ELASTICITY AND PETRI NETS

Jordi Cortadella, Universitat Politecnica de Catalunya, Barcelona
Mike Kishinevsky, Intel Corp., Strategic CAD Labs, Hillsboro

Moore’s law

Is the GHz race over?

Many-Core is here

Source: Intel Corp.
**Elasticity**

- Tolerance to delay variability

- Different forms of elasticity
  - Asynchronous: no clock
  - Synchronous: variability synchronized with a clock

- In all forms of elasticity, token-based computations are performed (req/ack, valid/stop signals are used)

**Why this tutorial?**

- Digital circuits are complex concurrent systems
- Variability and power consumption are key critical aspects in deep submicron technologies
- Multi (many)-core systems will become a novel paradigm:
  - System design
  - Applications
  - Concurrent programming
- Theory of concurrency may play a relevant role in this new scenario

**Outline**

- Asynchronous elastic systems
  - The basics: circuits and elasticity
  - Synthesis of asynchronous circuits from Petri nets
  - Modern methods for the synthesis of large controllers
  - De-synchronization: from synchronous to asynchronous

- Synchronous elastic systems
  - Basics of synchronous elastic systems
  - Early evaluation and performance analysis
  - Optimization of elastic systems and their correctness
THE BASICS: CIRCUITS AND ELASTICITY

Outline

- Gates, latches and flip-flops. Combinational and sequential circuits.
- Basic concepts on asynchronous circuit design.
- Petri net models for asynchronous controllers. Signal Transition Graphs.

Boolean functions

Composed from logic gates

\[ x = \neg a \land b \]
\[ y = a \lor b \]
\[ z = \neg ((\neg a \land b) \lor (c \lor d)) \]

Memory elements: latches

Active high:
- \( En = 0 \) (opaque): \( Q = \text{prev}(Q) \)
- \( En = 1 \) (transparent): \( Q = D \)

Active low:
- \( En = 0 \) (opaque): \( Q = \text{prev}(Q) \)
- \( En = 1 \) (transparent): \( Q = D \)
Memory elements: flip-flop

 Finite-state automata

 Network of Computing Units

 Marked Graph Model
**Basic Concepts on Asynchronous Circuit Design**

### Outline
- What is an asynchronous circuit?
- Asynchronous communication
- Asynchronous design styles (Micropipelines)
- Asynchronous logic building blocks
- Control specification and implementation
- Delay models and classes of async circuits
- Channel-based design
- Why asynchronous circuits?

**Synchronous Circuit**
- Implicit (global) synchronization between blocks
- Clock period > Max Delay (CL + R)

**Asynchronous Circuit**
- Explicit (local) synchronization: Req / Ack handshakes
Motivation for asynchronous

- Asynchronous design is often unavoidable:
  - Asynchronous interfaces, arbiters etc.
- Modern clocking is multi-phase and distributed – and virtually 'asynchronous' (cf. GALS – next slide):
  - Mesachronous (clock travels together with data)
  - Local (possibly stretchable) clock generation
- Robust asynchronous design flow is coming (e.g. VLSI programming from Philips, Balsa from Univ. of Manchester, NCL from Theseus Logic ...)

Key Design Differences

- Synchronous logic design:
  - proceeds without taking timing correctness (hazards, signal ack-ing etc.) into account
  - Combinational logic and memory latches (registers) are built separately
  - Static timing analysis of CL is sufficient to determine the Max Delay (clock period)
  - Fixed set-up and hold conditions for latches

Globally Async Locally Sync (GALS)

Key Design Differences

- Asynchronous logic design:
  - Must ensure hazard-freedom, signal ack-ing, local timing constraints
  - Combinational logic and memory latches (registers) are often mixed in “complex gates”
  - Dynamic timing analysis of logic is needed to determine relative delays between paths
- To avoid complex issues, circuits may be built as Delay-insensitive and/or Speed-independent (as discussed later)
Synchronous communication

- Clock edges determine the time instants where data must be sampled
- Data wires may glitch between clock edges (set-up/hold times must be satisfied)
- Data are transmitted at a fixed rate (clock frequency)

Dual rail

- Two wires with L(low) and H(high) per bit
  - “LL” = “spacer”, “LH” = “0”, “HL” = “1”
- n-bit data communication requires 2n wires
- Each bit is self-timed
- Other delay-insensitive codes exist (e.g. k-of-n) and event-based signalling (choice criteria: pin and power efficiency)

Bundled data

- Validity signal
  - Similar to an aperiodic local clock
- n-bit data communication requires n+1 wires
- Data wires may glitch when no valid
- Signaling protocols
  - level sensitive (latch)
  - transition sensitive (register): 2-phase / 4-phase

Example: memory read cycle

- Transition signaling, 4-phase
Example: memory read cycle

- Transition signaling, 2-phase

Asynchronous modules

- Signaling protocol:
  reqin+ start+ [computation] done+ reqout+ ackout+ ackin+
  reqin- start- [reset] done- reqout- ackout- ackin-
  (more concurrency is also possible)

Asynchronous latches: C element

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Z+</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Static Logic Implementation
[van Berkel 91]

C-element: Other implementations

Weak inverter

Dynamic

Quasi-Static
Dual-rail logic

- A.t connected to C.t
- B.t connected to C.t
- A.f connected to C.f
- B.f connected to C.f

Valid behavior for monotonic environment

Differential cascode voltage switch logic

Example of dual-rail design

- Asynchronous dual-rail ripple-carry adder (A. Martin, 1991)
  - Critical delay is proportional to logN (N=number of bits)
  - 32-bit adder delay (1.6m MOSIS CMOS): 11 ns versus 40 ns for synchronous
  - Async cell transistor count = 34 versus synchronous = 28

Completion detection

- Dual-rail logic
- Completion detection tree
- done
Bundled-data logic blocks

Conventional logic + matched delay

Micropipelines (Sutherland 89)

Micropipeline (2-phase) control blocks

Request-Grant-Done (RGD) Arbiter

Call

Data-path / Control

\[ A_{\text{out}} \rightarrow \text{delay} \rightarrow C \rightarrow \text{delay} \rightarrow A_{\text{in}} \]

\[ L \rightarrow \text{logic} \rightarrow L \rightarrow \text{logic} \rightarrow L \rightarrow \text{logic} \rightarrow L \]

\[ R_{\text{in}} \rightarrow C \rightarrow \text{delay} \rightarrow C \rightarrow R_{\text{out}} \]

\[ L \rightarrow \text{logic} \rightarrow L \rightarrow \text{logic} \rightarrow L \rightarrow \text{logic} \rightarrow L \rightarrow \text{logic} \rightarrow L \rightarrow L \rightarrow L \rightarrow L \rightarrow L \rightarrow L \rightarrow L \rightarrow L \]

\[ R_{\text{in}} \rightarrow A_{\text{out}} \rightarrow A_{\text{out}} \rightarrow \text{CONTROL} \rightarrow R_{\text{out}} \rightarrow A_{\text{in}} \rightarrow A_{\text{in}} \]
Control specification

A simple filter: specification

\[ y := 0; \]
\[ \text{loop} \]
\[ x := \text{READ (IN)}; \]
\[ \text{WRITE (OUT, } (x+y)/2); \]
\[ y := x; \]
\[ \text{end loop} \]

A simple filter: block diagram

A simple filter: control spec.

