QED & Symbolic QED

Pre-silicon Verification, Post-silicon Validation, Industrial Results

Subhasish Mitra, Eshan Singh

K Devarajegowda

Department of EE & CS
Stanford University

Infineon
Technologies AG

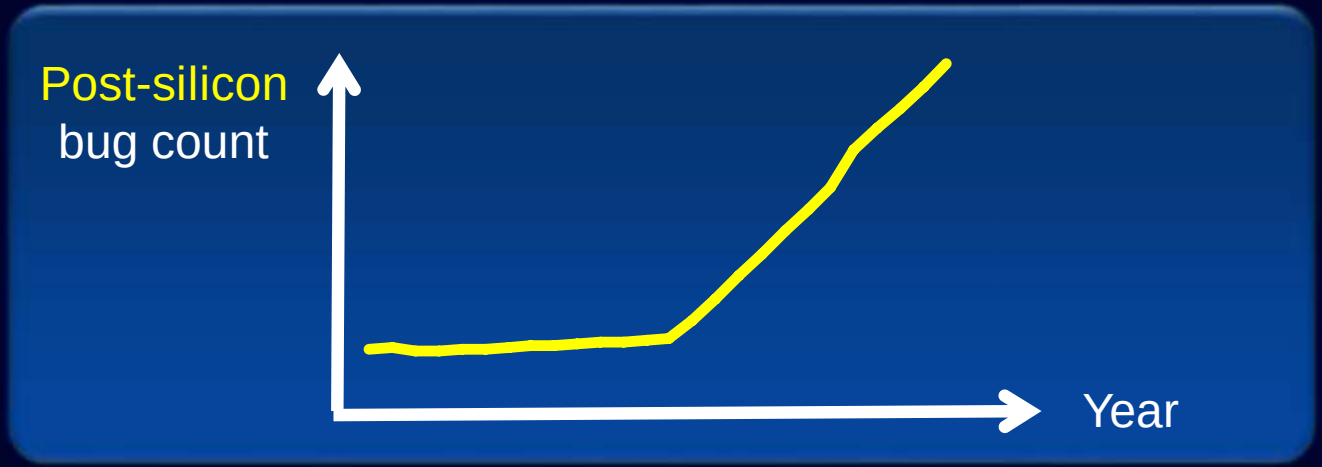
Students, Sponsors, Collaborators

Outline

- Introduction
- Infineon Case Study
- QED
- Symbolic QED
- Demos

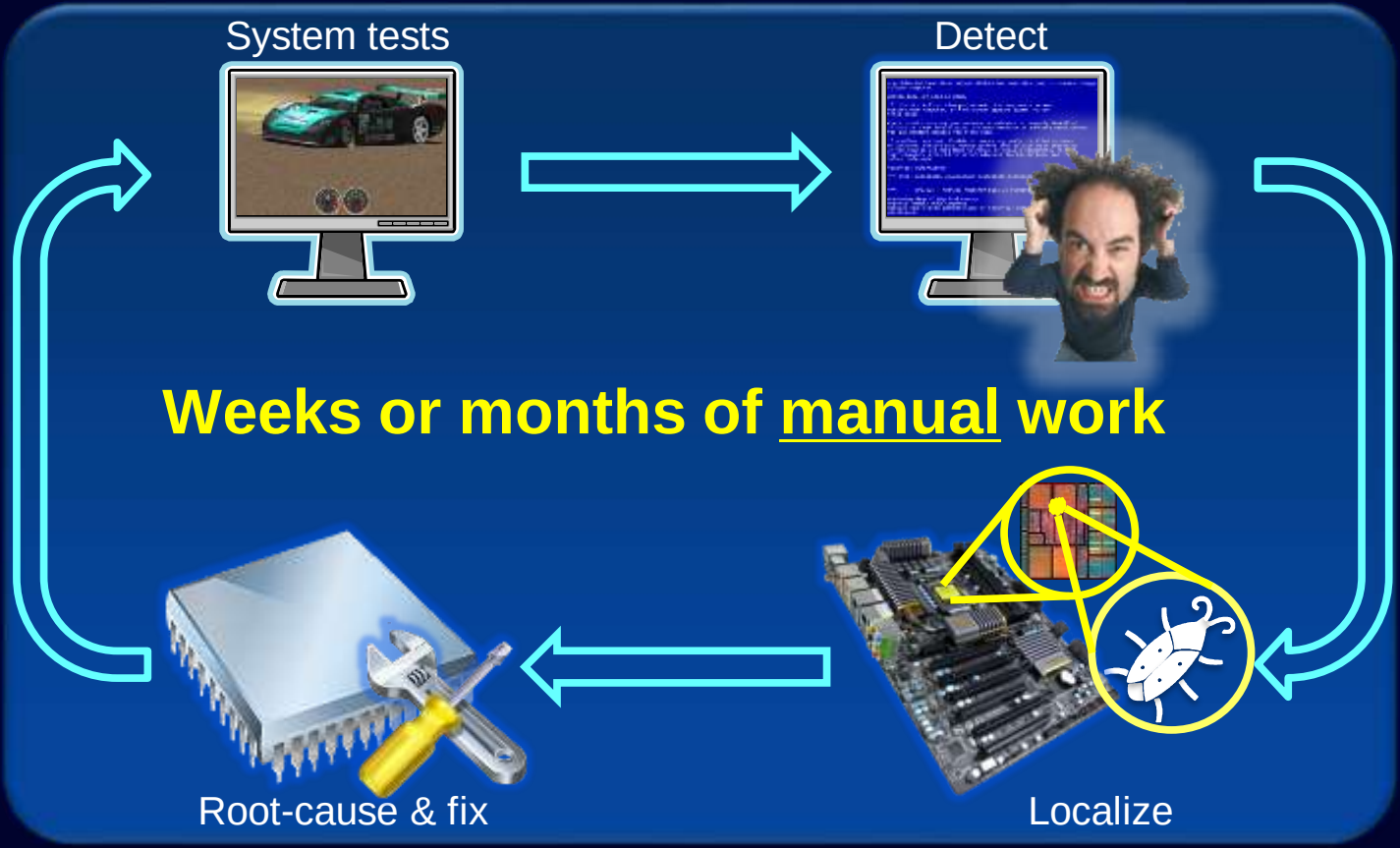
Pre-Silicon Verification Inadequate

It's only getting worse: custom hardware



Source: Intel

Post-Silicon Validation Difficult



Scalability Barriers

- System-level failure reproduction
- Full system simulation

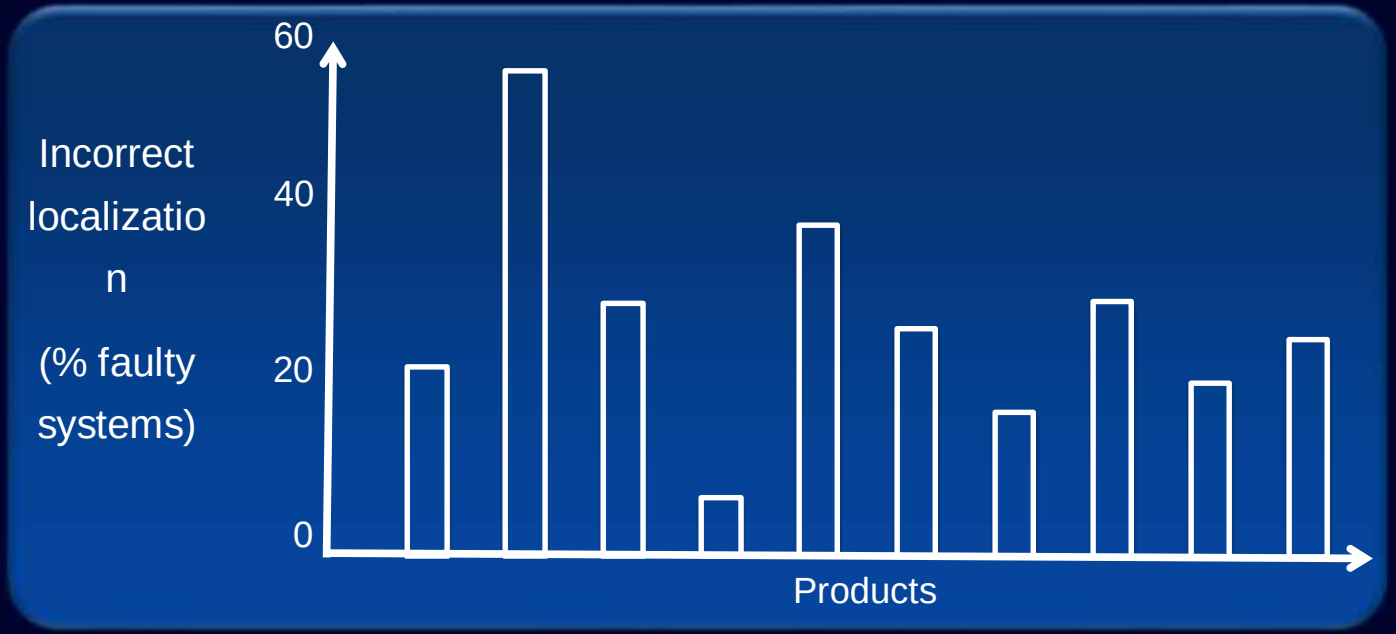
J. Stinson (ex-Intel)

Post-silicon costs rising faster than design cost

Bigger Obstacles at System Level

1. Localize faulty IC 2. No Trouble Found

Incorrect localization: up to 60%



Z. Conroy, Cisco

QED & Symbolic QED Results

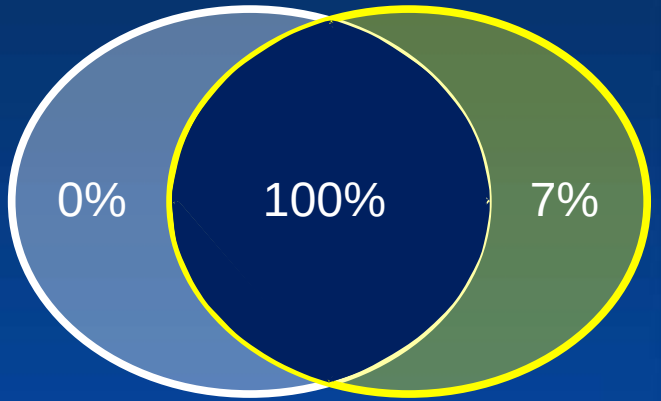
- Pre-silicon bugs

Few minutes to 7 hours
No manual properties
Billion-transistor scale designs: Processor cores, uncore, accelerators

