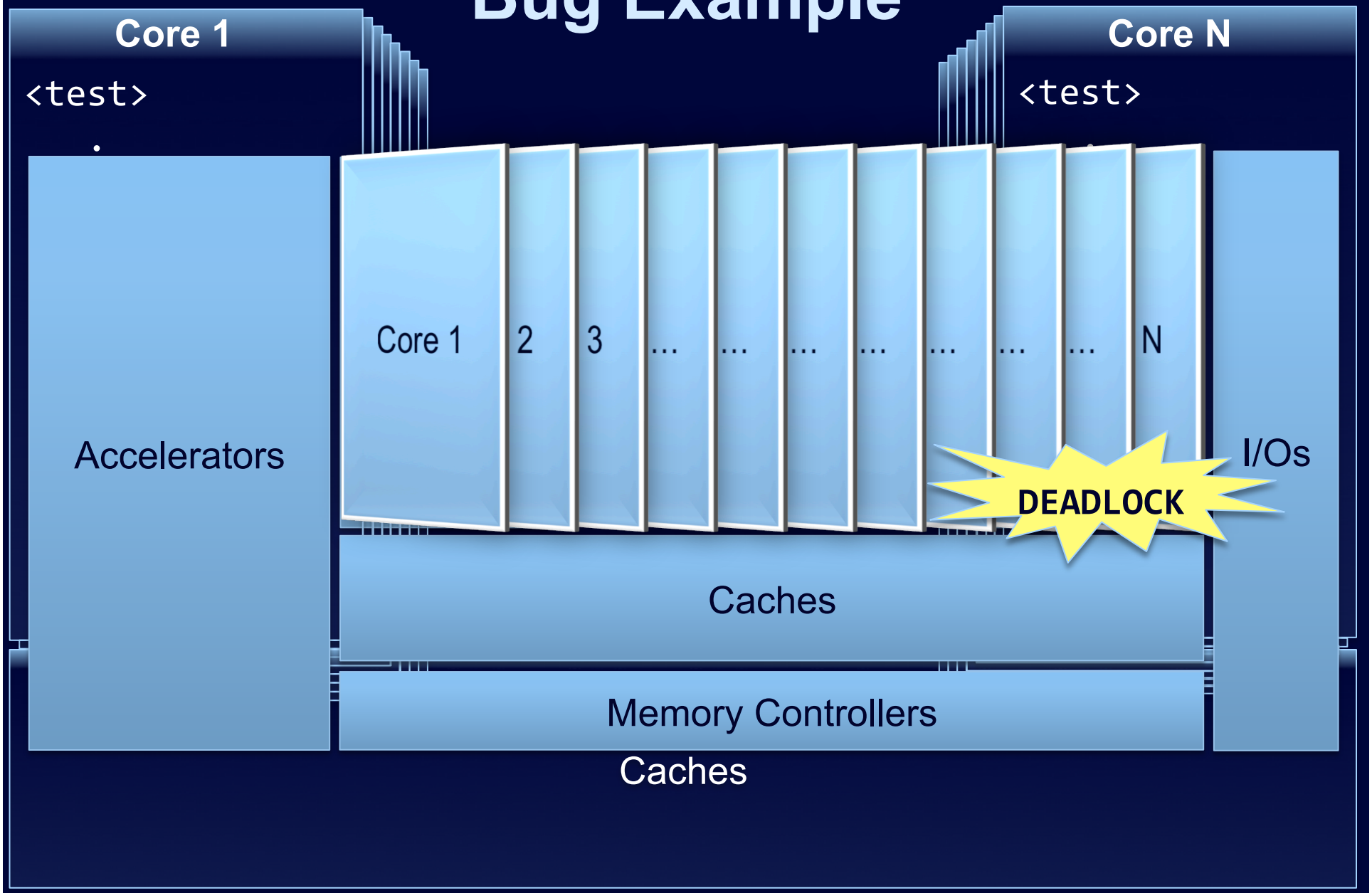
20 flip-flops (out of 1 Million) 9 instructions (out of billions)
Automatic
Few seconds to 9 hours
Billion-transistor designs

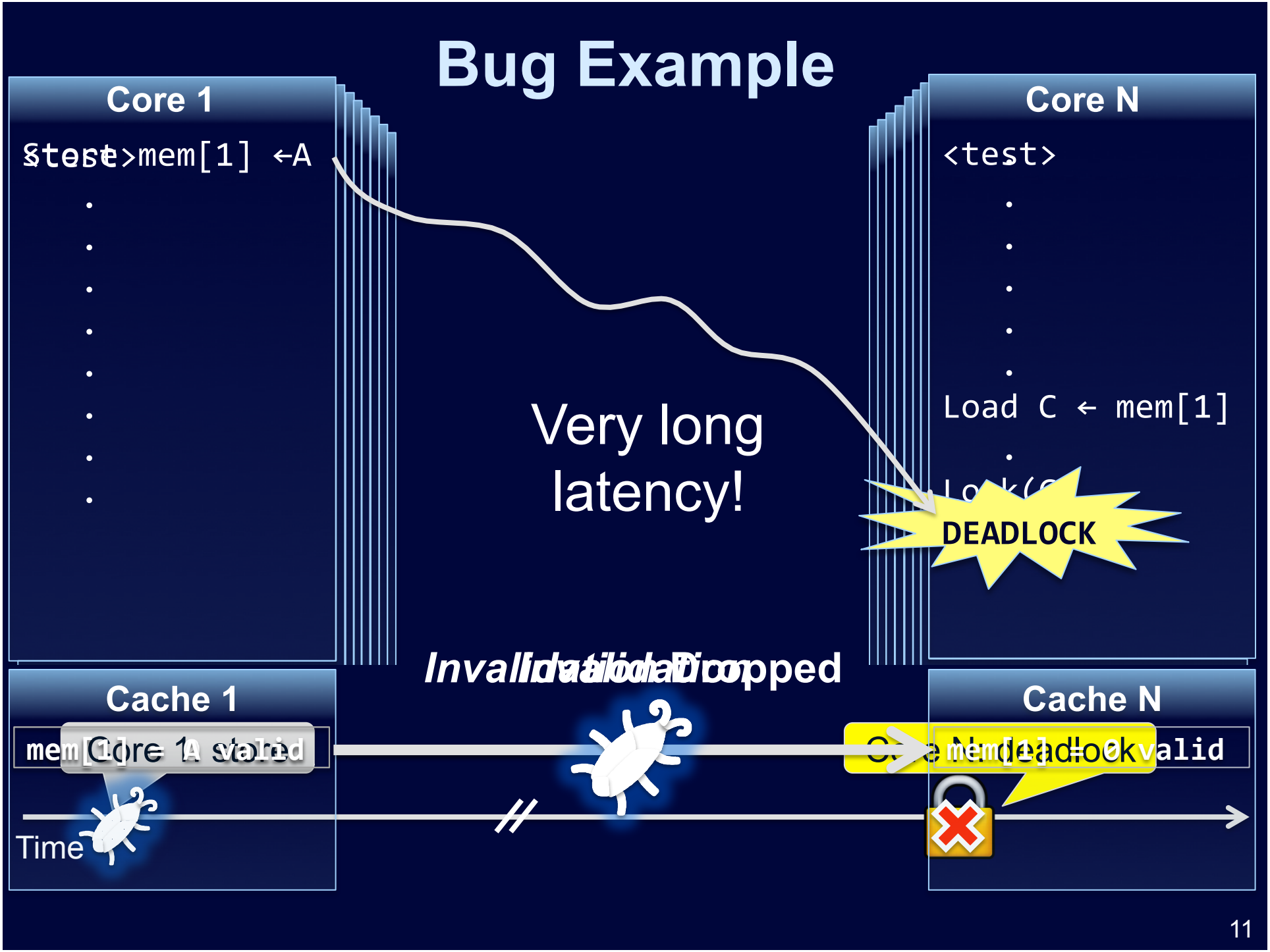
Collaborators: C. Barrett (Stanford), D. Chen (UIUC), W. Kunz, D. Stoffel (TU Kaiserslautern)

