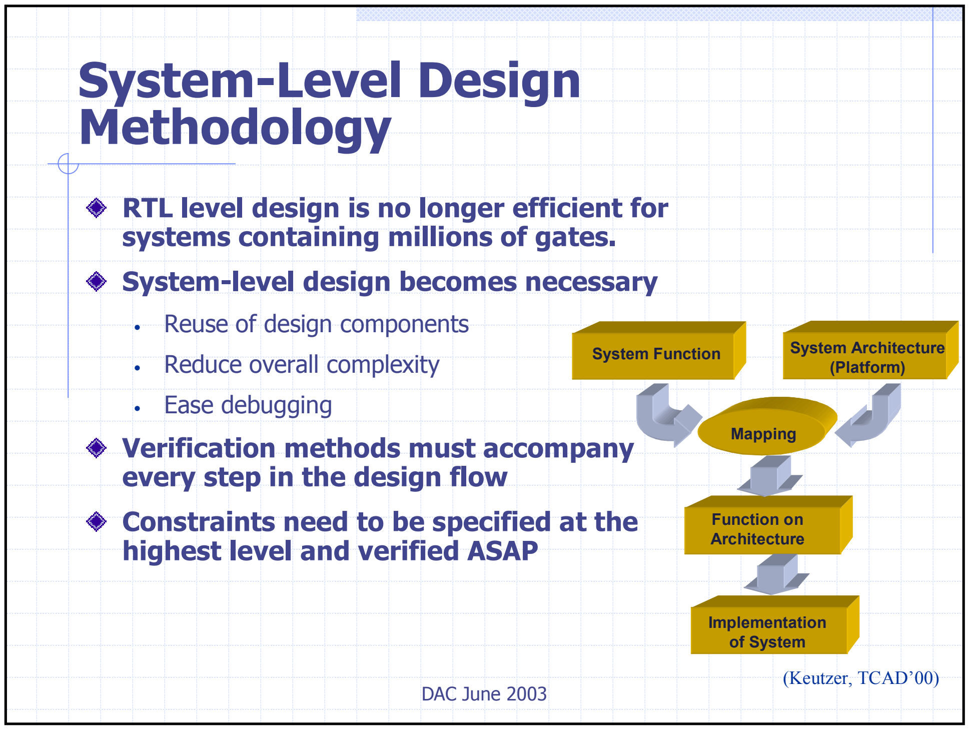
Automatic Trace Analysis for Logic of Constraints

Xi Chen, Harry Hsieh
University of California, Riverside

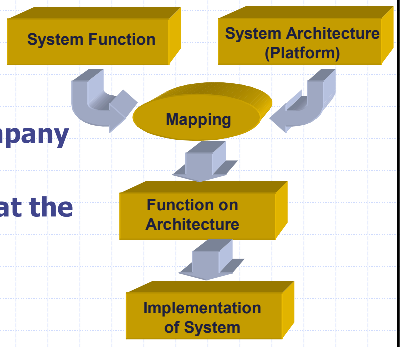
Felice Balarin, Yosinori Watanabe
Cadence Berkeley Laboratories