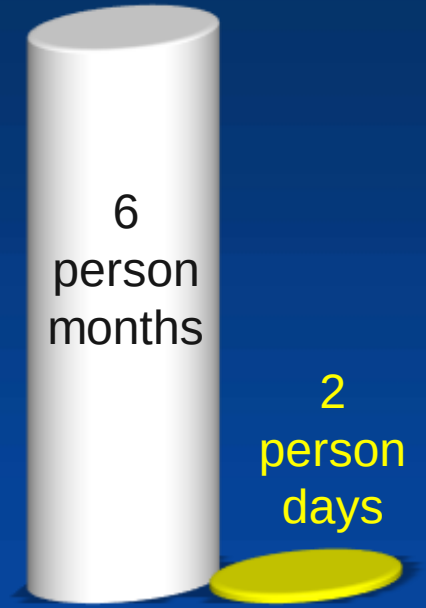
Symbolic QED: Infineon Study

- Several automotive microcontroller cores

All (known) bugs + more



***60*× productivity**



[Singh DATE 19]

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 scale designs

More Opportunities

- Hardware security
  - Derive new vulnerabilities
    - Beyond Spectre, Meltdown
- Firmware
- Large-scale systems

[Fadiheh DATE 19]

Outline

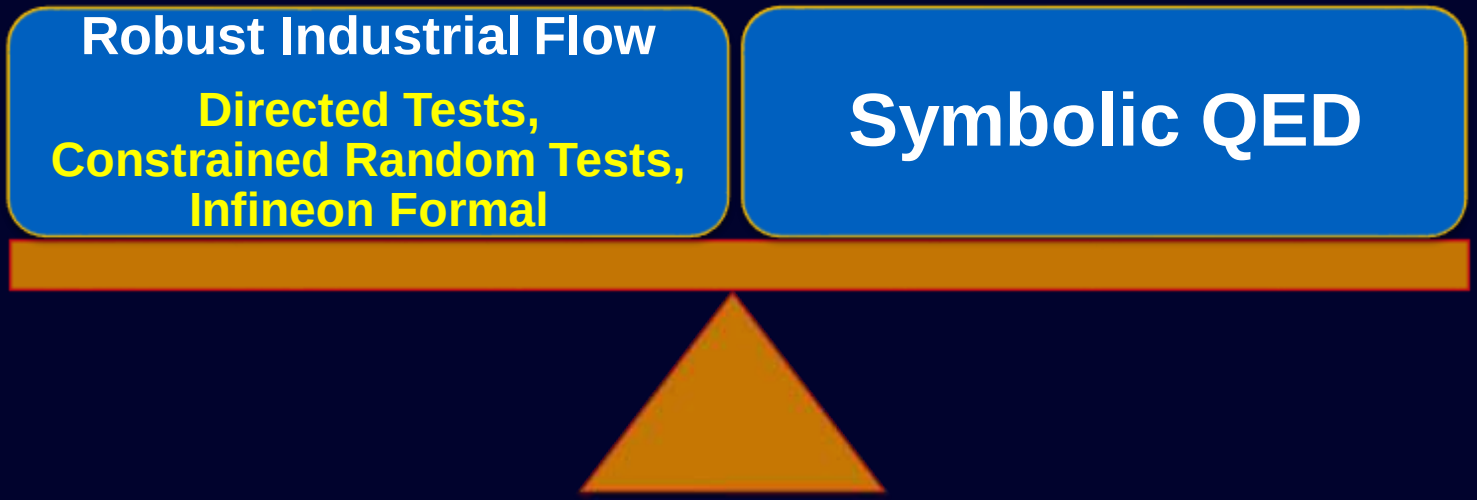
- Introduction
- Infineon Case Study
- QED
- Symbolic QED
- Demos

Pre-Silicon Verification

- Very few “real” innovations over past decade
  - Race against time & complexity
- Product features limited by verification
  - Existing techniques not sustainable

Must Rethink Pre-Silicon Verification!

Case Study Objectives



1. Characterize bugs detected
2. Quantify manual effort
3. Quantify runtime

Designs Analyzed

- Custom microcontroller core (16 versions, verified over 5 years)

3D camera

Airbag

Pressure Monitor

And more…

Automotive Microcontroller Core

- 60 instructions, 1.8K flip-flops, 70K gates
  - ASIL rated
  - Numerous corner case simulation scenarios
- 16 versions over 5 years
  - Feature updates & architecture upgrades
  - Bug fixes

ASIL: Automotive Safety Integrity Level

Infineon Verification Flow

- Robust Infineon approach: proven record
- Microcontroller extensively verified over 5 years
  - Directed Simulation Tests (DST)
  - Infineon Formal (IF)
  - Constrained Random Simulation (CRS)

Directed Simulation Tests

- Applied by Design Engineers
- Manual testbench with direct stimuli
- Not meant for comprehensive verification
- Errors found are not tracked
- Prior to design release to verification

Directed Simulation Tests

- Objective – verify specific design features
  - State transitions – reset → fetch → execute
  - Instruction class – single/dual opcode decoding

Initial Design	Subsequent Designs
10 person months	**1-3** person months

Infineon Formal

- Executed by Verification Engineers
- Automatic property generation
  - In-house automation framework
  - Property for each instruction & feature
- Commercial Formal tool – Onespin DV 360

Infineon Formal

- Property for ADD instruction

```
property check_add_instruction is
   assume:
     at t: instruction == ADD;
     at t: core_state == decode;
   prove:
     at t+1: core_state == execute;
     at t+1: result == op1 + op2;
     at t+2: regfile_data_in == result @ t+1;
endproperty;
```

CEX: False Failure

Infineon Formal

- Property for ADD instruction

```
property check_add_instruction is
   assume:
     at t: instruction == ADD;
     at t: branch_en != '1';
     at t: core_state == decode;
   prove:
     at t+1: core_state == execute;
     at t+1: result == op1 + op2;
     at t+2: regfile_data_in == result @ t+1;
endproperty;
```

CEX: False Failure

Infineon Formal

- Property for ADD instruction

```
property check_add_instruction is
   assume:
     at t: instruction == ADD;
     at t: branch_en != '1';
     at t: hzd_stall != '1';
     at t: core_state == decode;
   prove:
     at t+1: core_state == execute;
     at t+1: result == op1 + op2;
     at t+2: regfile_data_in == result @ t+1;
endproperty;
```

Infineon Formal

```
property check_add_instruction is
   assume:
      at t: branch_en != '1';
      at t: hzd_stall != '1';
      at t: instruction == ADD;
      at t: core_state == decode;
   prove:
      at t+1: core_state == execute;
      at t+1: result == op1 + op2;
      at t+2: regfile_data_in == result @ t+1;
endproperty;
```

- Add constraints to exclude **False Failures**
- Ensure over-constraining is absent
- Additional effort!

Infineon Formal

- Formal coverage
  - 100% assertion pass
  - 100% structural coverage

Initial Design	Subsequent Designs
5 person months	**1** person months

Constrained Random Simulation

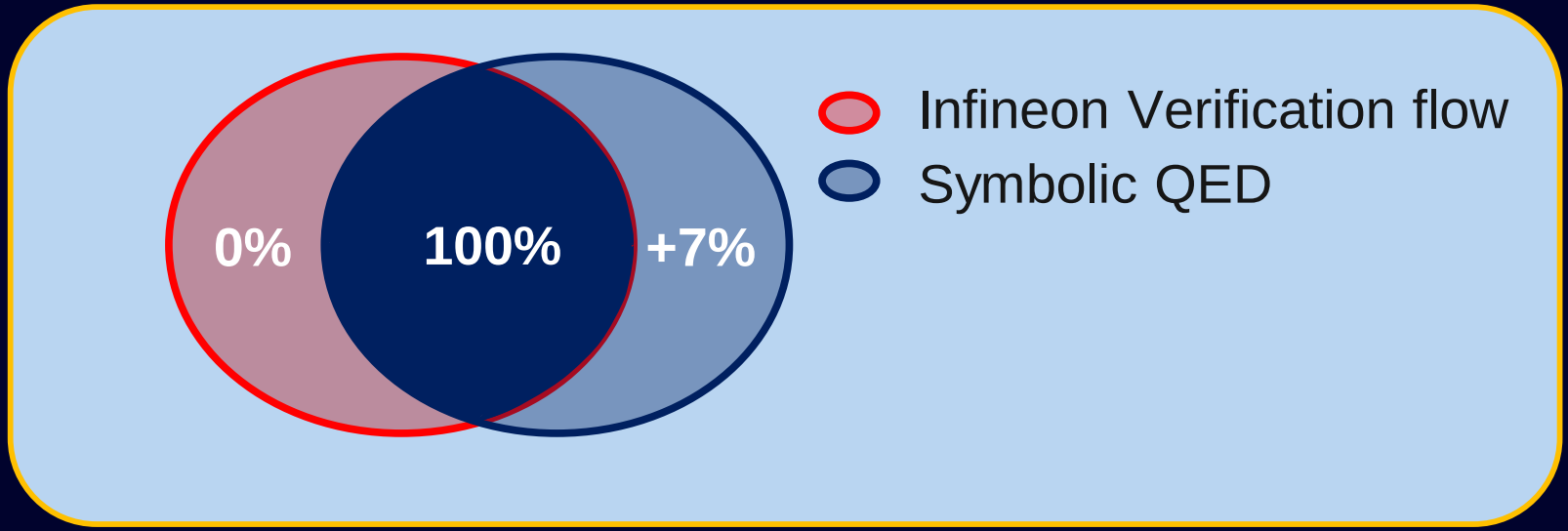
- UVM-SV testbench
- Extensive set of test cases
- Constrained random stimuli
  - Randomize -
    - state transitions
    - instructions and instruction sequences
    - operand values

Constrained Random Simulation

- Regression runs spanning days
- Commercial simulation and debugging tools
- Simulation Coverage
  - 100% functional
  - 100% structural