Error Detection Latency



Bug Example


Core 1
<test>
Core N
<test>
Core 1
2
3
...
...
...
...
...
...
...
N
Accelerators
I/Os
DEADLOCK
Caches
Memory Controllers
Caches


Bug Example
Core 1
<test>mem[1] ←A
Core N
<test>
Load C ← mem[1]
Very long latency!
DEADLOCK
Cache 1
mem[1] = A valid
Cache N
mem[1] = ? valid
Time
11

Existing Techniques Inadequate

- Very long error detection latencies

Deadlock detection

Design-specific assertions

Self-checking tests

Store readback tests

Checkpointing

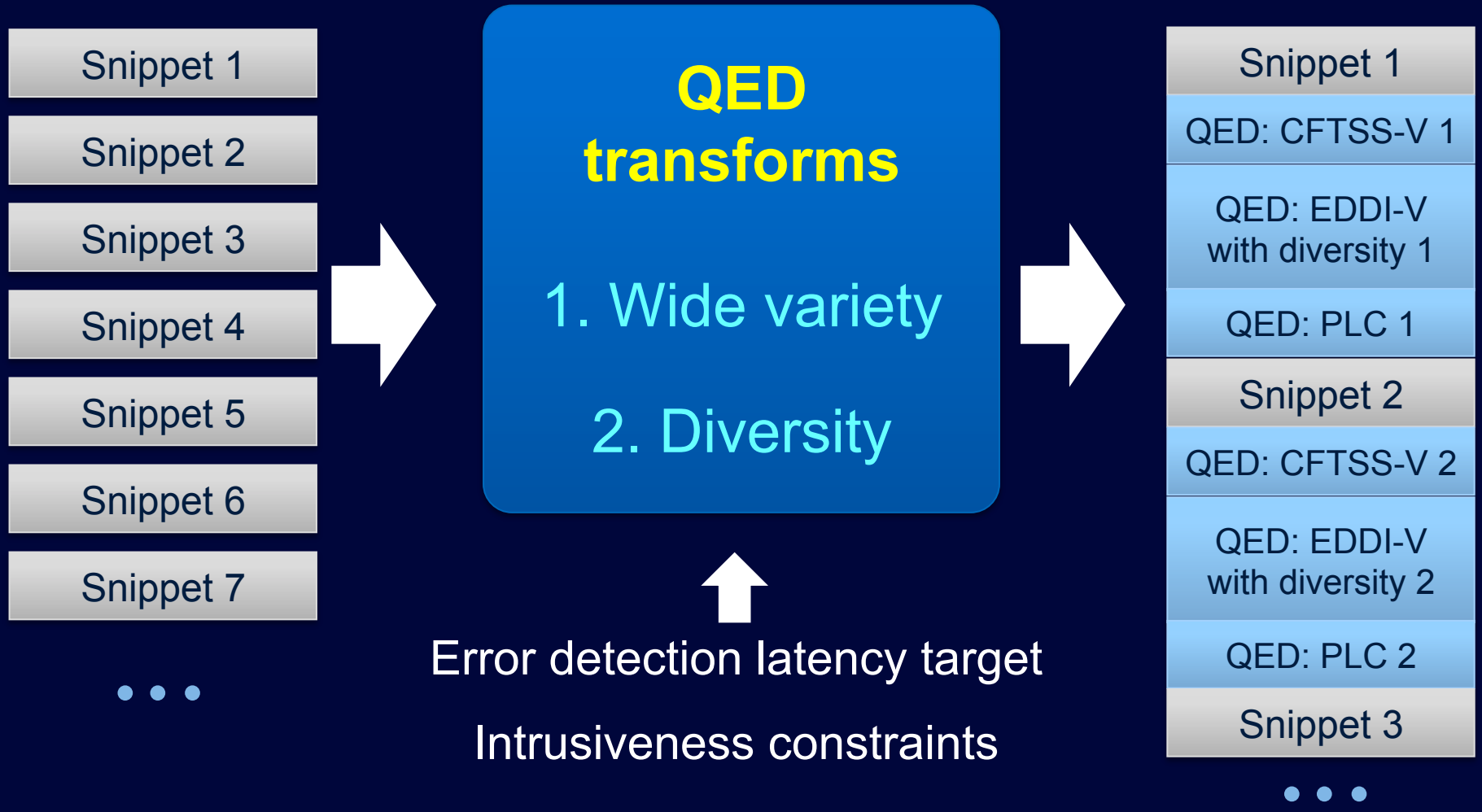
etc…

Quick Error Detection

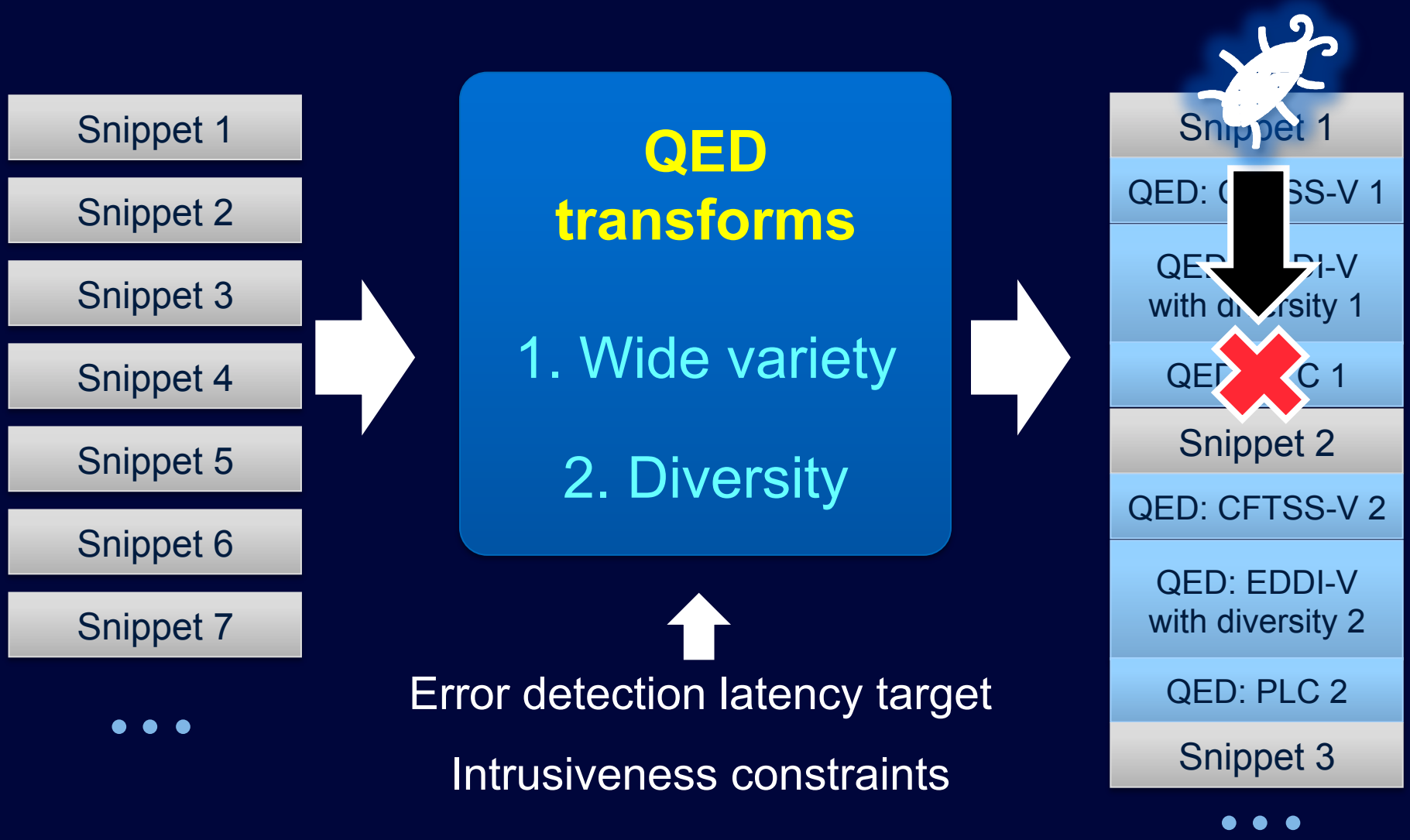
- Error detection latency: guaranteed short
- Coverage: improved
- Software-only: readily applicable