- x and y are level-sensitive latches (transparent when R=1)
- + is a bundled-data adder (matched delay between \( R_s \) and \( A_s \))
- \( R_w \) indicates the validity of IN
- After \( A_{in} \), the environment is allowed to change IN
- \((R_{out}A_{out})\) control a level-sensitive latch at the output
A simple filter: control impl.

Taking delays into account

Delay assumptions:
- Environment: 3 time units
- Gates: 1 time unit

events: $x^+ \rightarrow x'^- \rightarrow y^+ \rightarrow z^+ \rightarrow z'^- \rightarrow x^- \rightarrow x'^+ \rightarrow z^- \rightarrow z'^+ \rightarrow y^- \rightarrow$

time: 3 4 5 6 7 9 10 12 13 14

Taking delays into account

Delay assumptions: unbounded delays

events: $x^+ \rightarrow x'^- \rightarrow y^+ \rightarrow z^+ \rightarrow x^- \rightarrow x'^+ \rightarrow y^- \rightarrow$ failure!

time: 3 4 5 6 9 10 11

WHY ASYNCHRONOUS?
Motivation (designer’s view)

- Modularity for system-on-chip design
  - Plug-and-play interconnectivity
- Average-case performance
  - No worst-case delay synchronization
- Many interfaces are asynchronous
  - Buses, networks, ...

Motivation (technology aspects)

- Low power
  - Automatic clock gating
- Electromagnetic compatibility
  - No peak currents around clock edges
- Security
  - No ‘electro–magnetic difference’ between logical ‘0’ and ‘1’ in dual rail code
- Robustness
  - High immunity to technology and environment variations (temperature, power supply, …)

Dissuasion

- Concurrent models for specification
  - CSP, Petri nets, …: no more FSMs
- Difficult to design
  - Hazards, synchronization
- Complex timing analysis
  - Difficult to estimate performance
- Difficult to test
  - No way to stop the clock
SYNTHESIS OF ASYNCHRONOUS
CIRCUITS FROM PETRI NETS

Delay models (II)

- Separation between functionality and timing [Muller]
  - Every gate has a zero-delay atomic evaluator (Boolean function)
  - A delay is associated to every output (gate delay model) or every input (wire delay model)
  - Delays can be:
    - Unbounded (arbitrary finite delays)
    - Bounded (within given min/max bounds)

Gate delay model

Wire delay model

Delay models (III)

- Gate delay model: delays in gates, no delays in wires

- Wire delay model: delays in gates and wires

Abstractions are necessary to define delay models manageable for design, synthesis and verification. Abstractions introduce optimistic and pessimistic simplifications that must be carefully taken into account.
Delay models (IV)

- **Speed-independent circuit:**
  hazard-free under the unbounded gate delay model

- **Delay-insensitive circuit:**
  hazard-free under the unbounded wire delay model

- **Quasi-delay-insensitive circuit:**
  delay-insensitive with some isochronic forks

**Speed-independent model**

- Pessimistic, since delays are typically bounded
- Optimistic, since it assumes isochronic forks (negligible skew wrt the receiving gate delays)
- Efficient synthesis methods exist

**Fundamental mode of operation**

[Huffman 1964]: The circuit/environment interact with two phases
   (1) The environment sends inputs to the circuit
   (2) The circuit computes the outputs and the state signals

The environment does not send new inputs until the circuit stabilizes

*Normal Fundamental Mode:* Only one input changes at each communication cycle

**Input/Output mode of operation**

- Computation and communication can overlap (according to some specified protocol)
- Event-based specification models are often used to describe the behavior (e.g., Petri nets).

This tutorial will cover the synthesis of speed-independent circuits that work under the I/O mode of operation and are specified using Petri nets.
Delay models for async. circuits

- Bounded delays (BD): realistic for gates and wires.
  - Technology mapping is easy, verification is difficult
- Speed independent (SI): Unbounded (pessimistic) delays for gates and “negligible” (optimistic) delays for wires.
  - Technology mapping is more difficult, verification is easy
- Delay insensitive (DI): Unbounded (pessimistic) delays for gates and wires.
  - DI class (built out of basic gates) is almost empty
- Quasi-delay insensitive (QDI): Delay insensitive except for critical wire forks (isochronic forks).
  - In practice it is the same as speed independent

Outline

- Overview of the synthesis flow
- Specification
- State graph and next-state functions
- State encoding
- Implementability conditions
- Speed-independent circuit
  - Complex gates
  - C-element architecture
- Review of some advanced topics

Book and synthesis tool

- J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev,
  Logic synthesis for asynchronous controllers and interfaces,
  Springer-Verlag, 2002

- petrify:
  http://wwwlsi.upc.es/petrify

Design flow

- Specification (STG)
  - Reachability analysis
- State Graph
  - State encoding
- SG with CSC
  - Boolean minimization
- Next-state functions
  - Logic decomposition
- Decomposed functions
  - Technology mapping
- Gate netlist
**Specification**

![Signal Transition Graph (STG)](image)

**Token flow**

![Token flow diagram](image)

**State graph**

![State graph](image)

**Next-state functions**

\[
x = \overline{z} \cdot (x + \overline{y})
\]

\[
y = z + x
\]

\[
z = x + \overline{y} \cdot \overline{z}
\]
Gate netlist

\[ x = \overline{z} \cdot (x + \overline{y}) \]

\[ y = z + x \]

\[ z = x + \overline{y} \cdot \overline{z} \]

Design flow

STG for the READ cycle

VME bus

Read Cycle
Choice: Read and Write cycles

Circuit synthesis

- Goal:
  - Derive a hazard-free circuit under a given delay model and mode of operation

Speed independence

- Delay model
  - Unbounded gate/environment delays
  - Certain wire delays shorter than certain paths in the circuit

- Conditions for implementability:
  - Consistency
  - Complete State Coding
  - Persistency
Design flow

Binary encoding of signals

STG for the READ cycle

Binary encoding of signals
Excitation / Quiescent Regions

Karnaugh map for LDS

Next-state function

Design flow
Concurrency reduction

State encoding conflicts

Signal Insertion
Design flow

Complex-gate implementation

\[ LDS = D + \text{csc} \]
\[ DTACK = D \]
\[ D = LDTACK \cdot \text{csc} \]
\[ \text{csc} = DSr \cdot (\text{csc} + LDTACK) \]

Implementability conditions

- Consistency
  - Rising and falling transitions of each signal alternate in any trace

- Complete state coding (CSC)
  - Next-state functions correctly defined

- Persistency
  - No event can be disabled by another event (unless they are both inputs)

Implementability conditions

- Consistency + CSC + persistency

- There exists a speed-independent circuit that implements the behavior of the STG

(under the assumption that any Boolean function can be implemented with one complex gate)
Persistency

\[
\begin{array}{ccc}
100 & a & 000 & c & 001 \\
\downarrow & b & \downarrow & b & \\
& a & & c & \\
\end{array}
\]

Speed independence ⇒ glitch-free output behavior under any delay

\[d = c + ad\]
Implementation with C elements

\[ S \rightarrow S^+ \rightarrow z^+ \rightarrow S^+ \rightarrow R^+ \rightarrow z^- \rightarrow R^- \rightarrow \cdots \]

- \( S \) (set) and \( R \) (reset) must be mutually exclusive
- \( S \) must cover \( ER(z^+) \) and must not intersect \( ER(z^-) \cup QR(z^-) \)
- \( R \) must cover \( ER(z^-) \) and must not intersect \( ER(z^+) \cup QR(z^+) \)

\[ \text{but ...} \]

Assume that \( R=\overline{ac} \) has an unbounded delay
Starting from state 0000 (\( R=1 \) and \( S=0 \)):

\[ a^+ ; R^- ; b^+ ; a^- ; c^+ ; S^+ ; d^+ \]

\( R^+ \) disabled (potential glitch)
**Speed-independent implementations**

- Implementability conditions
  - Consistency
  - Complete state coding
  - Persistency

- Circuit architectures
  - Complex (hazard-free) gates
  - C elements with monotonic covers
  - ...