Initial Design	Subsequent Designs
12 person months	**3-6** person months

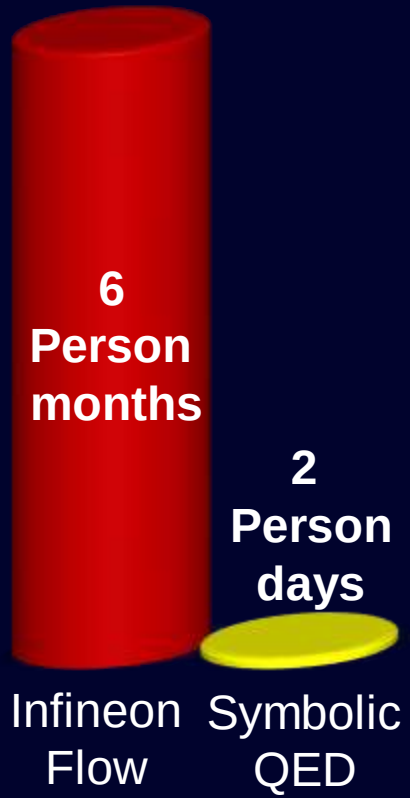
1. Characterize Bugs Detected



- Symbolic QED detected ALL (known) bugs
- And more (specification errors: +7%)

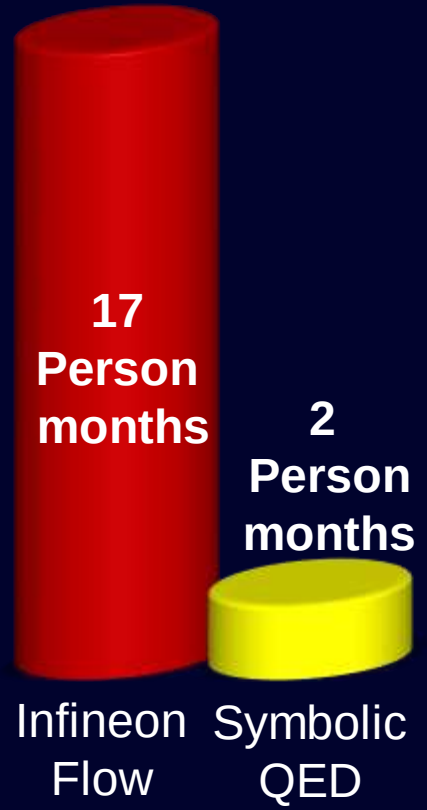
2. Quantify Effort: Reused Designs

- Most Infineon designs are reused
  - Feature updates
  - Architecture upgrades
- Symbolic QED: 60X improvement



2. Quantify Effort: Designs from Scratch

- Rarely designs from scratch
- Symbolic QED: 8X improvement



3. Quantify Runtime

	Error Trace Length [min, avg, max]	**Bug Detection Runtime** [min, avg, max]
Symbolic QED	[4, 6, 10] instructions	[6, 8, 12] seconds

- Short error trace length (4-10 instructions)
- Significant benefits: bug localization and debug
- Short runtime (6-12 seconds)

Conclusion

Symbolic QED

- Micro-controller Cores, automotive products
- **All** (previously detected) logic bugs & **more** detected
- **60x** verification productivity improvement

Product Impact

- Shorter Time-to-Market
- New design features

Ongoing

- Infineon Symbolic QED efforts
- Symbolic QED beyond design verification

Outline

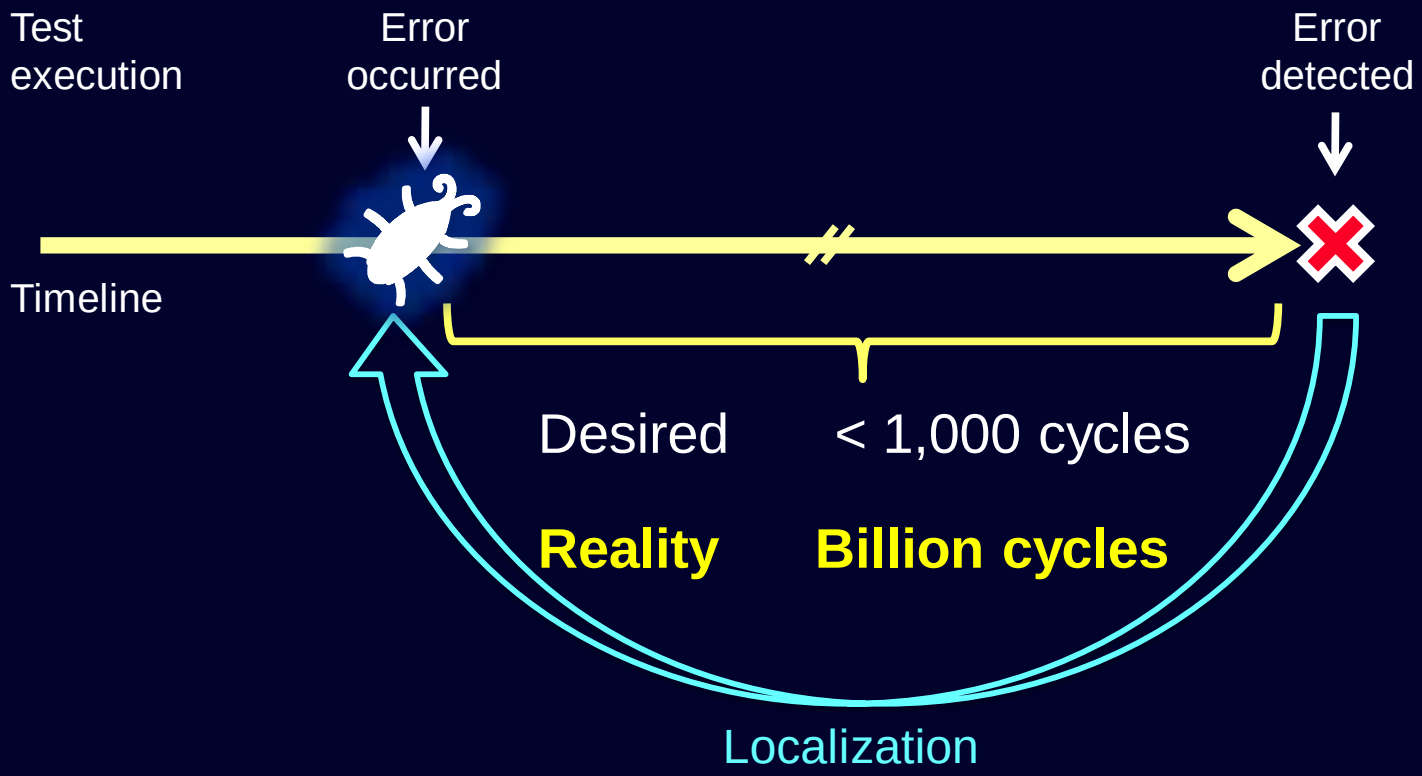
- Introduction
- Infineon Case Study
- QED
- Symbolic QED
- Demos

QED

- Post-silicon
  - Electrical bugs, logic bugs

[Lin IEEE TCAD 14]

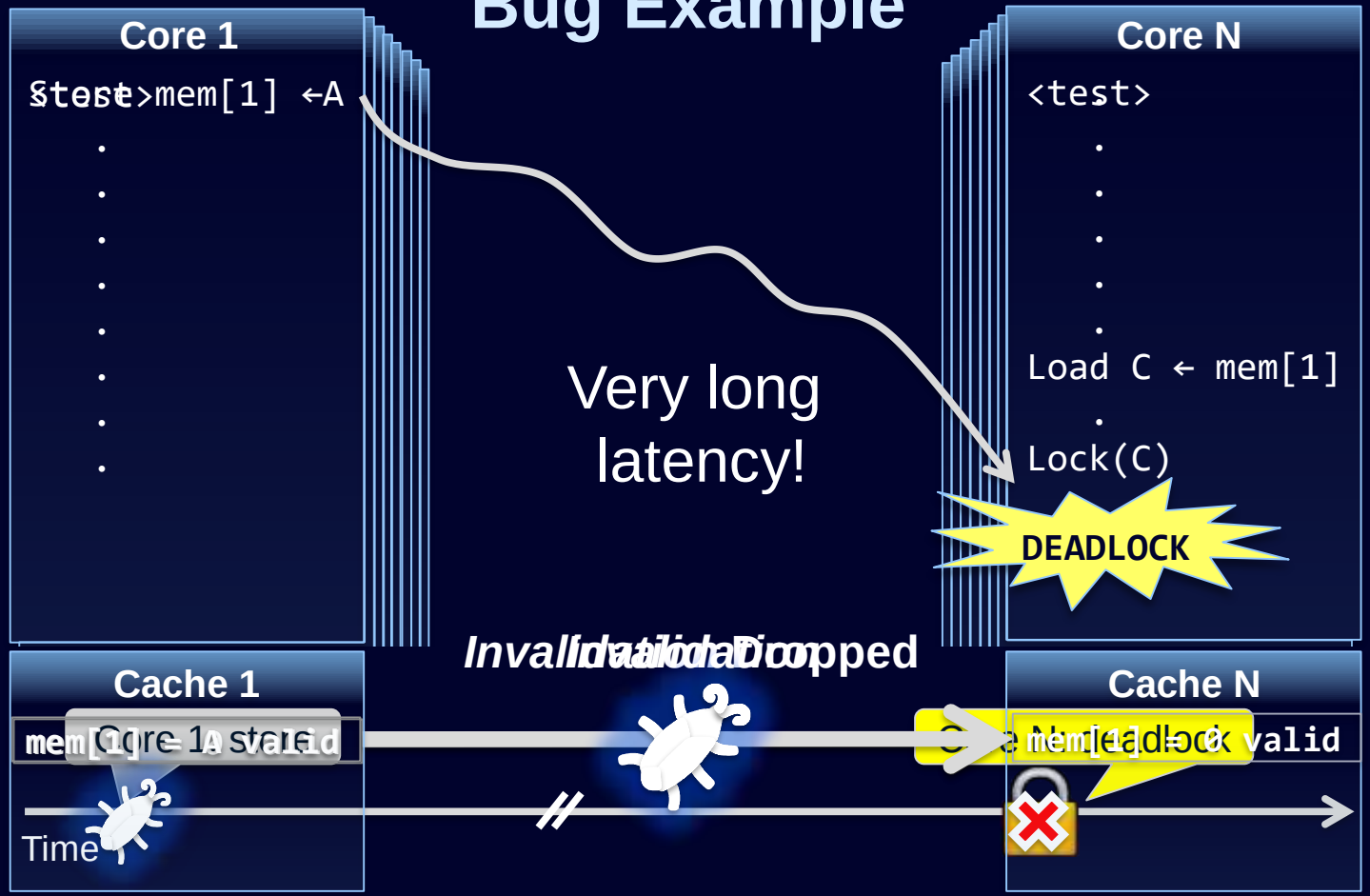
Error Detection Latency



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>
Store mem[1] ← A
Core N
<test>
Load C ← mem[1]
Lock(C)
DEADLOCK
Very long latency!
Invalidation Dropped
Cache 1
mem[1] = A valid
Core 1 stalls
Cache N
mem[1] = A valid
Time