[Lin IEEE TCAD 14]

Quick Error Detection

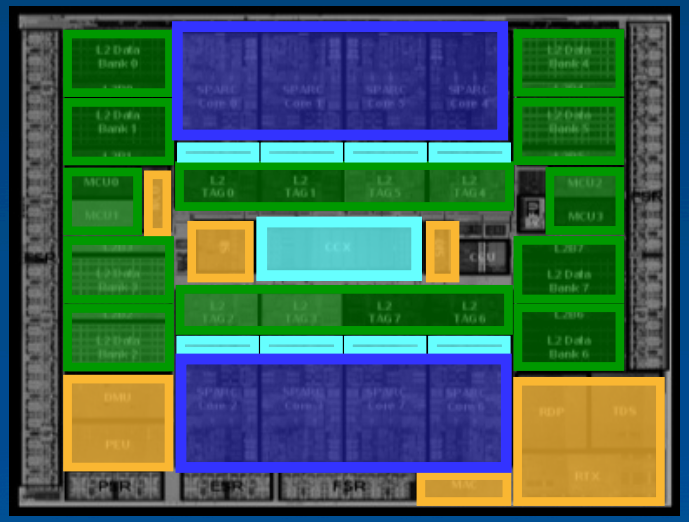


Quick Error Detection



QED Transforms

System **o**n **C**hip



Processor cores

Uncore, accelerators

EDDI-V

CFTSS-V

CFCSS-V

PLC

Fast QED

Hybrid QED

QED Example: Duplicate & Check

Validation program

```
void matgen (REAL a[], int lda, int n, REAL b[], REAL *norma)
{
    int init, i, j;
    init = 1325;
    *norma = 0.0;
    for (j = 0; j < n; j++) {
        for (i = 0; i < n; i++) {
            init = 3125*init % 0x1000000001;
            a[lda*j+i] = (init - 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,kp1,l,nm1;

    /*     gaussian elimination with partial pivoting      */

    *info = 0;
    nm1 = n - 1;
    if (nm1 >=  0) {
        for (k = 0; k < nm1; k++) {
            kp1 = 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[lda*k+l] != 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];
                dscal(n-(k+1),t,&a[lda*k+k+1],1);

                /* row elimination with column indexing */
                for (j = kp1; j < n; j++) {
                    t = a[lda*j+l];
                    if (l != k) {
                        a[lda*j+l] = 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;
            }
        }
    }

    ipvt[n-1] = 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,l,nm1;

    nm1 = n - 1;
    if (job == 0) {
        /* job = 0 , solve  a * x = b
           first solve  l*y = b         */
        if (nm1 >= 1) {
            for (k = 0; k < nm1; 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],1,&b[k+1],1 );
            }
        }

        /* now solve  u*x = y */
        for (kb = 0; kb < n; kb++) {
            k = n - (kb + 1);
            b[k] = b[k]/a[lda*k+k];
            t = -b[k];
            daxpy(k,t,&a[lda*k+0],1,&b[0],1 );
        }
    }
    else {
```

Trace

• • •

```
R1  ← R1  + 5
R2  ← R2  − R1
R3  ← R1  * R2
```

Very Long!

• • •

```
Check end_result
```

QED Example: Duplicate & Check

Validation program

```
void matgen (REAL a[], int lda, int n, REAL b[], REAL *norma)
{
    int init, i, j;
    init = 1325;
    *norma = 0.0;
    for (j = 0; j < n; j++) {
        for (i = 0; i < n; i++) {
            init = 3125*init % 0x1000000001;
            a[lda*j+i] = (init - 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,kp1,l,nm1;

    /*     gaussian elimination with partial pivoting      */

    *info = 0;
    nm1 = n - 1;
    if (nm1 >=  0) {
        for (k = 0; k < nm1; k++) {
            kp1 = 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[lda*k+l] != 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];
                dscal(n-(k+1),t,&a[lda*k+k+1],1);

                /* row elimination with column indexing */
                for (j = kp1; j < n; j++) {
                    t = a[lda*j+l];
                    if (l != k) {
                        a[lda*j+l] = 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;
            }
        }
    }

    ipvt[n-1] = 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,l,nm1;

    nm1 = n - 1;
    if (job == 0) {
        /* job = 0 , solve  a * x = b
           first solve  l*y = b         */
        if (nm1 >= 1) {
            for (k = 0; k < nm1; 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],1,&b[k+1],1 );
            }
        }

        /* now solve  u*x = y */
        for (kb = 0; kb < n; kb++) {
            k = n - (kb + 1);
            b[k] = b[k]/a[lda*k+k];
            t = -b[k];
            daxpy(k,t,&a[lda*k+0],1,&b[0],1 );
        }
    }
    else {
```

QED Trace

```
R16 ← R1; R18 ← R2; R19 ← R3
```

• • •

```
R1  ← R1  + 5
R16 ← R16 + 5
R1 == R16
R2  ← R2  - R1
R18 ← R18 - R16
R2 == R18
R3  ← R1  * R2
R19 ← R16 * R18
R3 == R19
```

• • •

```
Check end_result
```

Quick!

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 (j = 0; j < n; j++) {
        for (i = 0; i < n; i++) {
            init = 3125*init % 0x1000000001;
            a[lda*j+i] = (init - 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,kp1,l,nm1;

    /*     gaussian elimination with partial pivoting      */

    *info = 0;
    nm1 = n - 1;
    if (nm1 >=  0) {
        for (k = 0; k < nm1; k++) {
            kp1 = 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[lda*k+l] != 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];
                dscal(n-(k+1),t,&a[lda*k+k+1],1);

                /* row elimination with column indexing */
                for (j = kp1; j < n; j++) {
                    t = a[lda*j+l];
                    if (l != k) {
                        a[lda*j+l] = 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;
            }
        }
    }

    ipvt[n-1] = 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,l,nm1;

    nm1 = n - 1;
    if (job == 0) {
        /* job = 0 , solve  a * x = b
           first solve  l*y = b         */
        if (nm1 >= 1) {
            for (k = 0; k < nm1; 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],1,&b[k+1],1 );
            }
        }

        /* now solve  u*x = y */
        for (kb = 0; kb < n; kb++) {
            k = n - (kb + 1);
            b[k] = b[k]/a[lda*k+k];
            t = -b[k];
            daxpy(k,t,&a[lda*k+0],1,&b[0],1 );
        }
    }
    else {
```

Trace

• • •

R1 ← R1 + 1

R1

3	~~2~~

• • •

R1 ← 0

• • •

Check R1 == 0

Error masked!

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 (j = 0; j < n; j++) {
        for (i = 0; i < n; i++) {
            init = 3125*init % 0x1000000001;
            a[lda*j+i] = (init - 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,kp1,l,nm1;

    /*     gaussian elimination with partial pivoting      */

    *info = 0;
    nm1 = n - 1;
    if (nm1 >=  0) {
        for (k = 0; k < nm1; k++) {
            kp1 = 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[lda*k+l] != 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];
                dscal(n-(k+1),t,&a[lda*k+k+1],1);

                /* row elimination with column indexing */
                for (j = kp1; j < n; j++) {
                    t = a[lda*j+l];
                    if (l != k) {
                        a[lda*j+l] = 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;
            }
        }
    }

    ipvt[n-1] = 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,l,nm1;

    nm1 = n - 1;
    if (job == 0) {
        /* job = 0 , solve  a * x = b
           first solve  l*y = b         */
        if (nm1 >= 1) {
            for (k = 0; k < nm1; 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],1,&b[k+1],1 );
            }
        }

        /* now solve  u*x = y */
        for (kb = 0; kb < n; kb++) {
            k = n - (kb + 1);
            b[k] = b[k]/a[lda*k+k];
            t = -b[k];
            daxpy(k,t,&a[lda*k+0],1,&b[0],1 );
        }
    }
    else {
```

QED Trace

• • •

R1 ← R1 + 1

R18 ← R18 + 1

R1 == R18

• • •

R1 ← 0

• • •

Check R1 == 0

R1	R18
3 ~~2~~	2

Error masked!