**C-based implementations**

```
\[ \overline{abc} \rightarrow C \]
```

```
\[ \begin{array}{cccc}
  & a & b & c \\
\hline
0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 \\
2 & 1 & 1 & 1 \\
3 & 1 & 1 & 1 \\
\end{array} \]
```

**Synthesis exercise**

```
\[ \begin{array}{cccc}
  & x & y & z \\
\hline
0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 \\
2 & 1 & 1 & 1 \\
3 & 1 & 1 & 1 \\
\end{array} \]
```

Derive circuits for signals \( x \) and \( z \) (complex gates and monotonic covers)
Synthesis exercise

Logic decomposition: example

Synthesis exercise

Logic decomposition: example
Logic decomposition: example

Speed-independent Netlist

Adding timing assumptions
**Boolean domain**

<table>
<thead>
<tr>
<th>LDS=0</th>
<th>LDS=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
</tr>
</tbody>
</table>

One more DC vector for all signals

<table>
<thead>
<tr>
<th>LDS=0</th>
<th>LDS=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
</tr>
</tbody>
</table>

One state conflict is removed

**Netlist with one constraint**

(Diagram showing logic gates and signals)

**Netlist with one constraint**

(Diagram showing logic gates and signals with timing constraint)

TIMING CONSTRAINT
LDTACK- before DSr+
**Signal insertion**

- New signals need to be inserted to solve some synthesis problems (e.g., state encoding, logic decomposition).

- For each signal $s$, the events $s^+$ and $s^-$ must be inserted while preserving certain behavioral properties (consistency, persistency).

- Each new signal determines a new partition of states ($s=0$, $s=1$)

---

**From state graphs to Petri nets**

- A state graph may require transformations to meet certain properties (e.g., state encoding).

- The visualization of a state graph is not very informative. Event-based specifications explicitly represent the relations between events.

- Resort to the theory of regions
From state graphs to Petri nets

Region: [1]

Not a region: [3, 5]

From state graphs to Petri nets

Region: [0, 3, 5]

Theory of regions

- Region: all arcs of any event have the same relationship with the region (enter, exit, no cross).
- Minimal region: not included in any other region
- Pre-/post-region of an event: region such that the event exits/enters the region
- Property: excitation closure
  - The intersection of all pre-regions of an event is the excitation region of the event
Burst-Mode Specifications

Example: Burst-Mode (BM) Specification:

- Inputs in specified "input burst" can arrive in any order and at any time
- After all inputs arrive, generate "output burst"

Initial Values:
ABC = 000
YZ = 01

Note:
- Input bursts: must be non-empty
  (at least 1 input per burst)
- Output bursts: may be empty
  (0 or more outputs per burst)

Burst-Mode Specifications

"Extended Burst-Mode" (XBM):
[Yun/Dill ICCAD-93/95]

New Features:
1. "directed don't cares" (Rin*):
   allow concurrent inputs & outputs
2. "conditionals" (<Cnd>):
   allow "sampling" of level signals

Handles glitchy inputs, mixed sync/async inputs, etc.
Syntax-directed translation

2-Place “Ripple Register” (= FIFO) [van Berkel]

Tangram Program → Intermediate “Handshake Circuit”

Background: Channel-Based Communication

Components communicate using “4-phase handshaking”
- O1: initiates communication
- O2: completes communication

Channel impltn. => use 2 wires:
req => start operation
ack => operation done
(... can be extended to handle data)

A Larger Example

Intermediate “Handshake Circuit”

Handshake Components: Sequencer

2-Way Sequencer: activated on channel P; then activates 2 processes in sequence on channels A1 and A2

Goal: activate two sequential processes (i.e., operations)
Handshake Components: PAR Component

**PAR Component**: activated on channel P; then activates 2 processes in parallel on channels A1 and A2

**Goal**: activate two parallel processes

Intermediate Representation

```
procedure Buf1 (  
    input i: byte;  
    output o: byte) is  
local variable x : byte begin  
  loop begin  
    i -> x ;  
    o <- x  
  end  
end  
```

Conclusions

- STGs have a high expressiveness power at a low level of granularity (similar to FSMs for synchronous systems)
- Synthesis from STGs can be fully automated
- Synthesis tools often suffer from the state explosion problem (symbolic techniques are used)
- The theory of logic synthesis from STGs can be found in:

Structural methods for synthesis of large specifications

Thanks to Josep Carmona and Victor Khomenko

Outline

- Structural theory of Petri nets
  - Marking equation
  - Invariants

- ILP methods based on the marking equation
  - Detection of state encoding conflicts
  - Synthesis

- Methods based on unfoldings

State space explosion problem

- Even in bounded nets, the state space can be exponential on the size of the net
  - Concurrency (explosion of interleavings)

Event-based vs. State-based model

State Graph

Petri Net
Marking equation

\[ M' = M + Ax \]

Necessary reachability condition, but not sufficient.

Spurious markings

Checking Unique State Coding

\[ M_0 \xrightarrow{x} M_1 \xrightarrow{z = \{a+ b+ a- b-\}} M_2 \]

- \( M_1 \) and \( M_3 \) have the same binary code
  (\( z \) must be a complementary set of transitions)

- \( M_1 \) and \( M_2 \) must be different markings
  (they must differ in at least one place)
Checking Unique State Coding

\[ M_0 \xrightarrow{x} M_1 \xrightarrow{z} \{a+ b+ a- b-\} \rightarrow M_2 \]

**ILP formulation:**

\[
\begin{align*}
M_1 &= M_0 + Ax \\
M_2 &= M_1 + Az \\
bal(z) &\quad (x, z, M_1, M_2 \geq 0)
\end{align*}
\]

\[ bal(z) \equiv \forall a: \#(a+) - \#(a-) = 0 \]

Some experiments (USC)

| benchmark | |P| | |T| | signals | CPU(s) |
|-----------|-----|-----|-----|-----|
| PpWk(3,9) | 106 | 56  | 28  | 0.03 |
| PpWk(3,12)| 142 | 74  | 37  | 0.05 |
| PpWkCsc(3,9)| 108 | 56  | 28  | 0.67 |
| PpWkCsc(3,12)| 144 | 74  | 37  | 1.17 |
| PpArb(3,9) | 128 | 72  | 34  | 0.06 |
| PpArb(3,12)| 164 | 90  | 43  | 0.08 |
| PpArbCsc(3,9)| 131 | 72  | 34  | 1.05 |
| PpArbCsc(3,12)| 167 | 90  | 43  | 1.69 |