Existing Techniques Inadequate

- Very long error detection latencies

Deadlock detection

Design-specific assertions

Self-checking tests

Store readback tests

Checkpointing

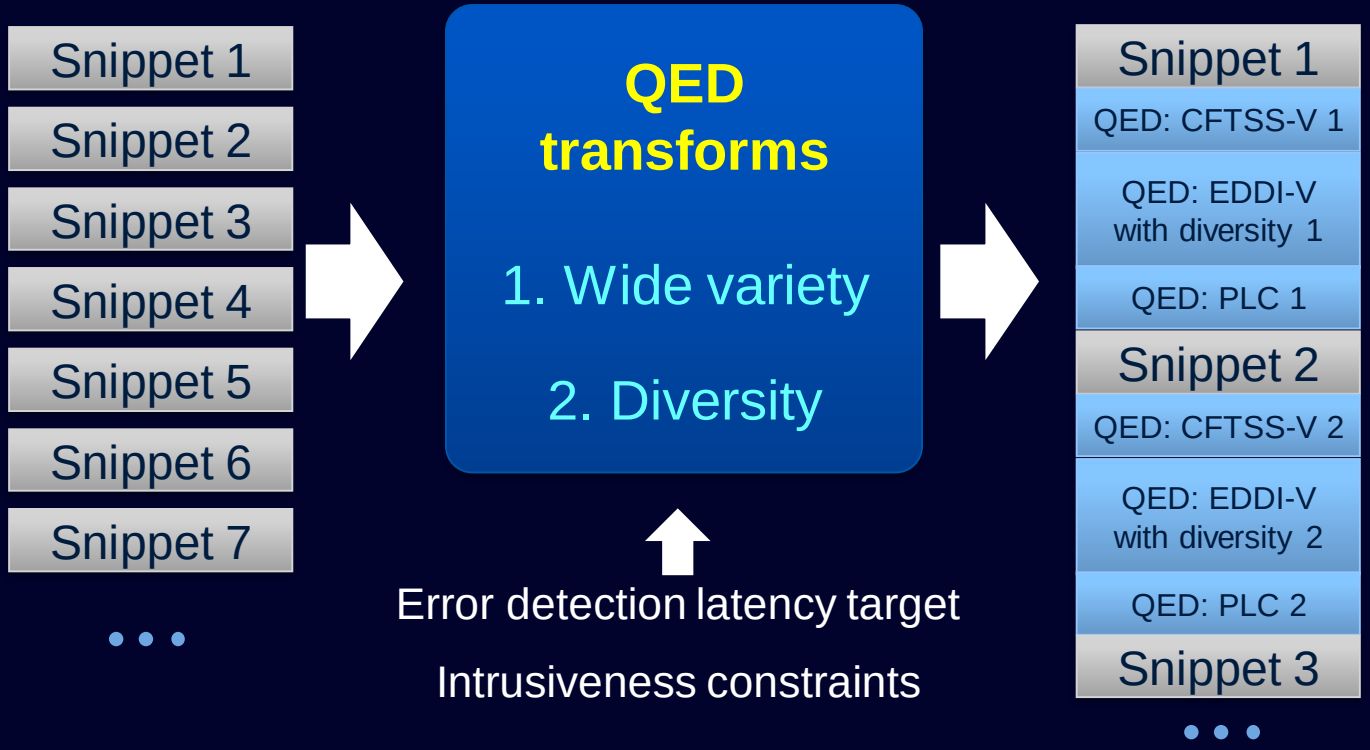
etc…

Quick Error Detection

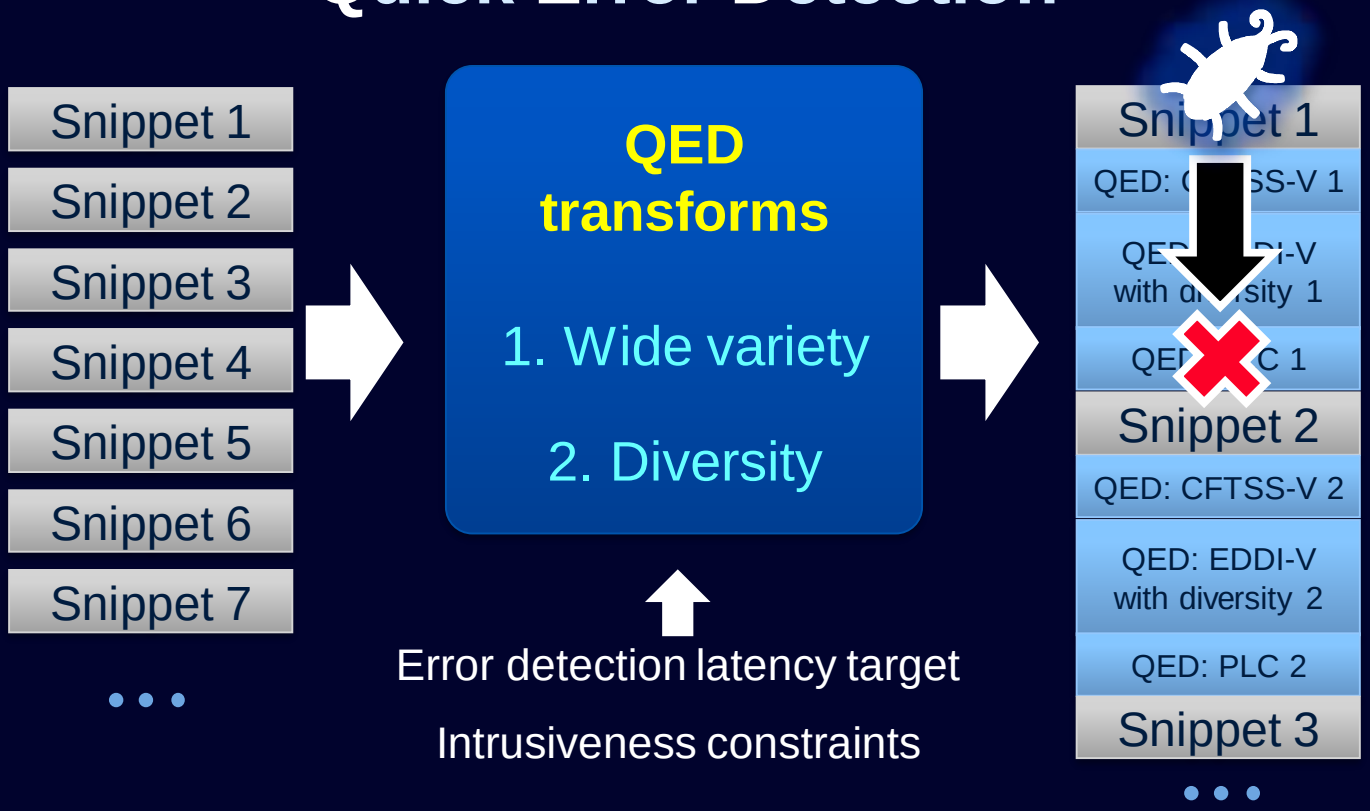
- Error detection latency: guaranteed short
- Coverage: improved
- Software-only: readily applicable

[Lin IEEE TCAD 14]

Quick Error Detection

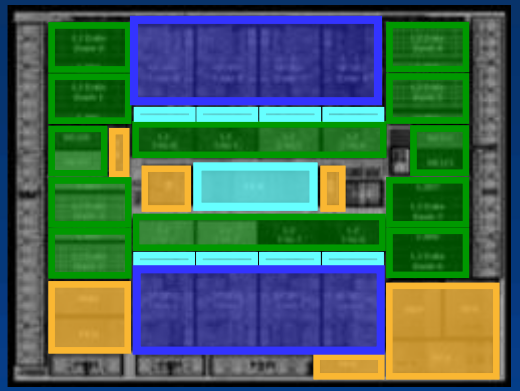


Quick Error Detection



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 % 0x10000000001;
            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+l],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
```

• • •

```
Check end_result
```

Very Long!

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 % 0x10000000001;
            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+l],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
```

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+l],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 ← 0