No QED: bug escape

QED: quick detection

Diversity-Enhanced QED

Validation program

```
void matgen (REAL a[], int lda, int n, REAL b[], REAL *norma)
{
    int init, i, j;
    init = 1325;
    *norma = 0.0;
    for (j = 0; j < n; j++) {
        for (i = 0; i < n; i++) {
            init = 3125*init % 0x1000000001;
            a[lda*j+i] = (init - 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,kp1,l,nm1;

    /*     gaussian elimination with partial pivoting      */

    *info = 0;
    nm1 = n - 1;
    if (nm1 >=  0) {
        for (k = 0; k < nm1; k++) {
            kp1 = 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[lda*k+l] != 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];
                dscal(n-(k+1),t,&a[lda*k+k+1],1);

                /* row elimination with column indexing */
                for (j = kp1; j < n; j++) {
                    t = a[lda*j+l];
                    if (l != k) {
                        a[lda*j+l] = 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;
            }
        }
    }

    ipvt[n-1] = 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,l,nm1;

    nm1 = n - 1;
    if (job == 0) {
        /* job = 0 , solve  a * x = b
           first solve  l*y = b         */
        if (nm1 >= 1) {
            for (k = 0; k < nm1; 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],1,&b[k+1],1 );
            }
        }

        /* now solve  u*x = y */
        for (kb = 0; kb < n; kb++) {
            k = n - (kb + 1);
            b[k] = b[k]/a[lda*k+k];
            t = -b[k];
            daxpy(k,t,&a[lda*k+0],1,&b[0],1 );
        }
    }
    else {
```

QED Trace

a ← 1; a' ← a; b ← 1, b' ← b

R1 ← a + b

R18 ← 5a' + 5b'

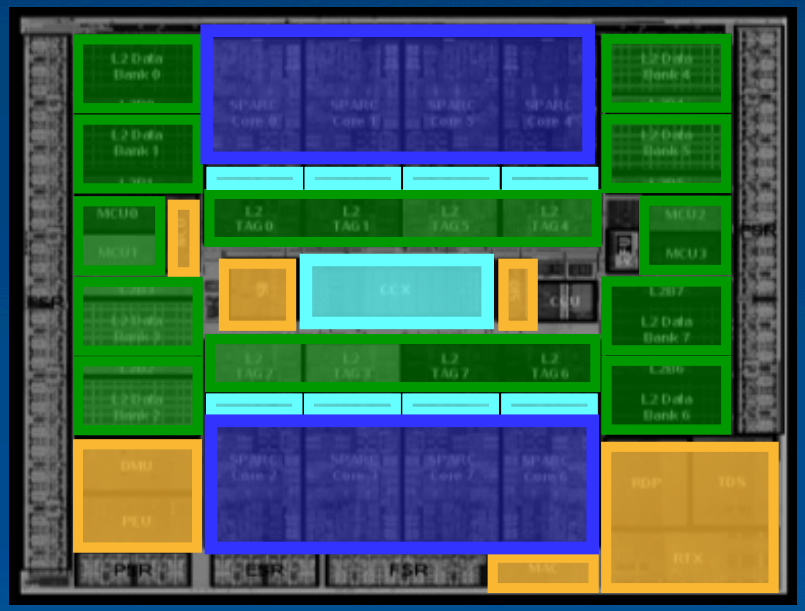
5×R1 == R18 ✗

6 ~~2~~	R1
14 ~~10~~	R18

Many diversity techniques

e.g., ED^{4}I [Oh IEEE Trans. Comp. 02]

QED Transforms



Processor cores

Uncore, accelerators

EDDI-V

CFTSS-V

CFCSS-V

PLC

Fast QED

Hybrid QED

Original Test

Core 1	Core 2		Core N
A = B * 2	E = F * G		
			I = E / 2
Store mem[1] ← C	H = D + E	• • •	
			Load J ← mem[7]
	Store mem[5] ← H		K = J + 1

QED: Duplicate & Check

Core 1

A’=A B’=B’ C’=C

A = B * 2

A’= B’* 2

Check(A==A’)

Store mem[1]← C

Store mem[1’]← C’

Core 2

D’=D E’=E F’=F

G’=G H’=H

E = F * G

E’= F’* G’

Check(E==E’)

H = D + E

H’= D’+ E’

Check(H==H’)

Store mem[5]← H

Store mem[5’]← H’

Core N

E’=E I’=E

J’=J K’=K

I = E / 2

I’= E’/ 2

Check(I==I’)

Load J ← mem[7]

Load J’← mem[7’]

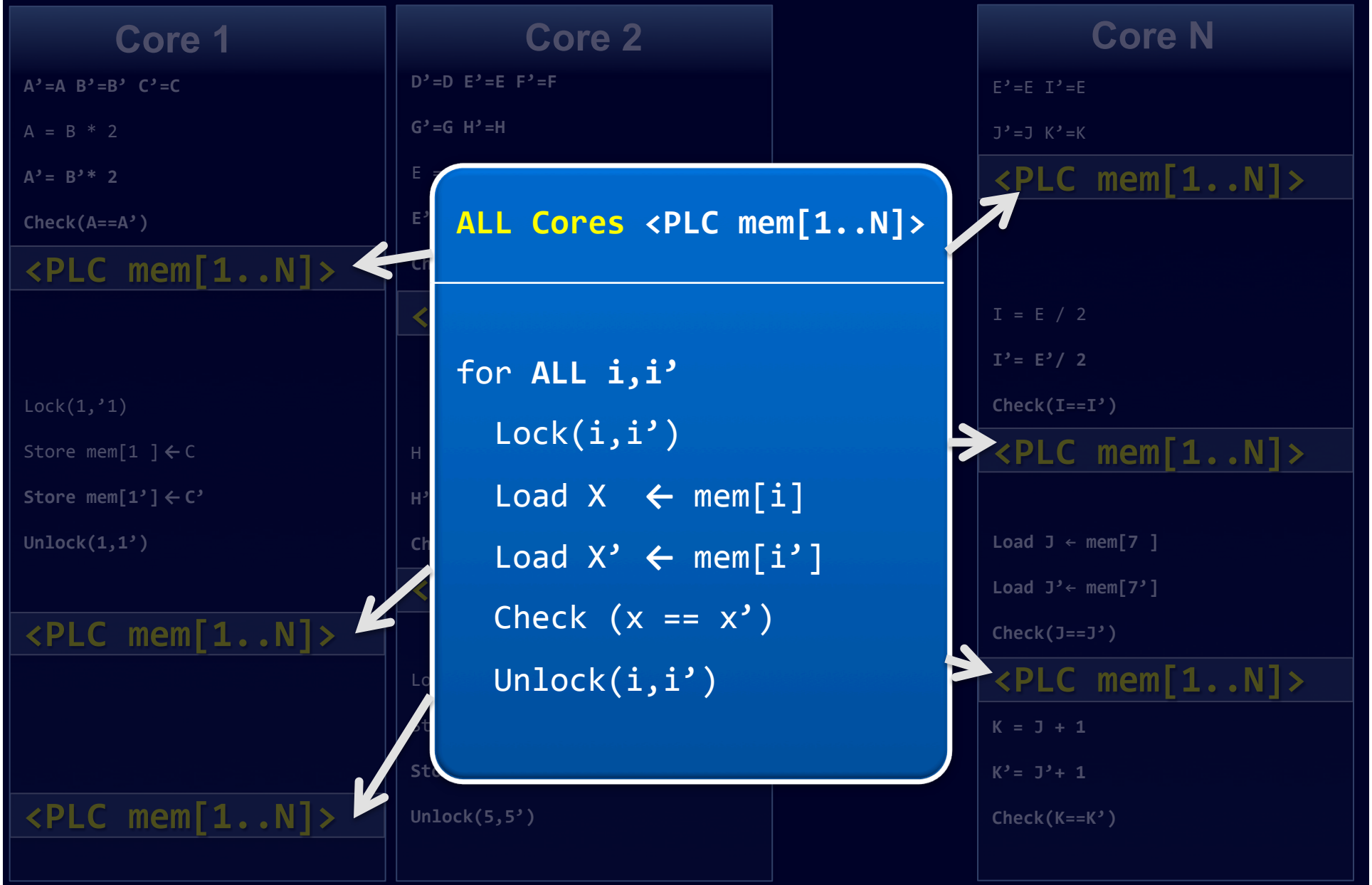
Check(J==J’)