Checking Complete State Coding

**Enabling conditions in ILP**

\[ M \in \text{ER}(a+) : M(p_1)+M(p_2) \geq 2 \quad \lor \quad M(p_3)+M(p_4)+M(p_5) \geq 3 \]

\[ M \notin \text{ER}(a+) : M(p_1)+M(p_2) \leq 1 \quad \land \quad M(p_3)+M(p_4)+M(p_5) \leq 2 \]

(*formulation for safe nets only)

\[ a=0 \quad a=1 \]

\[ a- \quad QR(a-) \quad ER(a-) \]

\[ a+ \quad ER(a+) \quad QR(a+) \]

\[ n \text{ ILP problems must be solved} \]

\[ (n \text{ is the number of transitions with label } a^*) \]
Some experiments (CSC)

| benchmark    | |P| | |T| | |signals| |CLP| |SAT| |ILP|
|--------------|-----|-----|-----|-----|-----|-----|-----|-----|
| Tangram(3,2) | 142 | 92  | 38  | 0.01 | 0.01 | 1.08 |
| Tangram(4,3) | 321 | 202 | 83  | 0.06 | 0.04 | 9.00 |
| Art(10,9)    | 216 | 198 | 99  | 0.00 | 0.42 | 0.06 |
| Art(20,9)    | 436 | 398 | 199 | 5.00 | 10.35| 0.24 |
| Art(30,9)    | 656 | 598 | 299 | 38.02| 81.82| 0.56 |
| Art(40,9)    | 876 | 798 | 399 | 138.04| 264.57| 0.92 |
| Art(50,9)    | 1096| 998 | 499 | 377.00| 630.41| 1.46 |
| ArtCsc(10,9)| 752 | 630 | 315 | time | mem  | 27 m|
| ArtCsc(20,9)| 1532| 1270| 635 | time | mem  | 27 m|
| ArtCsc(30,9)| 2312| 1910| 955 | time | mem  | 1.5 h|
| ArtCsc(40,9)| 3092| 2550| 1275| time | mem  | 3.5 h|
| ArtCsc(50,9)| 3872| 3190| 1595| time | mem  | 7 h 

Synthesis

- Each signal can be implemented with a subset of the STG signals in the support (typically less than 10)

- Synthesis of a signal:
  - Project the STG onto the support signals (hide the rest)
  - After projection, the STG still has CSC for the signal
  - Use state-based methods on each projection

Synthesis example
Checking the support for a signal

Let $\Sigma$ be the set of signals and $\Sigma'$ a potential support for $a$.
Let $z'$ be the projection of $z$ onto $\Sigma'$.
$\Sigma'$ is a valid support for $a$ if the following model has no solution:

$$
\text{ILP formulation:}
M_1 = M_0 + Ax
M_2 = M_1 + Az
b = M_1 \in \text{ER}(a^*)
M_2 \notin \text{ER}(a^*)
x, z, M_1, M_2 \geq 0
$$

Algorithm to find the support

$z' := \{a\} \cup \{\text{trigger signals of } a\}$;
forever

$z'' := \text{ILP\_check\_support} (\text{STG}, a, z')$;
if $z'' = 0$ then return $z'$;
$z' := z' \cup \{\text{unbalanced signals in } z''\}$;
end_forever

Experiments (Support + Synthesis)

<table>
<thead>
<tr>
<th>benchmark</th>
<th>States</th>
<th></th>
<th>P</th>
<th></th>
<th></th>
<th>T</th>
<th></th>
<th></th>
<th>signals</th>
<th>Literals</th>
<th>CPU</th>
<th>ILP</th>
</tr>
</thead>
<tbody>
<tr>
<td>PpWhCsc(2,6)</td>
<td>8192</td>
<td>47</td>
<td>26</td>
<td>19</td>
<td>57</td>
<td>57</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PpWhCsc(2,9)</td>
<td>524288</td>
<td>71</td>
<td>38</td>
<td>19</td>
<td>87</td>
<td>87</td>
<td>49</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PpWhCsc(3,9)</td>
<td>2.7 x 10E7</td>
<td>106</td>
<td>56</td>
<td>28</td>
<td>?</td>
<td>130</td>
<td>mem</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PpWhCsc(3,12)</td>
<td>2.2 x 10E11</td>
<td>142</td>
<td>74</td>
<td>37</td>
<td>?</td>
<td>117</td>
<td>time</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PpArbCsc(2,6)</td>
<td>61440</td>
<td>62</td>
<td>36</td>
<td>17</td>
<td>77</td>
<td>77</td>
<td>21</td>
<td>83</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PpArbCsc(2,9)</td>
<td>3.9 x 10E6</td>
<td>110</td>
<td>60</td>
<td>29</td>
<td>107</td>
<td>107</td>
<td>185</td>
<td>59</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PpArbCsc(3,9)</td>
<td>3.3 x 10E9</td>
<td>131</td>
<td>72</td>
<td>34</td>
<td>163</td>
<td>165</td>
<td>10336</td>
<td>289</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PpArbCsc(3,12)</td>
<td>1.7 x 10E12</td>
<td>167</td>
<td>90</td>
<td>43</td>
<td>?</td>
<td>210</td>
<td>time</td>
<td>608</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TangramCsc(3,2)</td>
<td>426</td>
<td>142</td>
<td>92</td>
<td>38</td>
<td>97</td>
<td>103</td>
<td>56</td>
<td>146</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TangramCsc(4,3)</td>
<td>9258</td>
<td>321</td>
<td>202</td>
<td>83</td>
<td>?</td>
<td>247</td>
<td>mem</td>
<td>2 h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Petrify (Cortadella et al.):
  - State-based & technology mapping
  - BDD

Design flow

- structural encoding
  - remove internal signals and check CSC
- structural transformations
  - optimized STG
- support for a
  - projection
- support for b
  - logic synthesis (petrify)
- support for z
  - circuit for a
  - circuit for b
  - circuit for z
Synthesis example
Detection of conflicting states

- ILP
  [Carmona & Cortadella, ICCAD’03]
- SAT-UNFOLD
  [Khomenko et al., Fund. Informaticae]

Disambiguation by consistent signal insertion

Disambiguate the conflicting states by introducing a new signal $s$:

STG Insertion of signal $s$ must:
1. Solve conflict
2. Preserve consistency
3. Preserve persistency

(CSC + consistency + persistency = SI-circuit)
Implicit place

DEF1 (Behavior): The behavior of the net does not depend on the place.
DEF2 (Petri net): it never disables the firing of a transition.

Consistency

Consecutive firings of a signal must alternate

Implicit Places & Consistency

Disambiguation by consistent signal insertion

Theorem (Colom et al.)
Places y=0 and y=1 are implicit if and only if signal y is consistent

Insertion of s into the STG:
- s- will precede LDS+
- s+ will precede DTACK-
Main algorithm for solving CSC conflicts

```
while CSC conflits exist do
    (σ, σ') := Find traces connecting conflict
    (s=0, s=1) := Find implicit places
to break conflict
    Insert s+/s- transitions connected to
    (s=0) or (s=1)
endwhile
```

State space explosion problem

- Goal: avoid state enumeration to check implicitness of a place.
- Classical methods to avoid the explicit state space enumeration:
  - Linear Algebra (LP/MILP)
  - Graph Theory
  - Symbolic representation (BDDs)
  - Partial order (Unfoldings)

LP model to check place implicitness

A place \( p \) is implicit if the following LP model is infeasible, where \( P' = P - \{p\} \):

- LP formulation:
  \[
  M_0 + Ax = M
  \]
  \[
  M[P'] - F[P', p\bullet] \cdot s \geq 0
  \]
  \[
  M[p] - F[p, p\bullet] \cdot s < 0
  \]
  \[
  s \cdot 1 = 1
  \]
  \[
  x, M, s \geq 0
  \]

LP model to check place implicitness

A place \( p \) is implicit if the following LP model is infeasible, where \( P' = P - \{p\} \):

- LP formulation:
  \[
  M_0 + Ax = M
  \]
  \[
  M[P'] - F[P', p\bullet] \cdot s \geq 0
  \]
  \[
  M[p] - F[p, p\bullet] \cdot s < 0
  \]
  \[
  s \cdot 1 = 1
  \]
  \[
  x, M, s \geq 0
  \]

[Silva et al.]