• • •

Check R1 == 0

R1

3	~~2~~

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 % 0x10000000001;
            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 % 0x10000000001;
            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 {
```

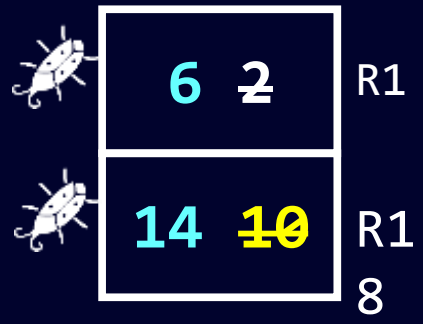
QED Trace

a ← 1; a' ← a; b ← 1, b' ← b

R1 ← a + b

R18 ← 5a' + 5b'

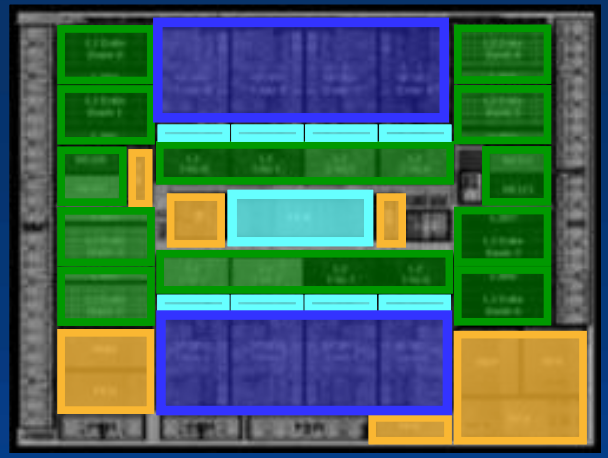
5 × R1 == R18



Many diversity techniques

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

QED Transforms



Processor cores

Uncore, accelerators

EDDI-V

CFTSS-V

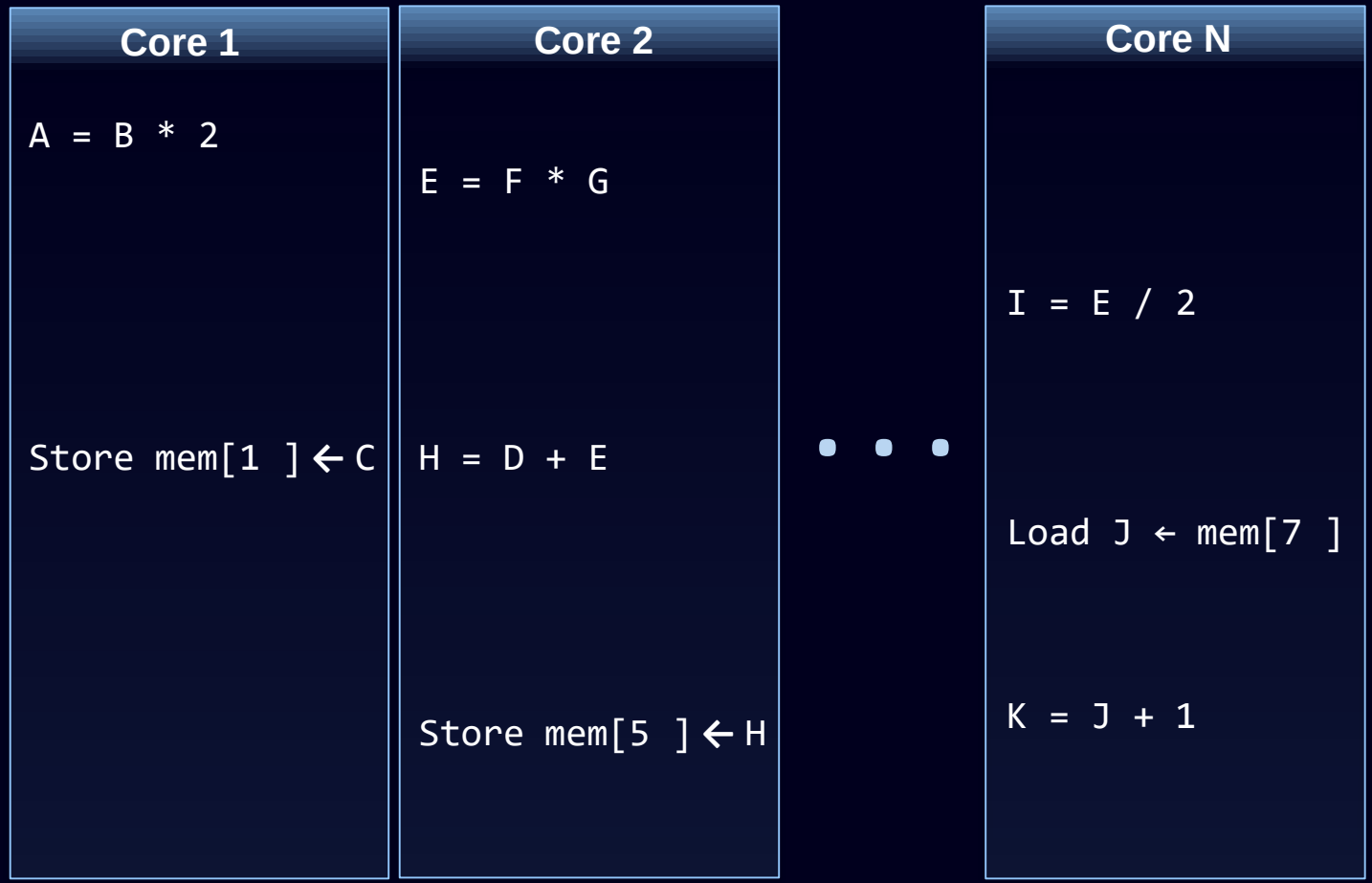
CFCSS-V

PLC

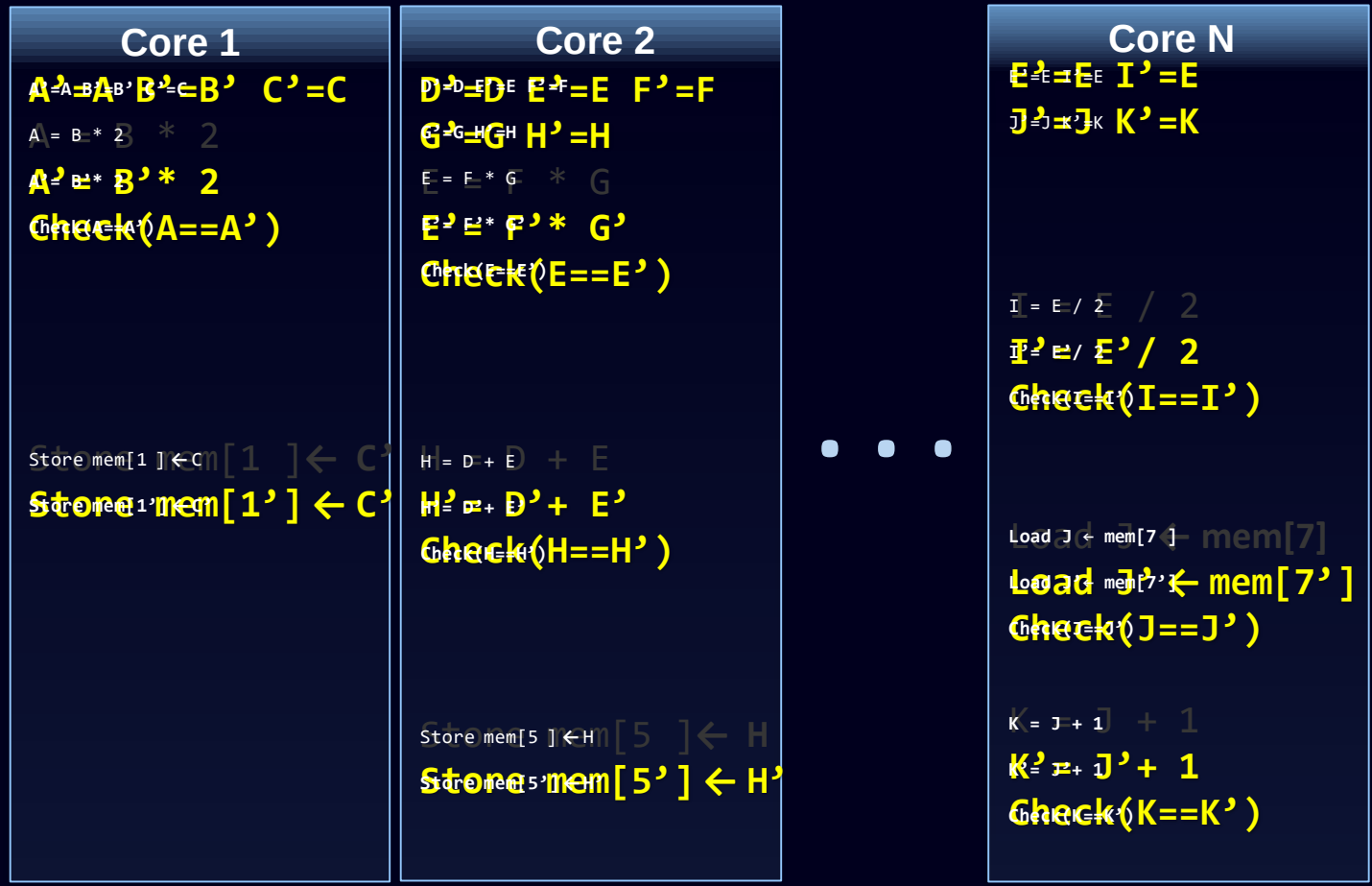
Fast QED

Hybrid QED

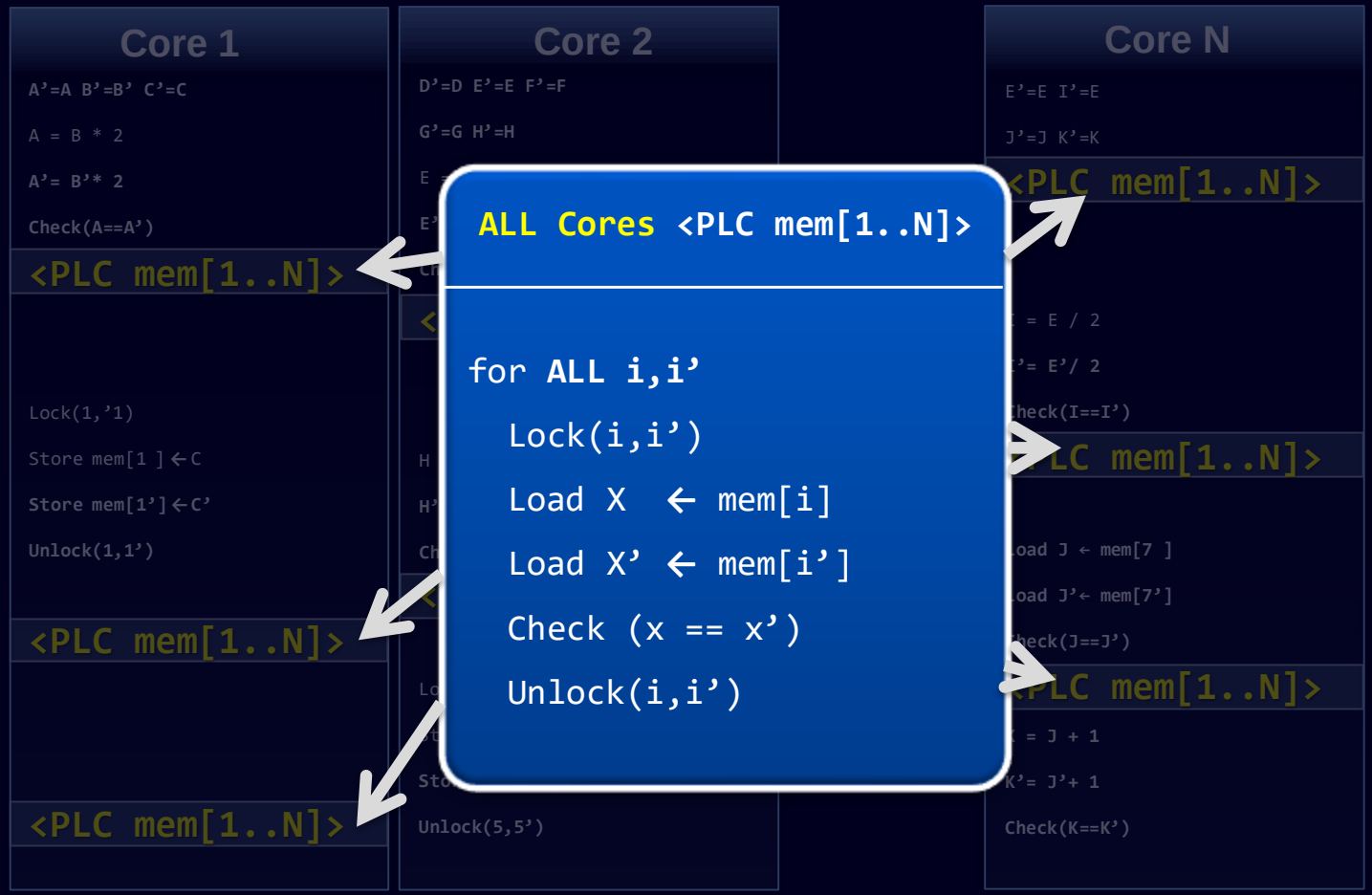
Original Test



QED: Duplicate & Check



QED: Proactive Load & Check



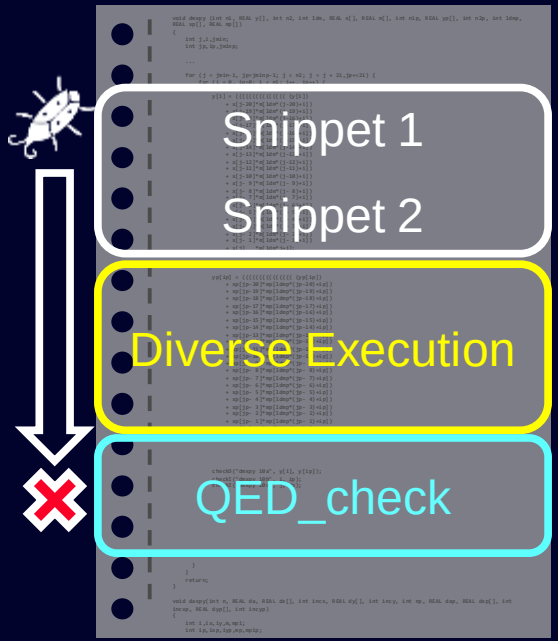
QED Coverage Considerations

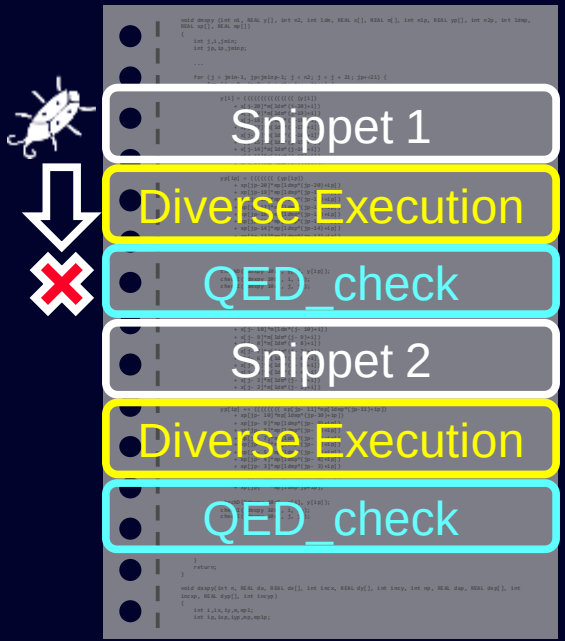
- Challenge
  - Intrusiveness: coverage impact ?
- Systematic solutions
  - QED family tests
  - Hardware-enhanced QED

Error Detection Latency vs. Intrusiveness

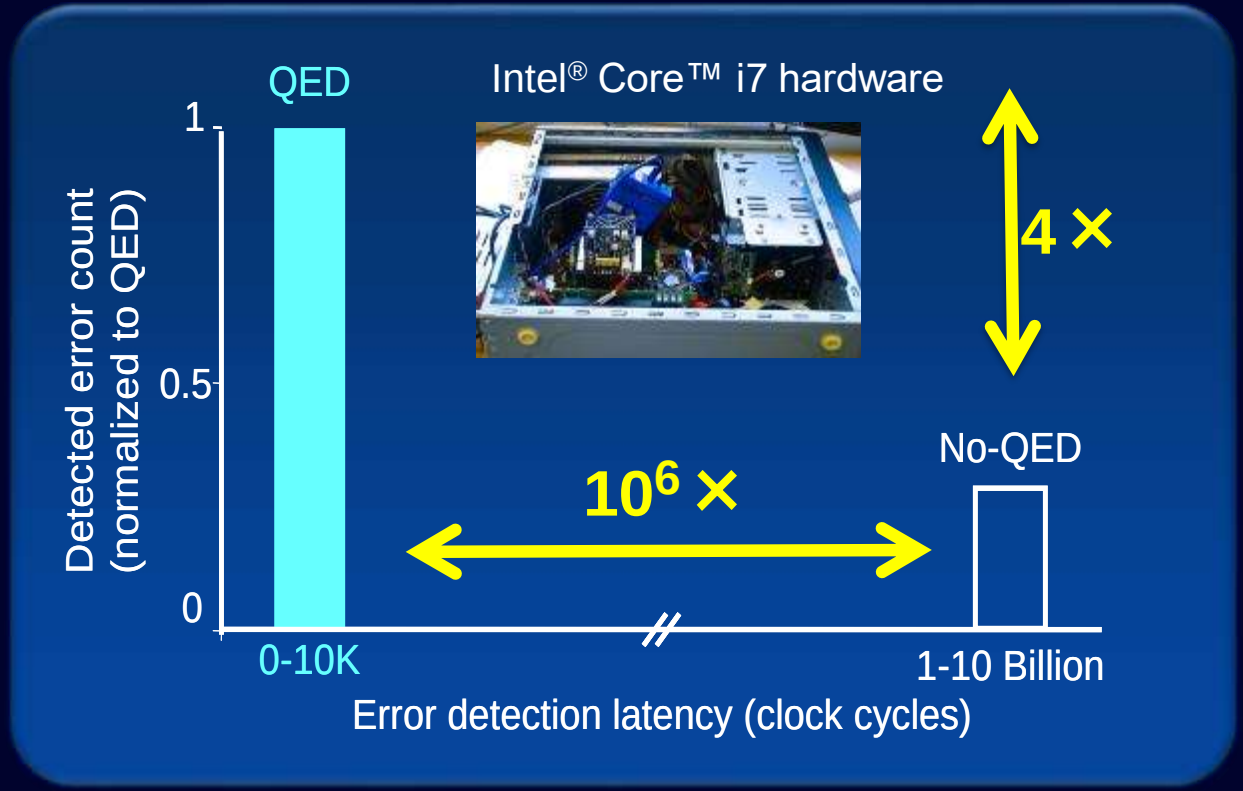
☺ Improved error detection latency

😐 Increased intrusiveness?





QED Effective for Electrical Bugs



[Lin IEEE TCAD 14]

EQED

- **Electrical bug localization**

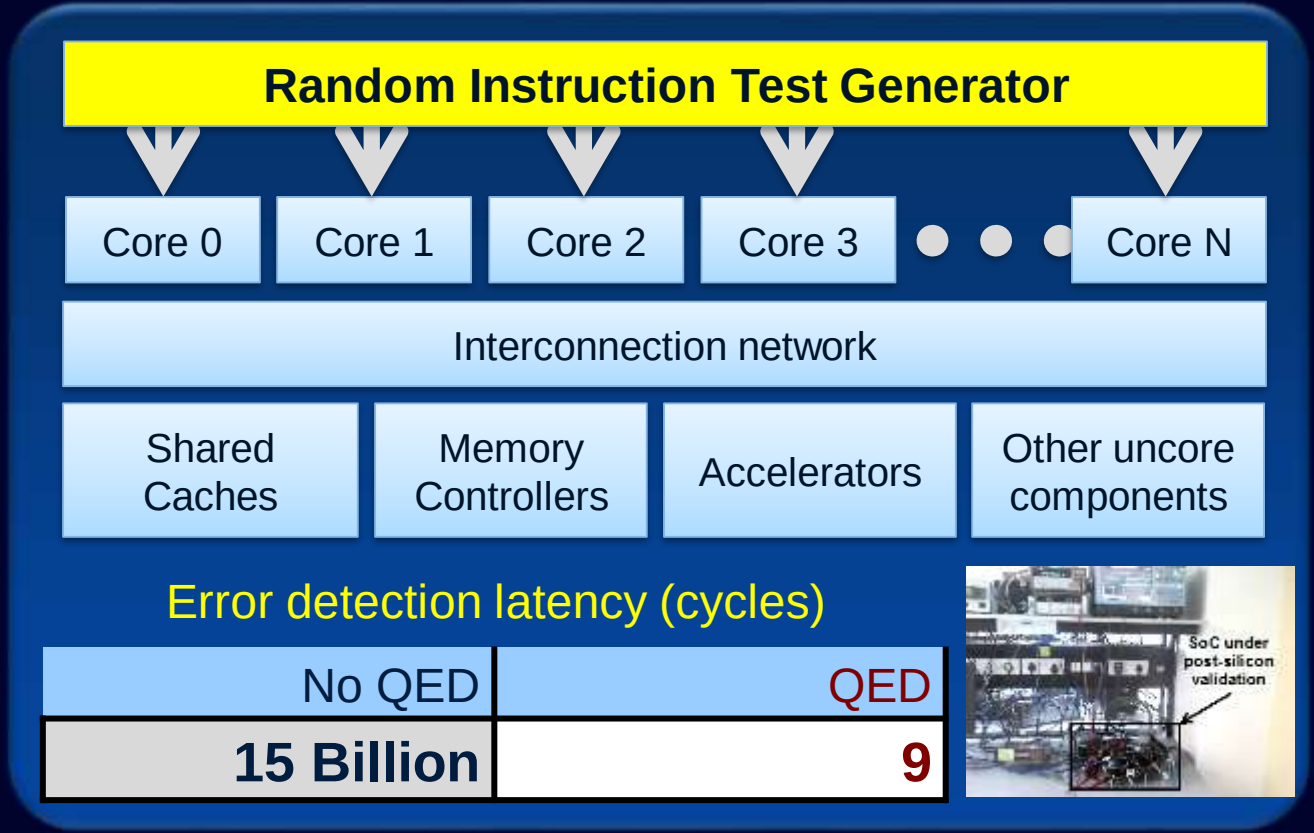
	E-QED
Flip-flop candidates	18 (1 million total) **50,000× localization**
Area impact	2.5% (actually 0%)
Debug effort	Automatic
Runtime	~ 9 hours

OpenSPARC T2 SoC (500M transistors)

[Singh CAV 17]

QED Effective for Logic Bugs

Freescale SoC hardware