K = J + 1

K’= J’+ 1

Check(K==K’)

QED: Proactive Load & Check



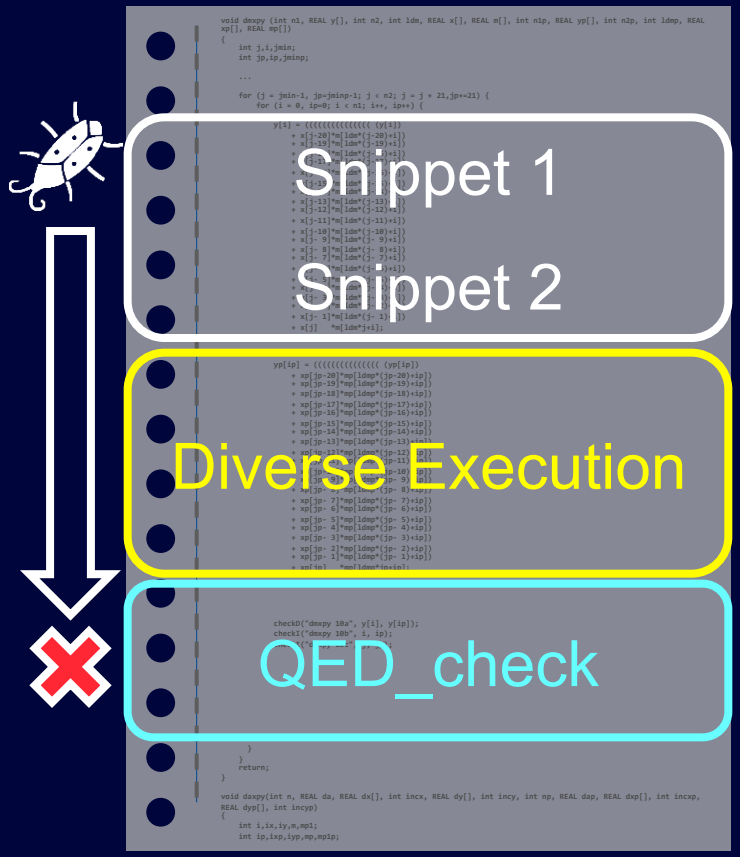
QED Coverage Considerations

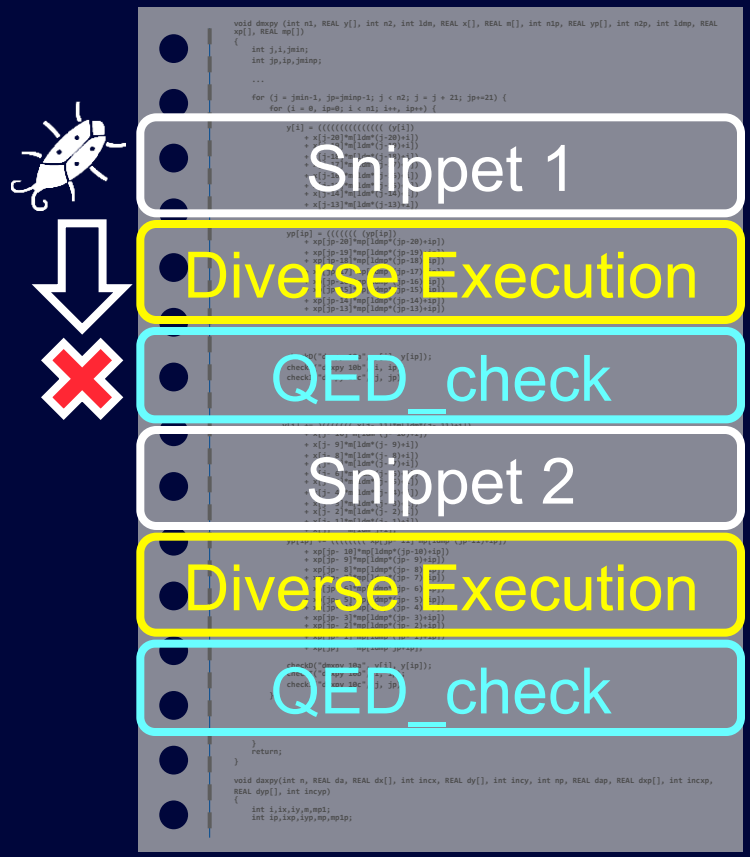
- Challenge
  - Intrusiveness: coverage impact ?
- Systematic solutions
  - QED family tests
  - Hardware-enhanced QED

Error Detection Latency vs. Intrusiveness

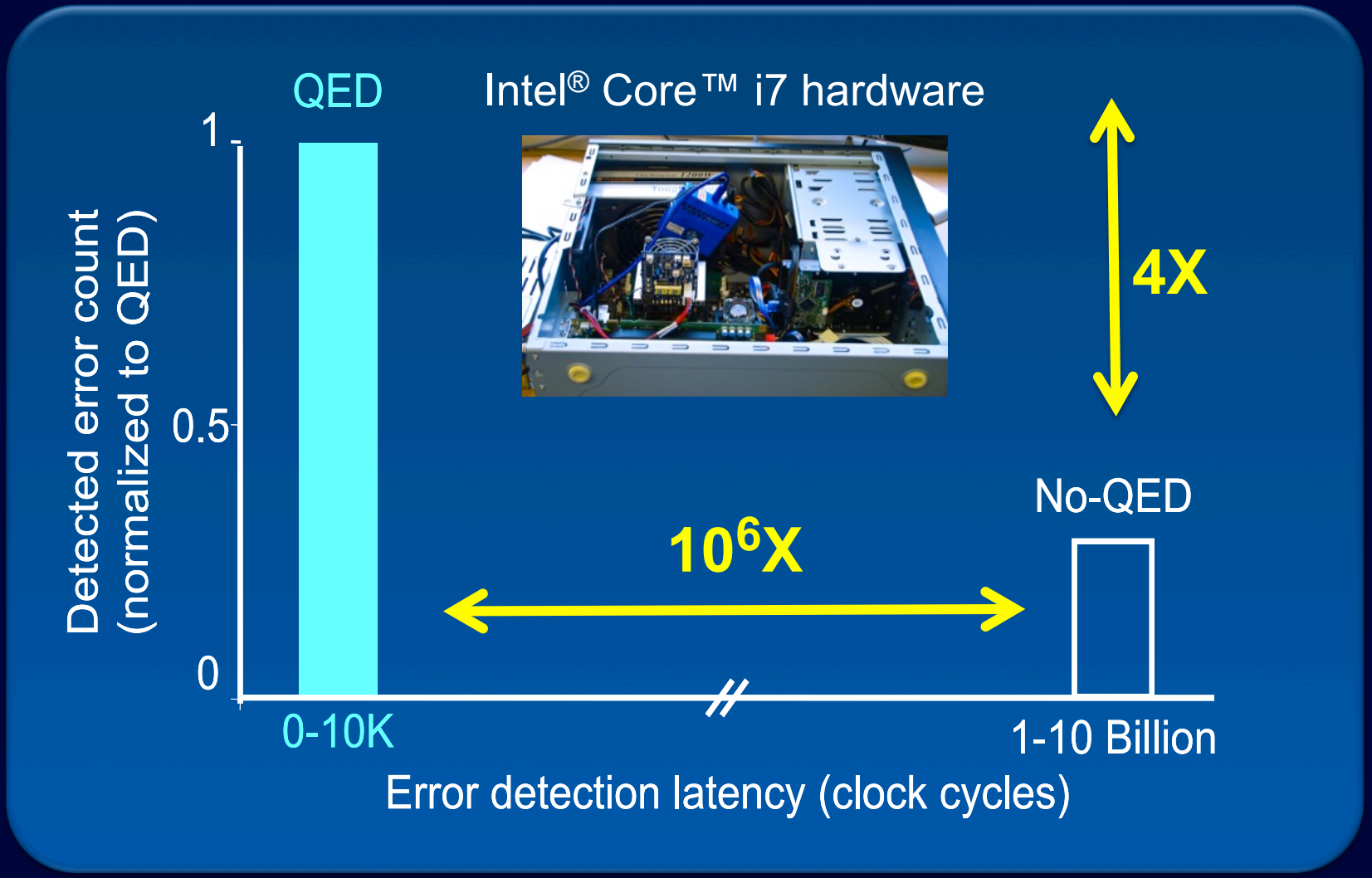
☺ Improved error detection latency

😐 Increased intrusiveness?