Outline



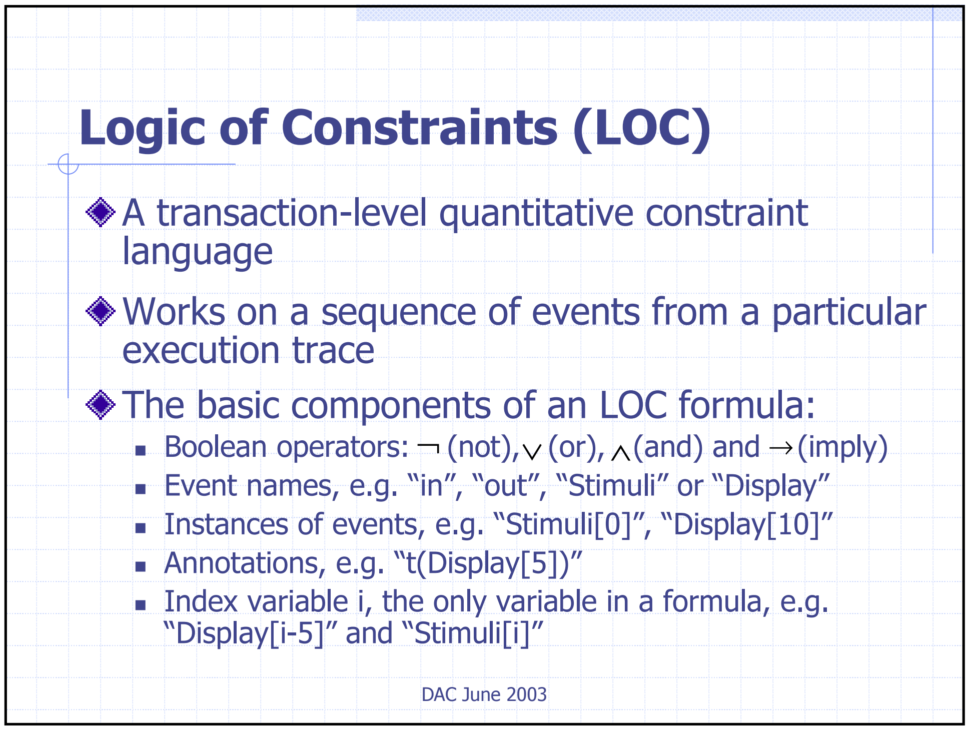
System-Level Design Methodology

- ◈ **RTL level design is no longer efficient for systems containing millions of gates.**
- ◈ **System-level design becomes necessary**
  - Reuse of design components
  - Reduce overall complexity
  - Ease debugging
- ◈ **Verification methods must accompany every step in the design flow**
- ◈ **Constraints need to be specified at the highest level and verified ASAP**



(Keutzer, TCAD'00)

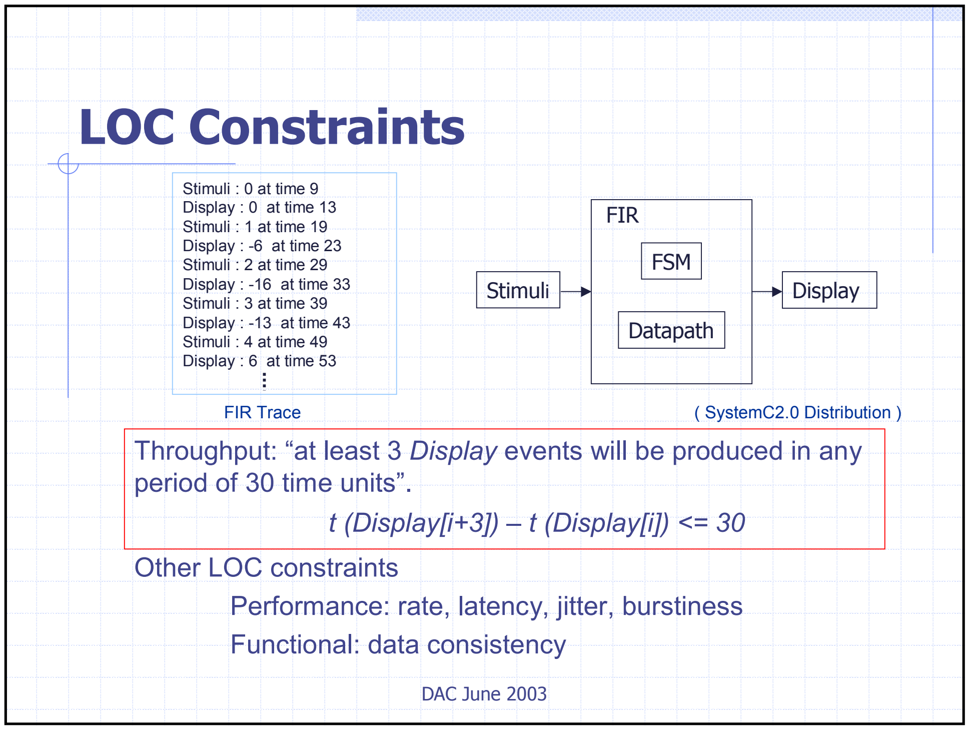
DAC June 2003



Logic of Constraints (LOC)

- ◈ A transaction-level quantitative constraint language
- ◈ Works on a sequence of events from a particular execution trace
- ◈ The basic components of an LOC formula:
  - Boolean operators: $\neg$ (not), $\vee$ (or), $\wedge$ (and) and $\rightarrow$ (imply)
  - Event names, e.g. "in", "out", "Stimuli" or "Display"
  - Instances of events, e.g. "Stimuli[0]", "Display[10]"
  - Annotations, e.g. "t(Display[5])"
  - Index variable i, the only variable in a formula, e.g. "Display[i-5]" and "Stimuli[i]"