LP model to check place implicitness

A place \( p \) is implicit if \( M_0[p] \) is greater than or equal to the optimal value of the following LP, where \( P' = P - \{p\} \):

- LP formulation:
  \[
  \min \ y \cdot M_0
  \]
  \[
  y \cdot A[P', T] \leq A[p, T]
  \]
  \[
  y \cdot F[P', p\bullet] \geq F[p, p\bullet]
  \]
  \[
  y \geq 0
  \]

[Silva et al.]
MILP model to insert a implicit place

\[ \begin{align*}
A' & \\
A & \\
p
\end{align*} \]

**MILP formulation:**

\[
\begin{align*}
\text{min } & \ y \cdot M_0 \\
y \cdot A'[p,T] & \leq A'[p,T] \\
y \cdot F'[p', p*] & \geq F[p, p*] \\
y & \geq 0 \\
p & \in \{-1,0,1\}
\end{align*}
\]

MILP variables: \( y, p \)

---

MILP model to find insertion points that disambiguate the conflict

**MILP formulation:**

- MILP “\( s=0 \) implicit”
- MILP “\( s=1 \) implicit”

\[
\begin{align*}
\#(\sigma_1, s) & = \#(\sigma_1, s^-) + 1 \\
\#(\sigma_2, s^-) & = \#(\sigma_2, s^+) + 1 \\
M_0[s=0] + M_0[s=1] & = 1
\end{align*}
\]

If there is a solution, rows in \( A' \) for \( s=0 \) and \( s=1 \) describe the insertion points (arcs in the net)

---

Number of inserted encoding signals

Benchmarks from [Cortadella et al., IEEE TCAD'97]

Number of literals (area)

Benchmarks from [Cortadella et al., IEEE TCAD'97]
Experimental results: large controllers

<table>
<thead>
<tr>
<th>example</th>
<th>Places</th>
<th>Trans</th>
<th>Signals</th>
<th>CPU(min)</th>
<th>#sig</th>
<th>Lits</th>
<th>HDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Art(10,9)</td>
<td>216</td>
<td>198</td>
<td>99</td>
<td>3.5</td>
<td>28</td>
<td>305</td>
<td>---</td>
</tr>
<tr>
<td>Art(20,9)</td>
<td>436</td>
<td>398</td>
<td>199</td>
<td>73.0</td>
<td>57</td>
<td>629</td>
<td>---</td>
</tr>
<tr>
<td>PpWk(3,12)</td>
<td>142</td>
<td>74</td>
<td>37</td>
<td>1.0</td>
<td>3</td>
<td>190</td>
<td>---</td>
</tr>
<tr>
<td>PpArb(3,12)</td>
<td>164</td>
<td>90</td>
<td>43</td>
<td>11.5</td>
<td>2</td>
<td>206</td>
<td>---</td>
</tr>
<tr>
<td>Var(9,5)</td>
<td>302</td>
<td>338</td>
<td>150</td>
<td>6.4</td>
<td>24</td>
<td>613</td>
<td>---</td>
</tr>
<tr>
<td>Var(12,1)</td>
<td>368</td>
<td>394</td>
<td>183</td>
<td>12.7</td>
<td>27</td>
<td>445</td>
<td>---</td>
</tr>
<tr>
<td>Par(12)</td>
<td>63</td>
<td>52</td>
<td>52</td>
<td>0.2</td>
<td>12</td>
<td>101</td>
<td>253</td>
</tr>
<tr>
<td>SeqPar(21,10)</td>
<td>160</td>
<td>128</td>
<td>64</td>
<td>2.2</td>
<td>23</td>
<td>269</td>
<td>398</td>
</tr>
<tr>
<td>SPM(7,16,18)</td>
<td>192</td>
<td>394</td>
<td>60</td>
<td>11.8</td>
<td>17</td>
<td>237</td>
<td>640</td>
</tr>
</tbody>
</table>

Synthesis with structural methods from [Carmona & Cortadella, ICCAD'03]

SYNTHESIS USING UNFOLDINGS

Work by Khomenko, Koutny and Yakovlev

It doesn’t always work ...

Behaviorally equivalent

Occurrence nets

Petri net

Occurrence net

Min-place
Occurrence nets

- The occurrence net of a PN $\mathcal{N}$ is a labelled (with names of the places and transitions of $\mathcal{N}$) net (possibly infinite!) which is:
  - Acyclic
  - Contains no backward conflicts (1)
  - No transition is in self-conflict (2)
  - No twin transitions (3)
  - Finitely preceded (4)

Unfolding of a PN

- The unfolding of Petri net $\mathcal{N}$ is a maximal labelled occurrence net (up to isomorphism) that preserves:
  - one-to-one correspondence (bijection) between the predecessors and successors of transitions with those in the original net
  - bijection between min places and the initial marking elements (which is multi-set)
Petri net and its unfolding

Prehistory (local configuration) of the transition instance

Final cut of prehistory and its marking (final state)

Truncation of unfolding

- At some point of unfolding the process begins to repeat parts of the net that have already been instantiated
- In many cases this also repeats the markings in the form of cuts
- The process can be stopped in every such situation
- Transitions which generate repeated cuts are called cut-off points or simply cut-offs
- The unfolding truncated by cut-off is called prefix

Cutoff transitions

Example: CSC Conflict
Example: enforcing CSC

Unfoldings

- Alleviate the state space explosion problem
- More visual than state graphs
- Proven efficient for model checking
- Quite complicated theory
- Not sufficiently investigated
- Relatively few algorithms

Translation Into a SAT Problem

\[ \text{Conf}(\text{conf'} \land \text{Conf}(\text{conf''})) \land \]
\[ \text{Code}(\text{conf'},... , \text{val}) \land \text{Code}(\text{conf'',... , val}) \land \]
\[ \text{Out}(\text{conf'},..., \text{out'}) \land \text{Out}(\text{conf'',..., out''}) \land \]
\[ \text{out'} \neq \text{out''} \]

Configuration constraint: \( \text{conf'} \) and \( \text{conf''} \) are configurations
Encoding constraint: \( \text{Code}(\text{conf'}) = \text{Code}(\text{conf''}) \)
Separating constraint: \( \text{Out}(\text{conf'}) \neq \text{Out}(\text{conf''}) \)
Configuration constraint

\[ \bigwedge_{f \in (\cdot, e)} (\text{conf}_e \rightarrow \text{conf}_f) \]

(no conflicts)

\[ \bigwedge_{f \in (e) \setminus \{e\}} \neg(\text{conf}_e \land \text{conf}_f) \]