QED Effective for Electrical Bugs

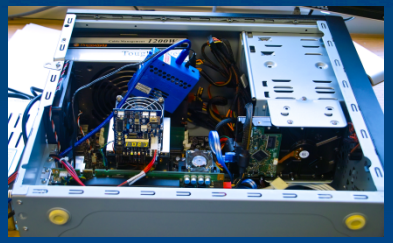


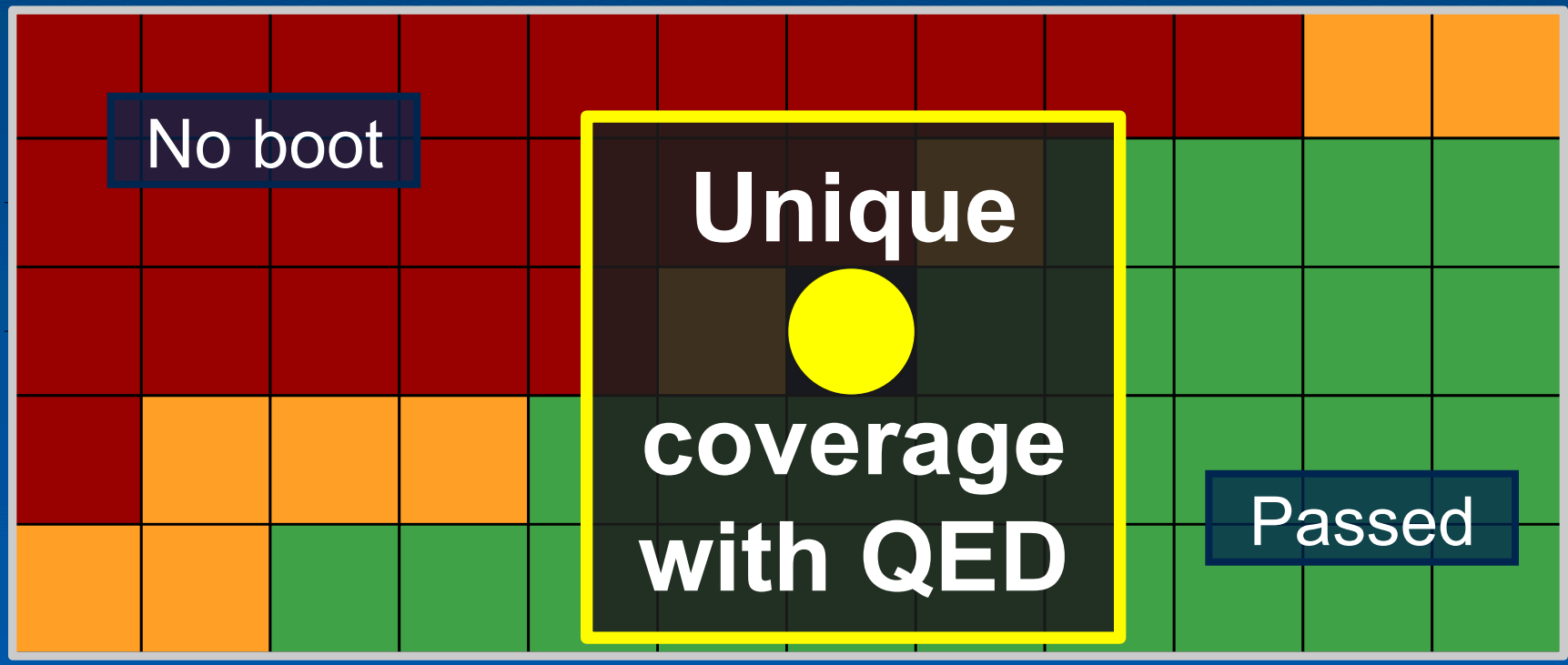
[Lin IEEE TCAD 14]

Intel® Core™ i7 Hardware

Error detected (QED and no-QED)

Error detected (QED only)





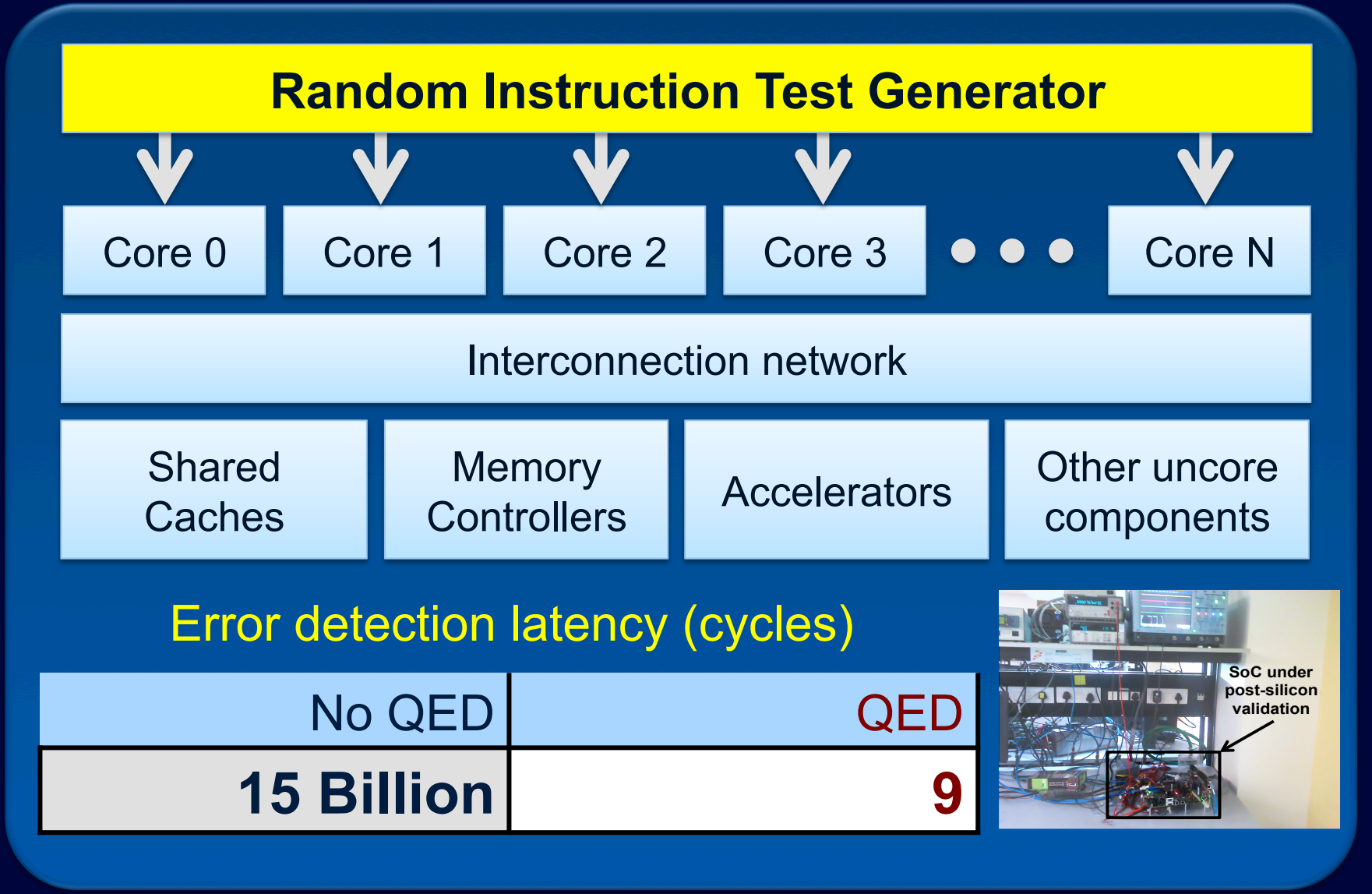
QED Effective for Logic Bugs

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

[Lin IEEE TCAD 14]

QED Logic Bugs

Freescale SoC hardware



Error detection latency (cycles)

No QED	QED
15 Billion	**9**

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

[Lin ITC 15, Singh IEEE D&T 16, Singh IEEE TCAD 17]

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 ?