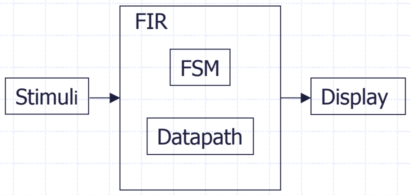
DAC June 2003



LOC Constraints

```
Stimuli : 0 at time 9
Display : 0  at time 13
Stimuli : 1 at time 19
Display : -6  at time 23
Stimuli : 2 at time 29
Display : -16  at time 33
Stimuli : 3 at time 39
Display : -13  at time 43
Stimuli : 4 at time 49
Display : 6  at time 53
⋮
```

FIR Trace



(SystemC2.0 Distribution)

Throughput: “at least 3 *Display* events will be produced in any period of 30 time units”.

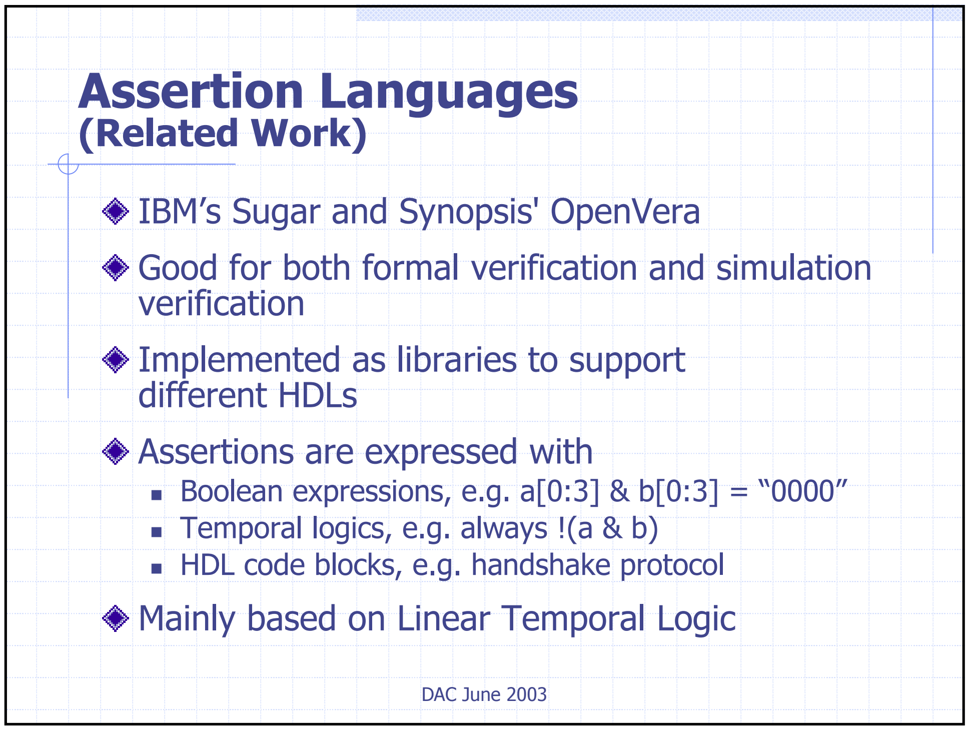
t (Display[i+3]) – t (Display[i]) <= 30

Other LOC constraints

Performance: rate, latency, jitter, burstiness

Functional: data consistency

DAC June 2003



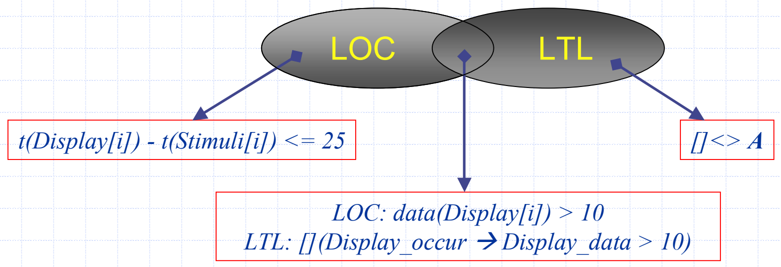
Assertion Languages (Related Work)

- IBM’s Sugar and Synopsis' OpenVera
- Good for both formal verification and simulation verification
- Implemented as libraries to support different HDLs
- Assertions are expressed with
  - Boolean expressions, e.g. a[0:3] & b[0:3] = “0000”
  - Temporal logics, e.g. always !(a & b)
  - HDL code blocks, e.g. handshake protocol
- Mainly based on Linear Temporal Logic

DAC June 2003

Characteristics of LOC Formulism

- Constraints can be automatically synthesized into static checkers, runtime monitors and formal verification models.
- Performance constraints in addition to functional constraints
- A different domain of expressiveness than LTL.