Error detection latency (cycles)

No QED	QED
15 Billion	**9**

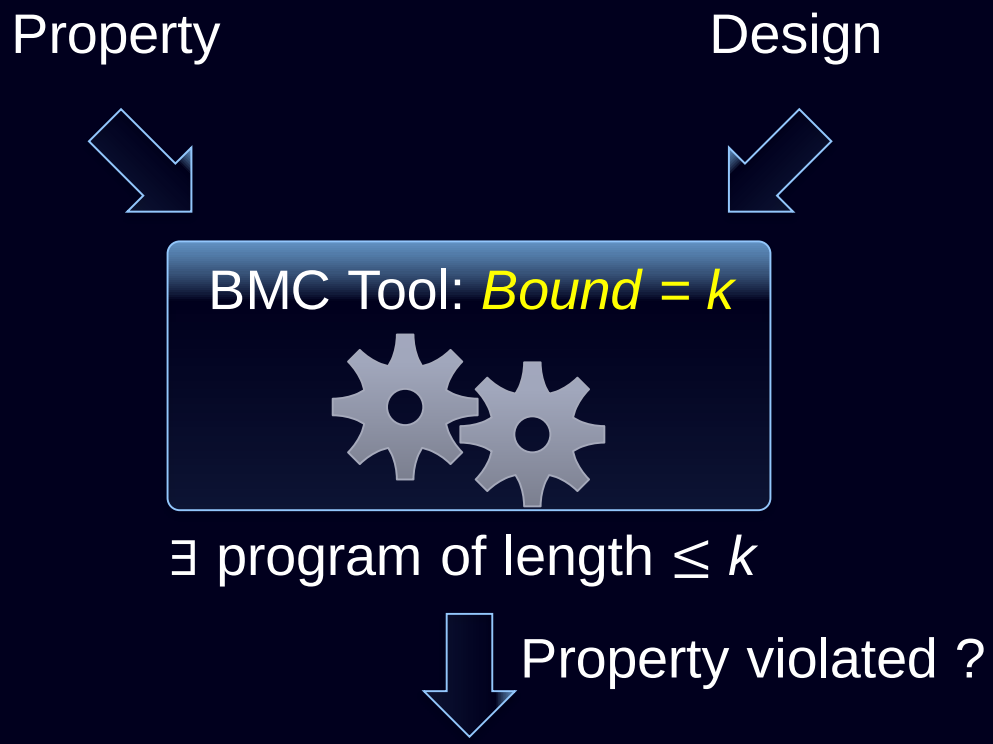
[Lin IEEE TCAD 14]

Symbolic QED

- Pre-silicon
  - Logic bugs: processors, accelerators, SoCs

[Lin ITC 15, Singh IEEE TCAD 18]

Traditional Bounded Model Checking



Traditional BMC vs. Symbolic QED

	Traditional BMC	Symbolic QED
Properties	Manual	**Automatic QED checks (Universal property)**
Design size	Small blocks	**Large SoCs**

BMC Symbolic QED

Design

BMC Tool: *Bound = k*

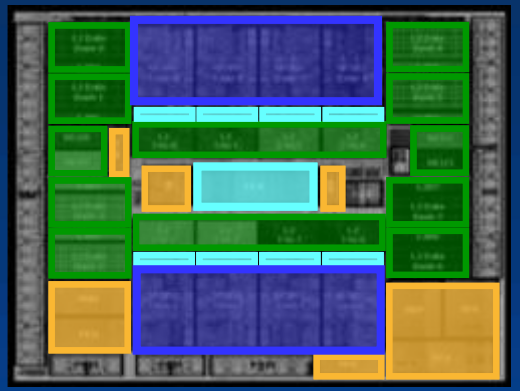
∃ **QED program** of length ≤ *k* that fails ?

Bug found

[Lin ITC 15, Singh IEEE TCAD 18]

QED Transforms

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



Processor cores

Uncore, accelerators

EDDI-V

CFTSS-V

CFCSS-V

PLC

Fast QED

Hybrid QED

“Universal” Property: QED Check

CMP Ra == Ra’

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

[Lin ITC 15, Singh IEEE TCAD 18]

BMC Symbolic QED

"Universal" Property

Design

If property violated

Counter-example = bug trace

[Lin ITC 15, Singh IEEE TCAD 18]

Need Input Constraints

- If unconstrained, BMC may choose any inputs

Ra ← 1

Ra’ ← 2

CMP Ra == Ra’

- False fail – not a bug

Input Constraints

- Original sequence then duplicated sequence
  - CMP Ra == Ra’ after both executed

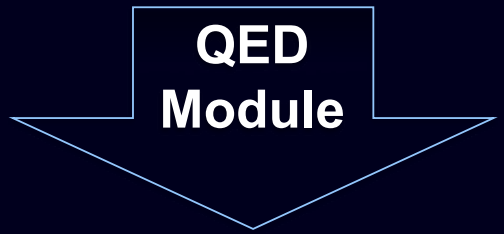
- Enforced by QED module automatically
  - Only during BMC
  - Not in fabricated design

Solution: QED Module

Only during BMC, not in fabricated chip

Input
(Chosen by BMC)

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

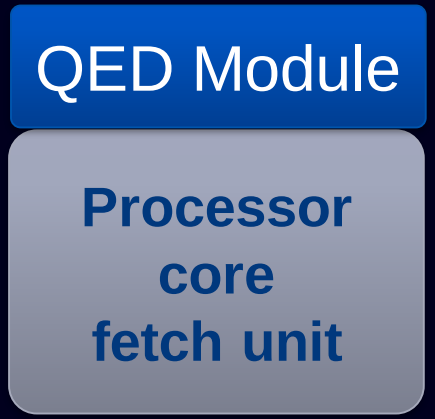


Output