Tracing the value of a signal

Computing the signals’ values

\[ \text{cut}_b \Leftrightarrow \text{conf}_e \land \bigwedge_{f \in b^+} \neg\text{conf}_f \]

\[ \text{val}_z \Leftrightarrow \bigvee_{h(b) = p_{z=1}} \text{cut}_b \]

Computing the enabled outputs

\[ \text{en}_e \Leftrightarrow \bigwedge_{f \in (\cdot, e)} \text{conf}_f \land \bigwedge_{f \in (e)} \neg\text{conf}_f \]

\[ \text{out}_z \Leftrightarrow \bigvee_{h(e) = z^0} \text{en}_e \]
Analysis of the Method

- A lot of clauses of length 2 – good for BCP
- The method can be generalized to other coding properties, e.g. USC and normalcy
- The method can be generalized to nets with dummy transitions
- Further optimization is possible for certain net subclasses, e.g. unique-choice nets

Synthesis using unfoldings

for each output signal $z$

compute (minimal) supports of $z$

for each ‘promising’ support $X$

calculate the projection of the set of reachable encodings onto $X$
sorting them according to the corresponding values of $\text{Nxt}_z$
apply Boolean minimization to the obtained ON- and OFF-sets
choose the best implementation of $z$

Experimental Results

- Unfoldings of STGs are almost always small in practice and thus well-suited for synthesis
- Huge memory savings
- Dramatic speedups
- Every valid speed-independent solution can be obtained using this method, so no loss of quality
- We can trade off quality for speed (e.g. consider only minimal supports): in our experiments, the solutions are the same as Petrify’s (up to Boolean minimization)
- Multiple implementations produced

ILP vs. Unfoldings

- ILP:
  - Efficient in runtime
  - Incomplete (spurious markings)
- Unfoldings:
  - Efficient in runtime (but slower than ILP)
  - Complete
- Both methods give synthesis results similar to state-based methods
Direct synthesis
(Varshavsky’s Approach)

Direct synthesis

Direct synthesis
De-synchronization: from synchronous to asynchronous

Outline

- What is de-synchronization?
- Behavioral equivalence
- 4-phase protocols for de-synchronization
- Concurrency
- Correctness
- An example


Synchronous circuit
De-synchronization

Distributed controllers substitute the clock network

The data path remains intact!

Design flow

- Think synchronous
- Design synchronous: one clock and edge-triggered flip-flops
- De-synchronize (automatically)
- Run it asynchronously

Prior work

- Micropipelines (Sutherland, 1989)
- Local generation of clocks
  - Varshavsky et al., 1995
  - Kol and Ginosar, 1996
- Theseus Logic (Ligthart et al., 2000)
  - Commercial HDL synthesis tools
  - Direct translation and special registers
- Phased logic (Linder and Harden, 1996)
  - Reese, Thornton, Traver, 2003
    - Conceptually similar
    - Different handshake protocol (2 phase vs. 4 phase)
Automatic de-synchronization

- Devise an automatic method for de-synchronization
- Identify a subclass of synchronous circuits suitable for de-synchronization
- Formally prove correctness

Outline

- What is de-synchronization?
- Behavioral equivalence
- 4-phase protocols for de-synchronization
- Concurrency
- Correctness
- An example

Synchronous flow
De-synchronized flow

Flow equivalence

[Guernic, Talpin, Lann, 2003]
Flow equivalence

Outline

- What is de-synchronization?
- Behavioral equivalence
- 4-phase protocols for de-synchronization
- Concurrency
- Correctness
- An example
A latch cannot read another data item until the successor has captured the current one.

A latch cannot become opaque before having captured the data item from its predecessor.
Outline

- What is de-synchronization ?
- Behavioral equivalence
- 4-phase protocols for de-synchronization
- Concurrency
- Correctness
- An example

Can we increase concurrency ?

not flow-equivalent
Can we reduce concurrency? How much?

(8 states)

(6 states)

(5 states)

(4 states)

fully decoupled (Furber & Day)

semi-decoupled (Furber & Day)

simple 4-phase

non-overlapping
4-phase latch controllers


Implementation note: Lt=0 (transparent), Lt=1 (opaque)

Simple 4-phase controller
4-phase latch controllers

4-phase latch controllers (state graphs)
Outline

- What is de-synchronization?
- Behavioral equivalence
- 4-phase protocols for de-synchronization
- Concurrency
- Correctness
- An example

Which protocols are valid for de-synchronization?
**Theorem:**
the de-synchronization protocol preserves flow-equivalence

**Proof:** by induction on the length of the traces

Induction hypothesis: same latch values at reset
Induction step: same values at cycle $i \Rightarrow$ same values at cycle $i+1$

**Theorem:**
any reduction in concurrency preserves flow-equivalence

Any hybrid approach preserves flow-equivalence!
Liveness

- Preservation of flow-equivalence:
  
  *all the generated traces are equivalent*

- Are all traces generated?  
  (Is the marked graph live?)

  *Not always!*

Flow-equivalence is preserved, … but …

Semi-decoupled 4-phase handshake protocol

Liveness: all cycles have at least one token [Commoner 1971]
Results about liveness

- At least three latches in a ring are required with only one data token circulating
  [Muller 1962]

- Theorem (this paper):
  any hybrid combination of protocols is live if the simple 4-phase protocol is not used

  Proof: any cycle has at least one token

Outline

- What is de-synchronization?
- Behavioral equivalence
- 4-phase protocols for de-synchronization
- Concurrency
- Correctness
- An example
Async DLX block diagram

Discussion

- The de-synchronization model provides an abstraction of the timing behavior

- All numbers are after Placement & Routing
- Total of 1500 flip-flops, 3000 latches
- DE-SYNC design includes 5 controllers, each driving 2 clock trees
- Power numbers include the clock tree
- Technology: UCM/Virtual Silicon 0.18 µm
Conclusions

- EDA tools require a *formal support* (they must work for *all* circuits)
- A complete characterization of 4-phase protocols has been presented (partial order based on concurrency)
- Design flow developed at Cadence Berkeley Labs
  - Automated from gate netlist
  - Static timing analysis to derive matched delays
  - Constrained P&R to meet timing constraints
Part 2: Synchronous Elastic Systems

Jordi Cortadella and Mike Kishinevsky

Synchronous elastic systems also called
– Latency tolerant systems or
– Latency insensitive systems

We use term “synchronous elastic” since better linked to asynchronous elastic

Agenda of Part 2

I. Basics of elastic systems

II. Early evaluation and performance analysis

III. Optimization of elastic systems and their correctness

What and Why

Intuition

How to design elastic systems

Converting synchronous system to elastic

Micro-arch opportunities

Marked Graph models

Performance evaluation
Synchronous Stream of Data

Clock cycle

Token (of data)

Synchronous Elastic Stream

Clock cycle

Bubble (no data)

Synchronous Circuit

Latency = 0

Synchronous Elastic Circuit

Latency can vary
Ordinary Synchronous System

\[ A \rightarrow C \rightarrow D \]

\[ B \rightarrow D \]

Changing latencies changes behavior

Synchronous Elastic (characteristic property)

\[ A^e \rightarrow C^e \]

\[ B^e \rightarrow D^e \]

Changing latencies does NOT change behavior

= time elasticity

Why

- Scalable
- Modular (Plug & Play)
- Better energy-delay trade-offs
  (design for typical case instead of worst case)
- New micro-architectural opportunities
  in digital design
- Not asynchronous: use existing design
  experience, CAD tools and flows

Example of elastic behavior
How to design elastic systems

We show an example of the implementation:
SELF = Synchronous Elastic Flow

Others are possible
Reminder: Memory elements. Flip-flop

![Flip-flop diagram]

Reminder: Clock cycle = two phases

![Clock cycle diagram]

Elastic channel protocol

![Elastic channel protocol diagram]
Elastic buffer keeps data while stop is in flight

Cannot be done with Single Edge Flops without double pumping
Can use latches inside Master-Slave

Communication channel

Long wires: slow transmission

Pipelined communication

What if the sender does not always send valid data?