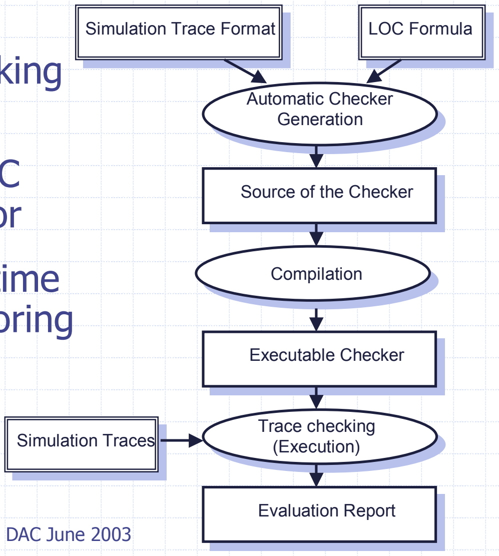

LOC
LTL
t(Display[i]) - t(Stimuli[i]) <= 25
[]<> A
LOC: data(Display[i]) > 10
LTL: [](Display_occur → Display_data > 10)

Outline

- Introduction
- **Trace analysis methodology**
  - Methodology and algorithms
  - Case studies
- Proving LOC formulas
- Summary

Trace Analysis Methodology

- An efficient checking algorithm
- An automatic LOC checker generator
- Extended to runtime constraint monitoring



Algorithm of the LOC Checker

Throughput: "at least 3 *Display* events will be produced in any period of 30 time units"

t (Display[i+3]) – t (Display[i]) <= 30

FIR Trace

Display[0]

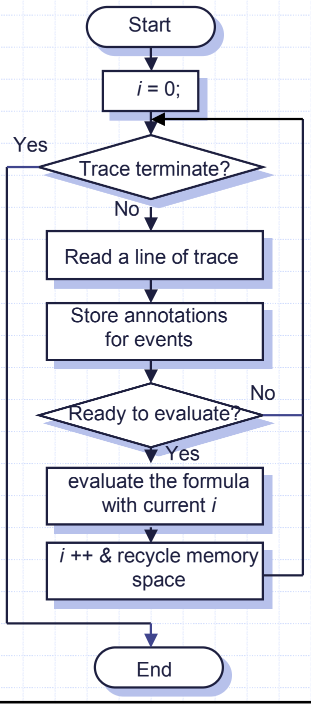
i = 0

Display[3]

```
Stimuli : 0 at time 9
Display : 0  at time 13
Stimuli : 1 at time 19
Display : -6  at time 23
Stimuli : 2 at time 29
Display : -16  at time 33
Stimuli : 3 at time 39
Display : -13  at time 43
Stimuli : 4 at time 49
Display : 6  at time 53
⋮
```

index	0	1	2	3	4	5
t(Display[i])	13	23	33	43	53	63

Queue data structure


Start
i = 0;
Trace terminate?
Yes
No
Read a line of trace
Store annotations for events
Ready to evaluate?
No
Yes
evaluate the formula with current i
i ++ & recycle memory space
End

Input and Output

Input of the checker generator – formula and trace format:

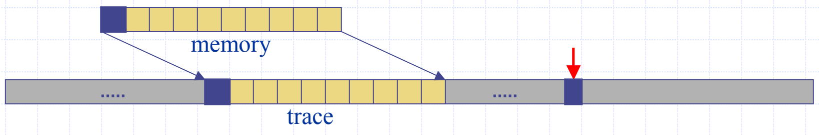
```
[LOC: rate]
    formula: t(Display[i + 1] – t(Display[i]) == 10
    annotation: event value t
    trace: "%s : %d at time %f"
[LOC: latency]
    formula: t(Display[i])–t(Stimuli[i]) <= 25
    annotation: event value t
    trace: "%s : %d at time %f"
```

Output of the trace checking – error report:

```
username@chimera $      checker  latency.trace
Reading from trace file "latency.trace" ...
Formula t(Display[i]) – t(Stimuli[i]) <= 25 is violated
at trace line# 278:     Display : –6 at time 87
where i = 23
t (Display[i]) =  87
t (Stimuli[i]) = 60
                              ⋮
```

Dealing with Memory Limitation

- scan trace and store the annotations only once.
- If the memory limit has been reached,
  - stop storing more annotations
  - search the rest of trace for current i
  - resume storing annotations after freeing memory