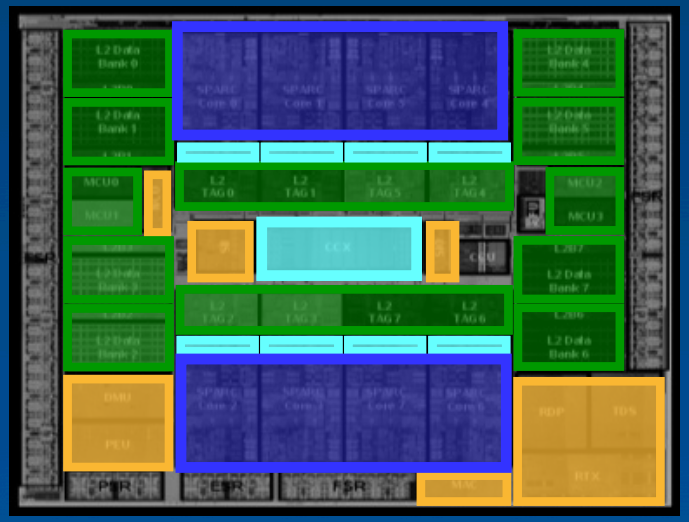
- Bugs not known *a priori*

2. Design size ?

- Big designs, cannot load

QED Transforms

System **o**n **C**hip



Processor cores

Uncore, accelerators

EDDI-V

CFTSS-V

CFCSS-V

PLC

Fast QED

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 ← 1

Ra' ← 2

CMP Ra == Ra'

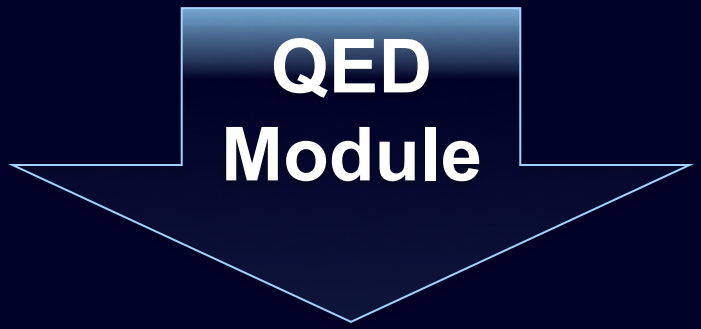
- False fail – not a bug

Solution: QED Module

Only during BMC, not in fabricated chip

Input
(Chosen by BMC)