```
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

Design + QED Module
(no hardware overhead)

If property violated

Counter-example = bug trace

[Lin ITC 15, Singh IEEE TCAD 18]

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

- S^2QED: Symbolic initial state [Fadiheh DATE 2018]

BMC Symbolic QED

"Universal" Property +
QED-consistent initial state

Design + QED Module
(no hardware overhead)

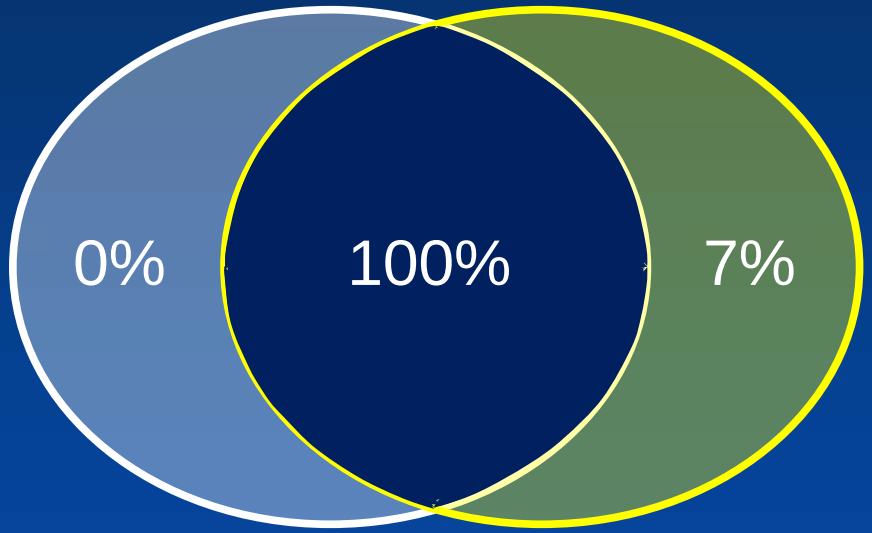
If property violated

Counter-example = bug trace

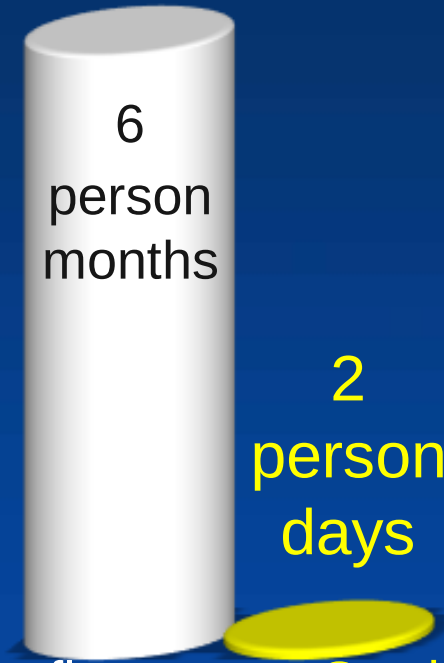
[Lin ITC 15, Singh IEEE TCAD 18]

Symbolic QED: Infineon Study

All (known) bugs + more



60 × productivity



[Singh DATE 19]

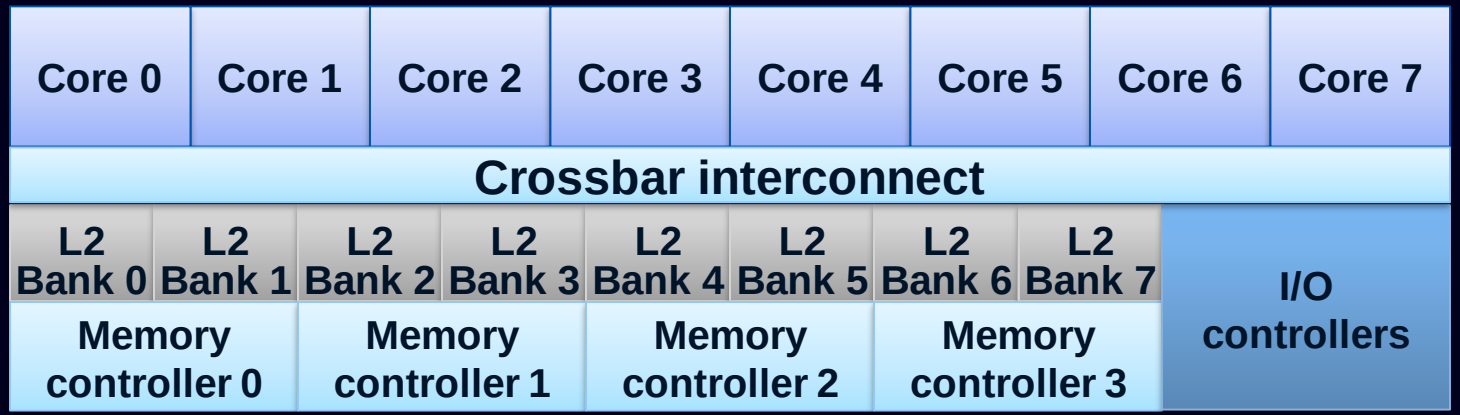
BUT…

- Big designs ?