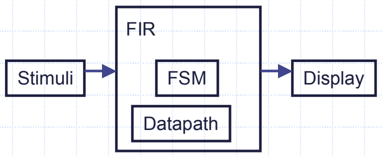


Static Trace Checking v.s. Runtime Monitoring

- Runtime constraint monitoring:
  - Integrate the trace checker into a compiled-code simulator, e.g. SystemC modules
  - At runtime, the events and annotations are passed to the monitor module directly
  - Static trace checking v.s. runtime monitoring

	Static Trace Checking	Runtime Monitoring
Steps	Simulation, then checking	Simulation || Checking
Time	More	Less
Space	More	Less
Debug	Easy	Hard
Simulator	Independent	Dependent

Case Study – FIR Filter

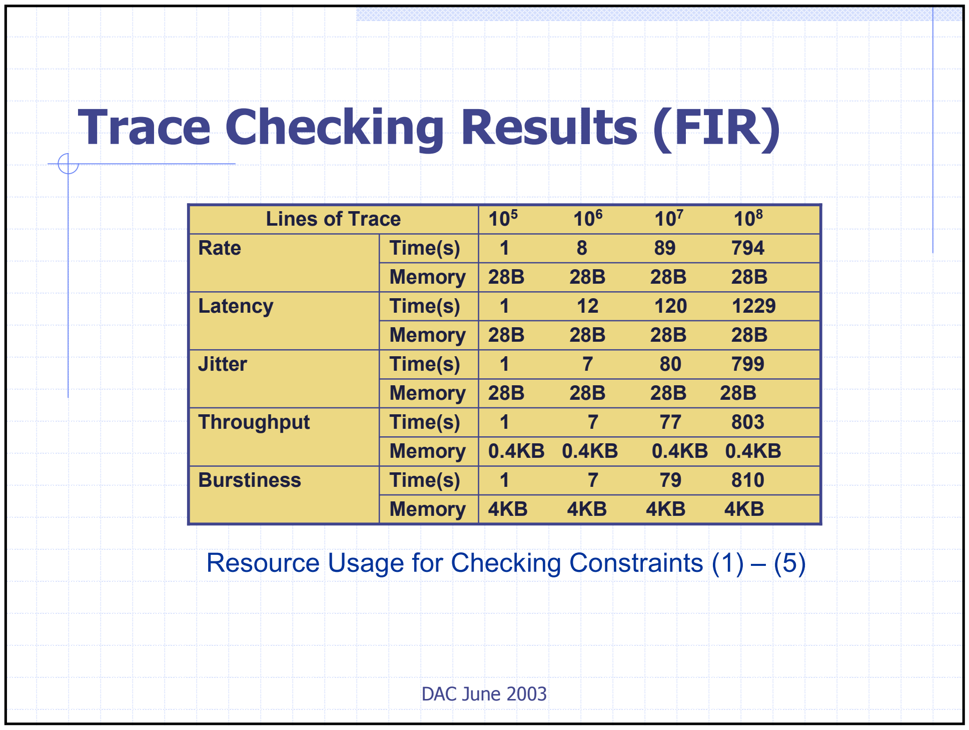


(SystemC Distribution)

Stimuli : 0 at time 9
Display : 0 at time 13
Stimuli : 1 at time 19
Display : -6 at time 23
Stimuli : 2 at time 29
Display : -16 at time 33
Stimuli : 3 at time 39
Display : -13 at time 43
Stimuli : 4 at time 49
Display : 6 at time 53
⋮