The Valid bit

What if the receiver is not always ready?
The Stop bit

"Back-pressure"

Cyclic structures

One can build circuits with combinational cycles (constructive cycles by Berry), but synthesis and timing tools do not like them.

Example: pipelined linear communication chain with transparent latches

Shorthand notation (clock lines not shown)

Master and slave latches with independent control
SELF (linear communication)

SELF

Basic VS block

VS block + data-path latch = elastic HALF-buffer
Variable Latency Units (to be changed)

Elasticization

Synchronous  \[\rightarrow\]  Elastic
Elastic control layer
Generation of gated clocks

Micro-architectural opportunities

Circuit vs. μarchitectural cycles
Variable-latency cache hits

12-cycle miss

L2-cache

2-way associative
32KB
2-cycle hit

L1-cache

1-cycle hit

suggested by Joel Emer for ASIM experiment

Sequential access: if hit in first access \( L = 1 \), if not \( L = 2 \)
Trade-off: faster, or larger, or less power cache

Pseudo-associative
64KB
{2-3} cycle hit

L1-cache

1-cycle hit

Variable-latency RF with less ports

By-pass control

By-pass

Regfile

8 read ports
1 cycle read

ALU

Cache of recent computations

4-way superscalar or 4 threads assumed, 1 way shown

 LD  R2, A(R5)
 ADD  R1, R2, R3
 MUL  R4, R1, R6
 CMP  R4, #100

1 by-pass, 1 read port
1 by-pass, 1 read port
1 by-pass
Variable-latency RF with less ports

- By-pass control
- Regfile
  - 4 read ports
  - 1-2 cycle read
- ALU
  - Cache of recent computations
- LD R2, A(R5)
- ADD R1, R2, R3
- MUL R4, R1, R6
- CMP R4, #100

1 by-pass, 1 read port
1 by-pass, 1 read port
1 by-pass

Variable-latency ALUs

- In most of the ADD/SUB operations, only few LSBs are used. The rest are merely for sign extension.
- Use the idea of telescopic units:
  - 1-cycle addition for 16 bits and sign extension
  - 2 or more cycles for 64-bit additions (rare case)
  - maybe there is no need for CLA adders ...
  - or do all additions with 16-bit adders only

Variable Latency Adder

Benchmark “Patricia” from Media Bench

12 bits of an adder do 95% of additions
Power-delay [preliminary]

Pre-compute and tag size information

0 long data

1 short data

... and select functional unit according to the size of the data

Partitioned register file

or heterogeneous register files, register caches

Reminder:
Petri Nets and Marked Graphs
### Petri nets

- **Places (P)**
- **Transitions (T)**
- **Arcs**
- **Tokens**

### Petri nets. Token game

**Enabling Rule:**
- A transition is enabled if all its input places are marked
- Enabled transition can fire at any time

**Firing Rule:**
- One token is removed from every input place
- One token is added to every output place
- Change of marking is atomic

### Timed Petri nets

- **Assign a delay ($\delta$) to every transition**
  - An enabled transition fires $\delta$ time units after enabling

- Assume $\delta=1$

- $t = 0$
Timed Petri nets

- Assign a delay ($\delta$) to every transition
  - An enabled transition fires $\delta$ time units after enabling

Assume $\delta=1$ - $t = 1$

Marked Graph models of elastic systems

Timed Petri nets

- Assign a delay ($\delta$) to every transition
  - A transition with marked input places will fire after $\delta$ time units

Assume $\delta=1$ - $t = 2$

Modelling elastic control with Petri nets
Modelling elastic control with Petri nets

Hiding internal transitions of elastic buffers

Modelling elastic control with Marked Graphs

Forward (Valid or Request)

Backward (Stop or Acknowledgement)

Elastic control with Timed Marked Graphs.
Continuous time = asynchronous

Delays in time units

250ps

151ps
Elastic control with Timed Marked Graphs.
Discrete time = synchronous elastic

Latencies in clock cycles

Elastic control with Timed Marked Graphs.
Discrete time. Multi-cycle operation

Elastic control with Timed Marked Graphs.
Discrete time. Variable latency operation

E.g. discrete probabilistic distribution:
average latency $0.8 \times 1 + 0.2 \times 2 = 1.2$

Modeling forks and joins
Elastic Marked Graphs

- An Elastic Marked Graph (EMG) is a Timed MG such that for any arc \( a \) there exists a complementary arc \( a' \) satisfying the following condition:
  \[ *a = a' \text{ and } *a' = a \]

- Initial number of tokens on \( a \) and \( a' \) \((M_0(a) + M_0(a')) = \) capacity of the corresponding elastic buffer.

- Similar forms of “pipelined” Petri Nets and Marked Graphs have been previously used for modeling pipelining in HW and SW (e.g. Patil 1974; Tsirlin, Rosenblum 1982)

Performance analysis on Marked Graphs

\[ Th = \text{operations / cycle} \]

\[ Th = 3 / 7 \]
Performance

\[ T_h = \frac{3}{5} \]

Performance

\[ T_h = \frac{2}{5} \]

Performance

\[ T_h = \min(0.43, 0.6, 0.4) \]

Performance

\[ T_h = \min(0.43, 0.6, 0.4) \]

Minimum mean-weight cycle (Karp 1978)

Many efficient algorithms (some reviewed in Dasdan, Gupta 1998)
II

- Early evaluation
- Dual Marked Graphs
- Implementing early evaluation
- Performance analysis

Examples of early evaluation
Goal: Improve system performance and power

MULTIPLIER

\[
\text{if } a = 0 \text{ then } c := 0 \quad \text{-- don't wait for } b
\]

MULTIPLEXOR

\[
\begin{align*}
\text{if } s = T & \text{ then } c := a \quad \text{-- don't wait for } b \\
\text{else } & c := b \quad \text{-- don't wait for } a
\end{align*}
\]

Early evaluation

- Naïve solution: introduce choice places
  - issue tokens at choice node only into one (some) relevant path
  - problem: tokens can arrive to merge nodes out-of-order
  - later token can overpass the earlier one

- Solution: change enabling rule
  - early evaluation
  - issue negative tokens to input places without tokens,
    i.e. keep the same firing rule
  - Add symmetric sub-channels with negative tokens
  - Negative tokens kill positive tokens when meet

- Two related problems:
  Early evaluation and Exceptions (how to kill a data-token)

Example: next-PC calculation

- PC+4
- Branch target address
Related work

- Petri nets
  - Extensions to model OR causality
    [Kishinevsky et al. 1994, Yakovlev et al. 1996]