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

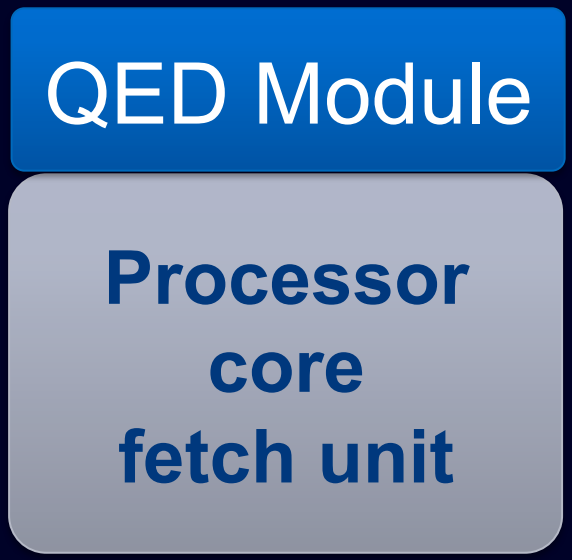


Output

```
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)

If property violated

Counter-example = bug trace

Initial State Important

- If unconstrained, BMC may choose any initial state

Ra = 1 Ra' = 2 // initial state

Ra = Ra + 1; Ra' = Ra' + 1; 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)

If property violated

Counter-example = bug trace

Symbolic QED Results

RIDECORE: RISC-V super-scalar core

3 new bugs: previously unknown

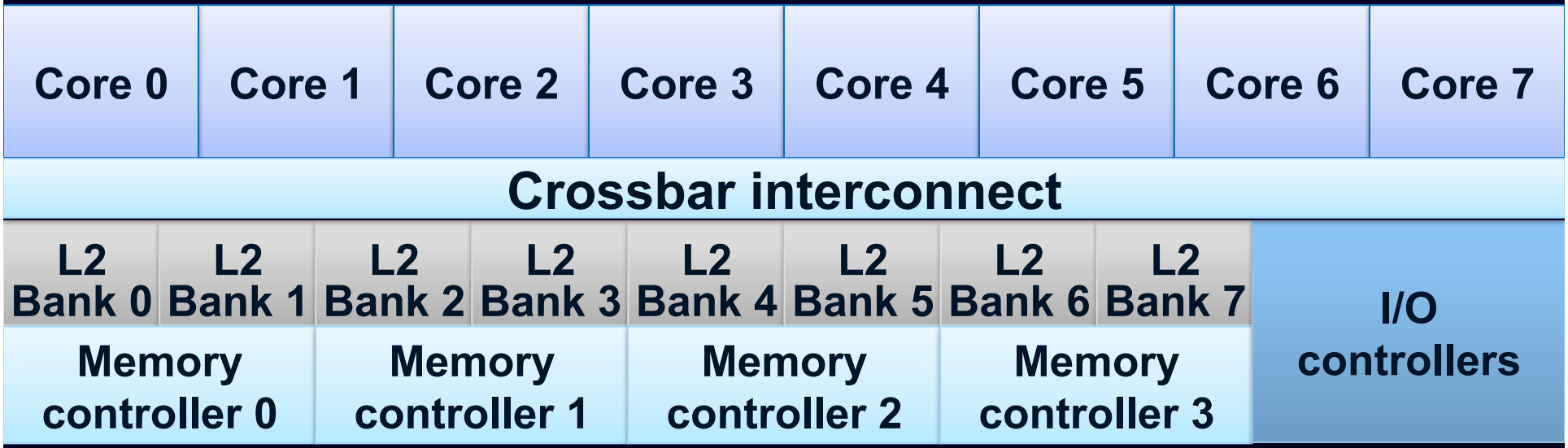
Difficult & complex corner cases
~ 1 minute Symbolic QED run
Automatic: no manual properties, constraints, etc.
Confirmed & fixed by designers (https://github.com/ridecore/ridecore/issues/4)

BUT…

- Big designs ?

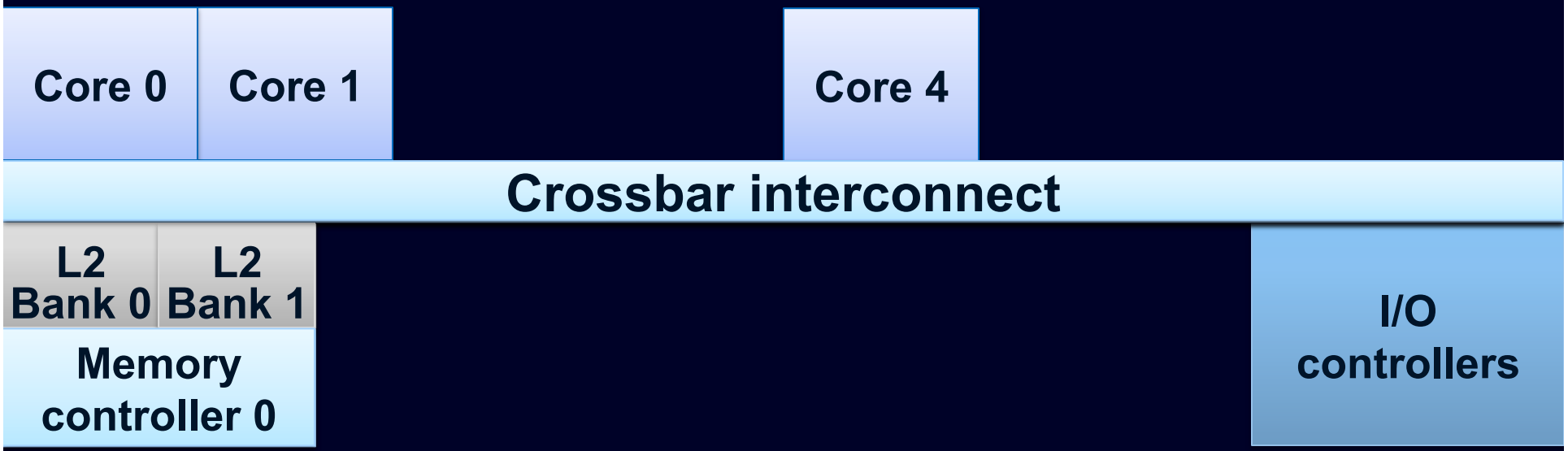
- **Solution: QED checks compositional**
  - Preserved across partial instances
  - Not design- / implementation-specific

Solution: Partial Instantiation

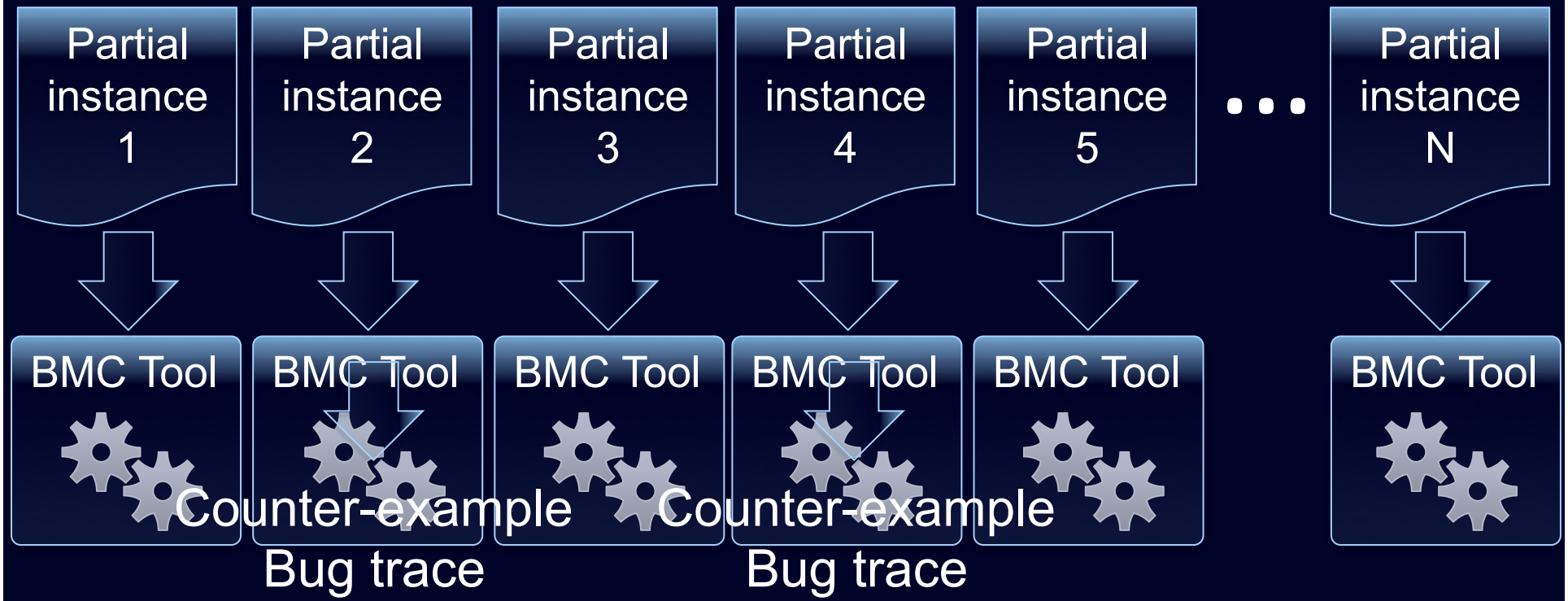


Solution: Partial Instantiation

- 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**

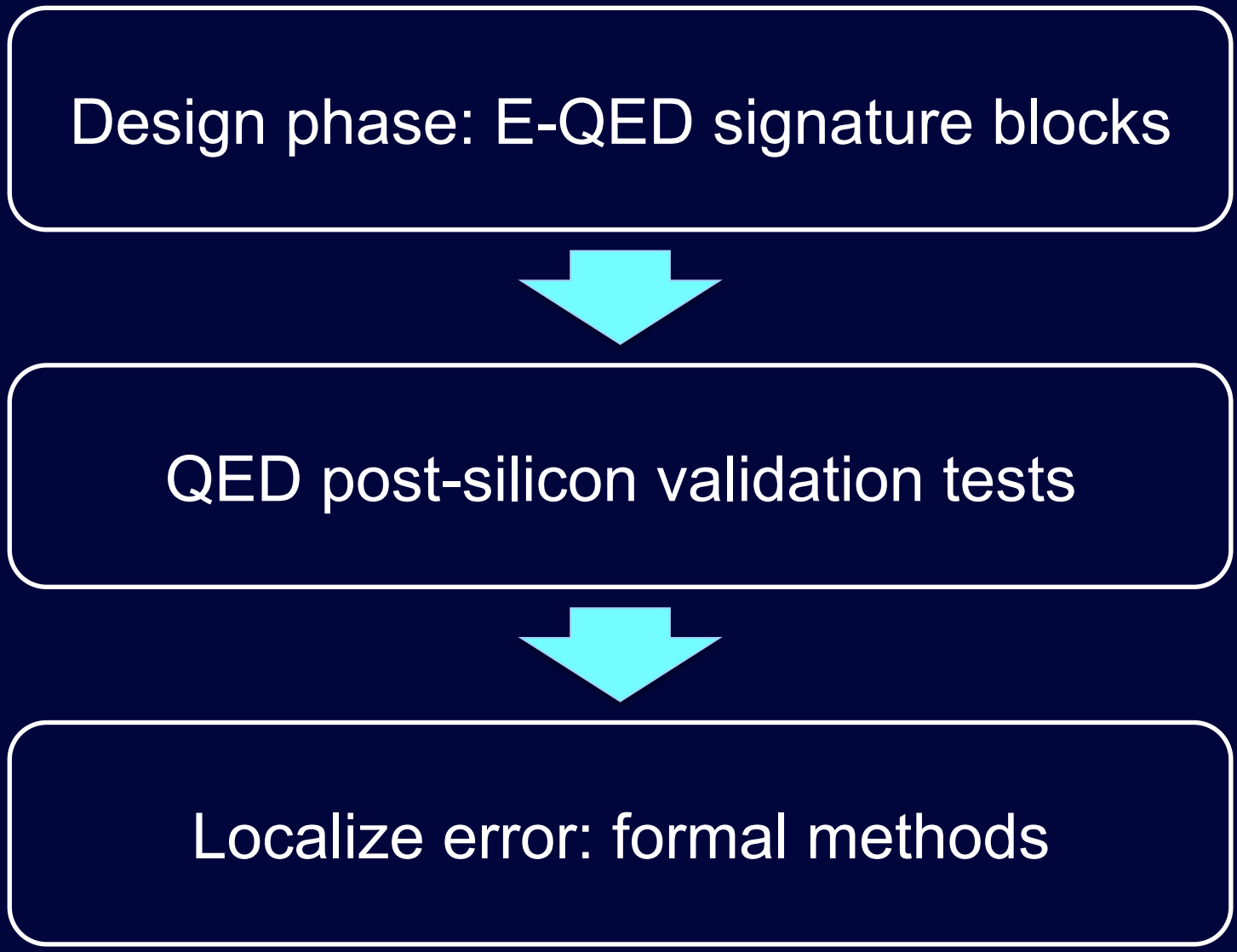
OpenSPARC T2 SoC (500M transistors): difficult bugs inserted

E-QED

- **Electrical bugs: post-silicon debug**

[Singh CAV 17]

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)

[Singh CAV 17]

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