FIR Trace

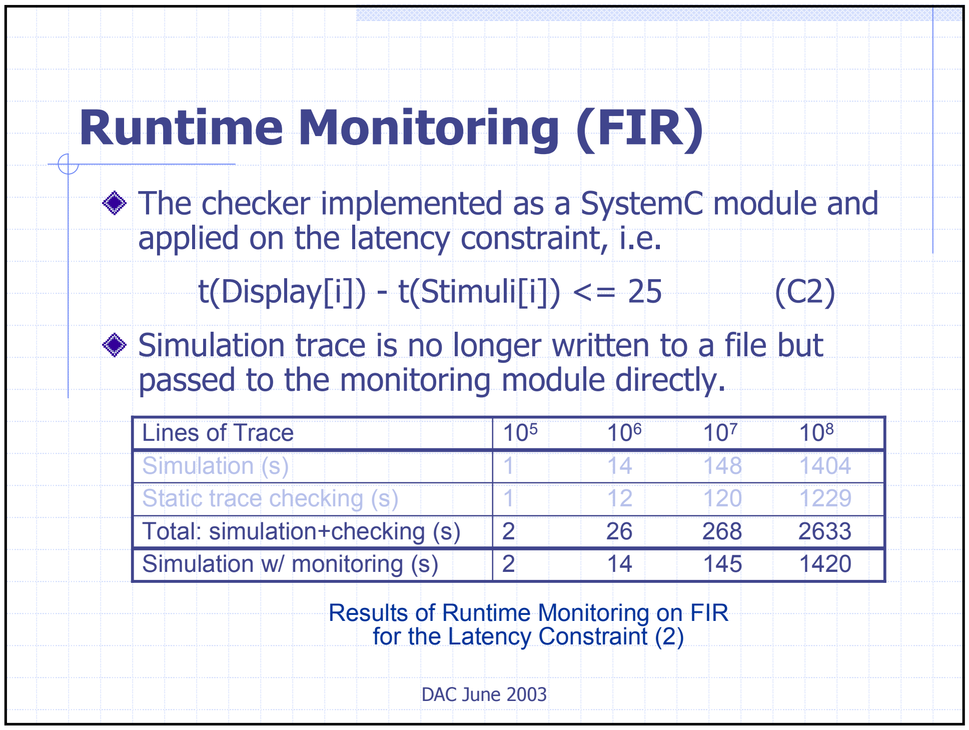
Rate:	t(Display[i+1])-t(Display[i]) = 10
Latency:	t(Display[i]) - t(Stimuli[i]) <= 25
Jitter:	| t(Display[i]) - (i+1) * 10 | <= 4
Throughput:	t (Display[i+100])-t(Display[i]) <= 1001
Burstiness:	t (Display[i+1000])-t(Display[i]) > 9999



Trace Checking Results (FIR)

Lines of Trace		10^5	10^6	10^7	10^8
Rate	Time(s)	1	8	89	794
	Memory	28B	28B	28B	28B
Latency	Time(s)	1	12	120	1229
	Memory	28B	28B	28B	28B
Jitter	Time(s)	1	7	80	799
	Memory	28B	28B	28B	28B
Throughput	Time(s)	1	7	77	803
	Memory	0.4KB	0.4KB	0.4KB	0.4KB
Burstiness	Time(s)	1	7	79	810
	Memory	4KB	4KB	4KB	4KB

Resource Usage for Checking Constraints (1) – (5)



Runtime Monitoring (FIR)

- The checker implemented as a SystemC module and applied on the latency constraint, i.e.

  t(Display[i]) - t(Stimuli[i]) <= 25 (C2)

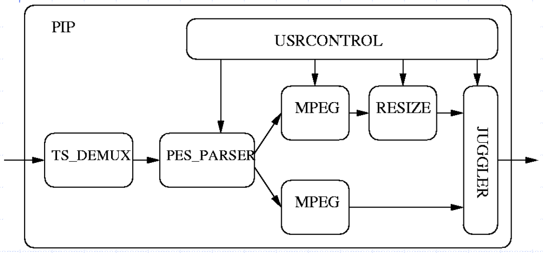
- Simulation trace is no longer written to a file but passed to the monitoring module directly.

Lines of Trace	10^5	10^6	10^7	10^8
Simulation (s)	1	14	148	1404
Static trace checking (s)	1	12	120	1229
Total: simulation+checking (s)	2	26	268	2633
Simulation w/ monitoring (s)	2	14	145	1420

Results of Runtime Monitoring on FIR
for the Latency Constraint (2)

Case Study – Picture In Picture

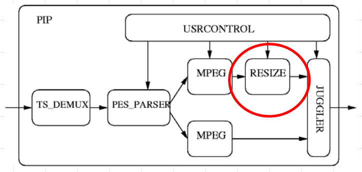
- Picture-In-Picture (PIP)
  - a system level design for set-top video processing
  - 19, 000 lines of source code
  - 120, 000 lines of simulation output (control trace)



```
                              ⋮
WINDOW_DATA_OUT 23483 87000
WINDOW win_params_update x_begin: 12 y_begin: 6
RESIZE field_start field_count: 2 size: 6720
                              ⋮
WINDOW win_params_update x_begin: 12 y_begin: 6
USRCONTROL write pixels_out: 144
RESIZE field_start field_count: 3 size: 10368
                              ⋮
USRCONTROL write lines_out: 64
THSRC_CTL_OUT finfo_write value: 12876
RESIZE field_start field_count: 4 size: 14016
                              ⋮
```