- **Solution: QED checks compositional**
  - Preserved across partial instances

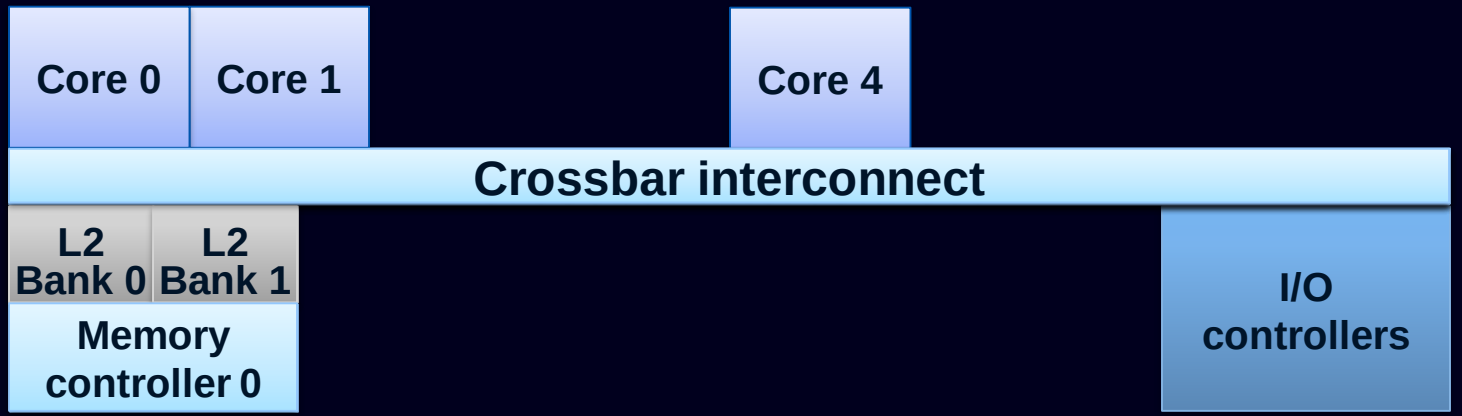
[Lin ITC 15, Singh IEEE TCAD 18]

Solution: Partial Instantiation

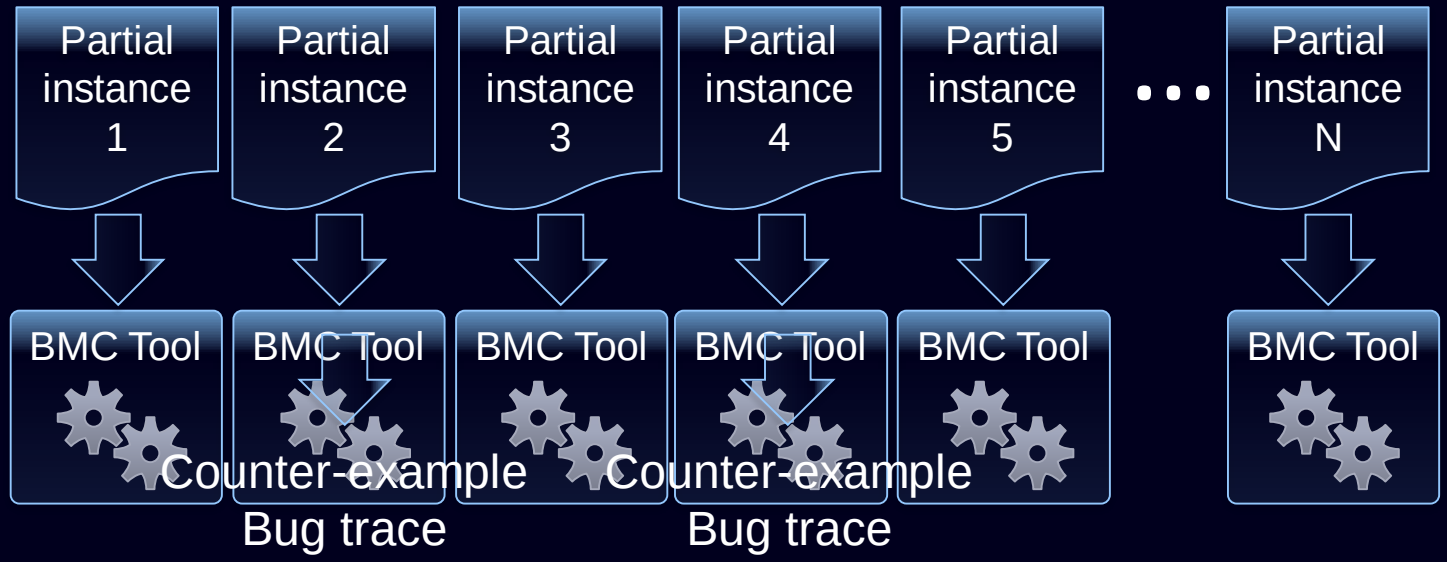


Solution: Partial Instantiation

- At least 1 processor core per partial instance



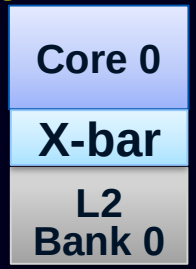
BMC on Partial Instances



BMC on Partial Instances

Counter-example
Bug trace

Bug localized



Bug trace

```
ST [0x10000], R1
ST [0x10040], R2
LD R3, [0x10000]
```

- Smallest partial instance → best localization

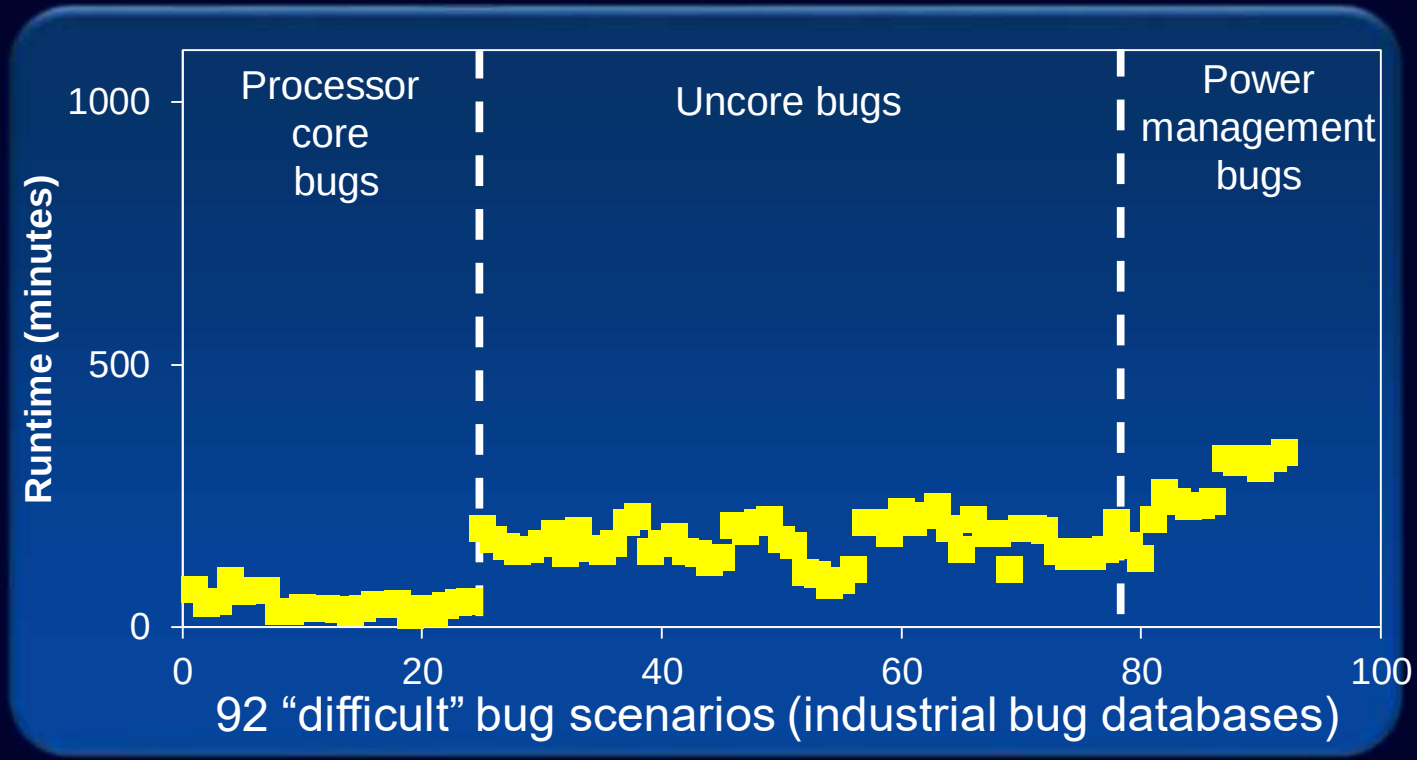
Effective for Logic Bugs

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

[Lin IEEE TCAD 14]

Symbolic QED: Billion-Transistor SoCs

20 mins. to 7 hours, automatic



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

QED and Symbolic QED

- Pre-silicon, post-silicon
  - Automatic, overnight, billion-transistor SoCs
- Broadly applicable
  - Core, uncore, accelerator, logic & electrical bugs
- More opportunities
  - Security, firmware, System-Level Testing