PIP trace

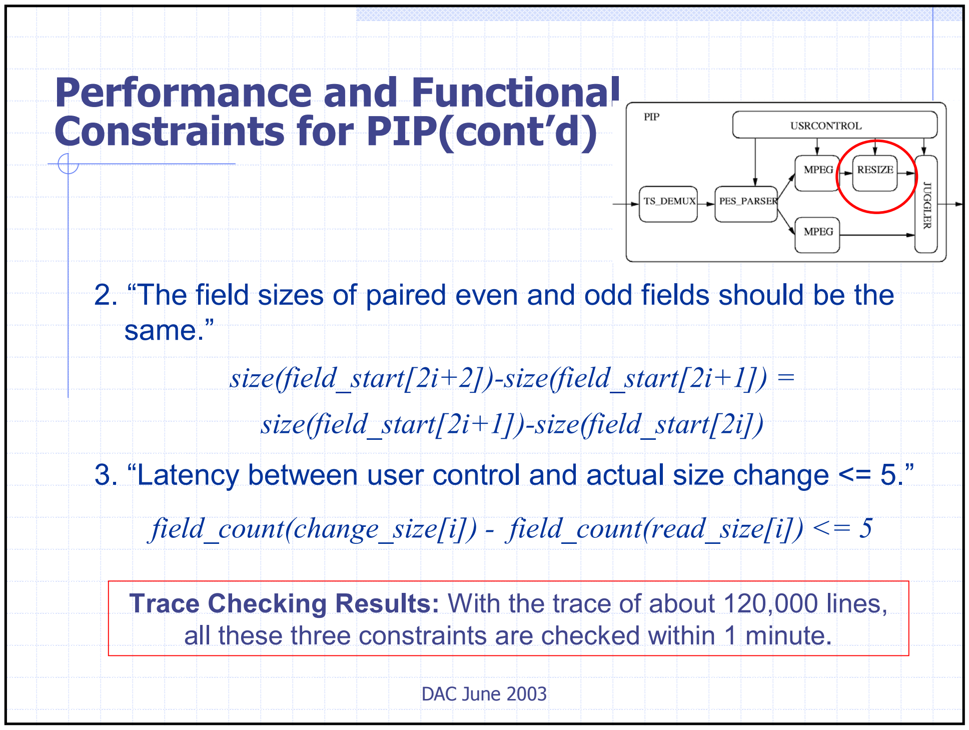
Performance and Functional Constraints for PIP(cont'd)



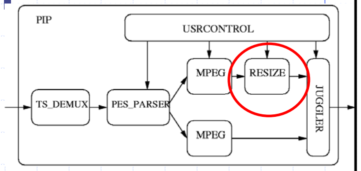
1. Data consistency: "The numbers of the fields read in and produced should be equal."

$$field\_count(in[i]) = field\_count(out[i])$$

- *field_count* is an annotation: number of fields processed
- *in, out* are events: reading a field and producing a field



Performance and Functional Constraints for PIP(cont'd)



2. "The field sizes of paired even and odd fields should be the same."

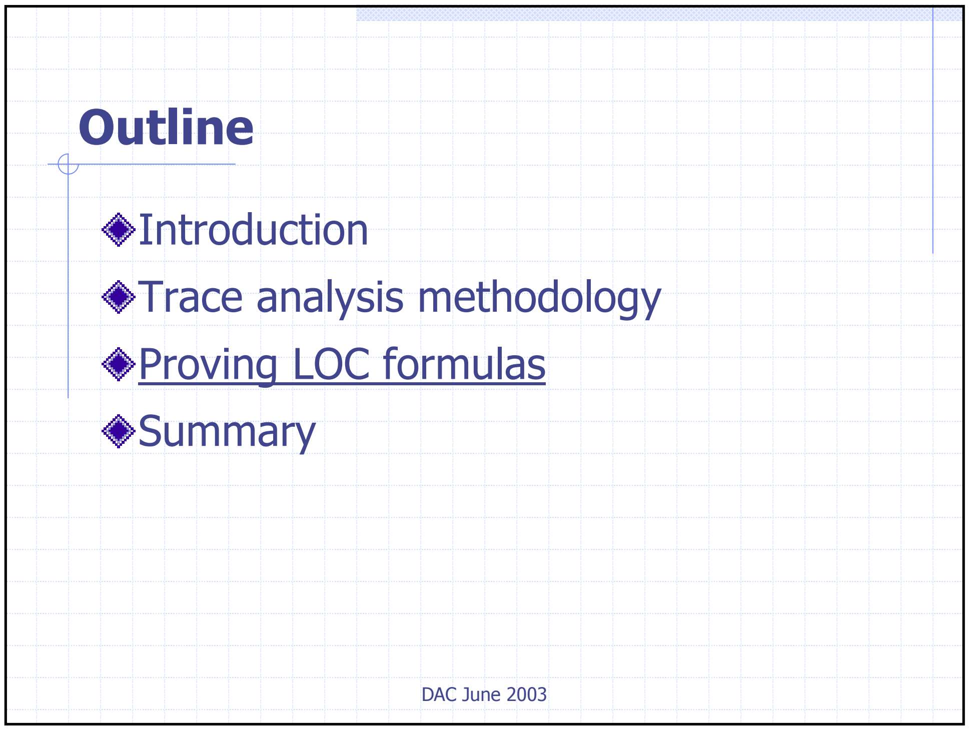
$$size(field\_start[2i+2])-size(field\_start[2i+1]) = size(field\_start[2i+1])-size(field\_start[2i])$$

3. "Latency between user control and actual size change <= 5."

$$field\_count(change\_size[i]) - field\_count(read\_size[i]) <= 5$$

Trace Checking Results: With the trace of about 120,000 lines, all these three constraints are checked within 1 minute.

DAC June 2003



Outline

- Introduction
- Trace analysis methodology
- Proving LOC formulas
- Summary

DAC June 2003

Formal Verification Tools and Methods

- Model checkers, e.g. SPIN, SMV
- Check if a finite state system(model) satisfy some property
- Properties are expressed with temporal logics, e.g. LTL
- Limitation
  - – state explosion
  - – finite state

Formal Verification for LOC

- We define a subset of LOC that has finite-state equivalents
  - – represent the LOC formula with LTL
  - – use LTL model checking directly
  - – Example:

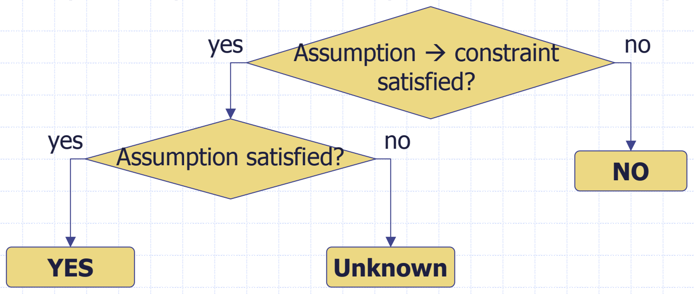
t(Display[i+1]) – t(Display[i]) = 10

$\Downarrow$

Display_occur → Display_t – Display_t_last = 10

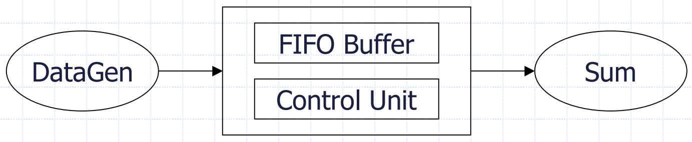
Formal Verification for LOC(cont'd)

- Other LOC formulas are beyond the finite-state domain, e.g. the latency constraint
  - make assumption to limit the system to finite-state domain
  - verify <u>assumption</u> and <u>assumption → constraint</u> separately



Verification Outcomes

Case Study – A FIFO Channel



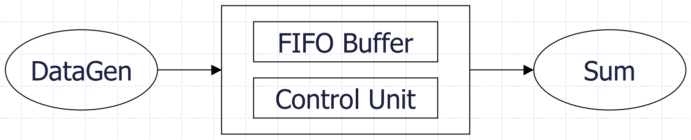
- Data consistency constraint on the channel:

  $data(DataGen\_write[i]) = data(Sum\_read[i])$

- Assumption – "*Sum_read* always follows *DataGen_write*, between a write and its corresponding read, only 30 more writes can be produced"

DAC June 2003

Case Study – A FIFO Channel (cont'd)



- Using the model checker SPIN, the assumption is verified in 1.5 hours and assumption → constraint is verified in 3 hours
- The FIFO channel is a library module
  - Repeated use
  - Small 600 lines of source code v.s. PIP (19,000 lines)

Summary

- LOC is useful and is different from LTL
- Automatic trace analysis
- Case studies with large designs and traces
- Formal verification approach