- Asynchronous systems
  - Reese et al 2002: Early evaluation
  - Brej 2003: Early evaluation with anti-tokens
  - Ampalan & Singh 2006: preemption using anti-tokens

---

Dual Marked Graphs

**Dual Marked Graph**

- **Marking:** Arcs (places) \( \rightarrow \mathbb{Z} \)
- Some nodes are labeled as early-enabling
- Enabling rules for a node:
  - Positive enabling: \( M(a) > 0 \) for every input arc
  - Early enabling (for early enabling nodes):
    \( M(a) > 0 \) for some input arcs
  - Negative enabling: \( M(a) < 0 \) for every output arc
- Firing rule: the same as in regular MG

---

Dual Marked Graphs

- Early enabling is only defined for nodes labeled as early-enabled. Models computations that can start before all the incoming data available
- Early enabling can be associated with an external guard that depends on data variables (e.g., a select signal of a multiplexor)
- In DMG actual enabling guards are abstracted away
- Anti-token generation: When an early enabled node fires, it generates anti-tokens in the predecessor arcs that had no tokens
- Anti-token propagation counterflow: When negative enabled node fires, it propagates the anti-tokens from the successor to the predecessor arcs
Dual Marked Graph model

Properties of DMGs

- **Firing invariant**: Let node \( n \) be simultaneously positive (or early) and negative enabled in marking \( M \). Let \( M_1 \) be the result of firing \( n \) from \( M \) due to positive (or early) enabling. Let \( M_2 \) be the result of firing \( n \) from \( M \) due to negative enabling. Then, \( M_1 = M_2 \).
- **Token preservation**: Let \( c \) be a cycle of a strongly connected DMG. For every reachable marking \( M \), \( M(c) = M_0(c) \).
- **Liveness**: A strongly connected DMG is live iff for every cycle \( c: M(c) > 0 \).
- **Repetitive behavior**: In a SC DMG: a firing sequence \( s \) from \( M \) leads to the same marking iff every node fires in \( s \) the same number of times.
- DMGs have properties similar to regular MGs

Passive anti-token

- Passive DMG = version of DMG without negative enabling
- Negative tokens can only be generated due to early enabling, but cannot propagate
- Let \( D \) be a DMG and \( D_p \) be a corresponding passive DMG. If environment (consumers) never generate negative tokens, then throughput \( (D) = \) throughput \( (D_p) \)
  - If capacity of input places for early enabling transitions is unlimited, then active anti-tokens do not improve performance
  - Active anti-tokens reduce activity in the data-path (good for power reduction)

Implementing early enabling
How to implement anti-tokens?

Positive tokens

Negative tokens

How to implement anti-tokens?

Positive tokens

Negative tokens

How to implement anti-tokens?

Valid

Valid

Stop

Stop

Controller for elastic buffer
Dual controller for elastic buffer

Dual fork/join and early join

Example

DLX processor model with slow bypass

<table>
<thead>
<tr>
<th>Evaluation</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>No early evaluation</td>
<td>0.277</td>
</tr>
<tr>
<td>Passive anti-tokens M2 → W</td>
<td>0.280</td>
</tr>
<tr>
<td>Passive anti-tokens F3 → W</td>
<td>0.387</td>
</tr>
<tr>
<td>Active anti-tokens</td>
<td>0.400</td>
</tr>
</tbody>
</table>

Throughput: $Th = \text{operations / cycle}$

Applying early evaluation on “Execution” and “Write-back”:

System performance

$Th=0.5$

$Th = 0.7$ (\(\alpha=0.3; \beta=0.3\))
Conclusions

- Early evaluation can increase performance beyond the min cycle ratio
- The duality between tokens and anti-tokens suggests a clean and effective implementation

Performance analysis with early evaluation

(joint work with Jorge Júlvez)

Reminder: Performance analysis of Marked graphs

\[ Th = \frac{\text{operations}}{\text{cycle}} = \text{number of firings per time unit} \]

The throughput is given by the minimum mean-weight cycle

\[ Th(A) = \frac{3}{7} \]
\[ Th(B) = \frac{3}{5} \]
\[ Th(C) = \frac{2}{5} \]

\[ Th = \min(Th(A), Th(B), Th(C)) = \frac{2}{5} = 0.4 \]

Efficient algorithms: (Karp 1978), (Dasdan,Gupta 1998)

Marked graphs. Performance analysis

The throughput can also be computed by means of linear programming

Average marking

\[ \overline{m}_p = \lim_{t \to \infty} \frac{1}{t} \int_0^t m_p(\tau) d\tau \]

Throughput

\[ th = \min(\overline{m}_{p1}, \overline{m}_{p2}) \]
\[ th = \min_{p} \overline{m}_p \]
Marked graphs. Performance analysis

The throughput can also be computed by means of linear programming

\[ th = \min_p \overline{m}_p \]

Max \( m_{i1} = 1 + t_a - t_b \)
Max \( m_{i2} = 0 + t_b - t_a \)
Max \( m_{i3} = 1 + t_b - t_a \)
Max \( m_{i4} = 0 + t_a - t_b \)
Max \( m_{i5} = 1 + t_a - t_b \)

Reachability

\( m_{i1} \leq m_{i2} \) // transition \( b \)
\( m_{i3} \leq m_{i4} \) // transition \( c \)
\( m_{i5} \leq m_{i5} \) // transition \( d \)
\( m_{i5} \leq \min(m_{i1}, m_{i3}) \) // transition \( a \)

Th = 0.5

[Campos, Chiola, Silva 1991]

Multi-guarded Dual Marked Graph

- Execution of transitions:
  - At the initial marking and each time \( t \) fires one of guards \( g_i \) from \( G(t) \) is non-deterministically selected
  - Selection is persistent (cannot change between firings of \( t \))
  - Accurate non-deterministic abstraction of the early evaluation conditions (e.g., multiplexor select signals)
  - \( t \) is enabled when for every place \( p \) in selected \( g_i \): \( M(p) > 0 \)
  - enabled \( t \) can fire (regular firing rule)
- Single-server semantics: no multiple-instances of the same transition can fire simultaneously
  - Abstraction for systems that communicate through FIFO channels

GMG = Multi-guarded Dual Marked Graph

- Refinement of passive DMGs
- Every node has a set of guards
- Every guard is a set of input arcs (places)
- Simple transition has one guard with all input places

Example:

![Diagram](image)

\( G(t4) = \{\{t1,t3\},\{t2,t3\}\} \)

Timed GMG

- Every transition is assigned a non-negative delay \( \delta(t) \)
  \( \delta(t) = 1 \) unless specified otherwise
- Every guard \( g \) of every guarded transition is assigned a strictly positive probability \( p(g) \) such that for every \( t \):

\[ \sum_{g \in G(t)} p(g) = 1 \]

- Guard selection for every transition is non-deterministic, but respects probabilities in the infinite executions
- Probabilities assumed to be independent (generous abstraction)
- Firing of transition \( t \) takes \( \delta(t) \) time units, from the time it becomes enabled until the firing is completed
Early evaluation

Timed GMG

{Places, Transitions, \( \delta \), Prob}  

Marked graphs with early evaluation  

\[ \text{Throughput?} \]

stochastic dynamic system

Alternatives to compute the throughput:
- Simulation
- Markov chain
- Linear programming

Throughput

\[ Th = \lim_{\tau \to \infty} \frac{\sigma(\tau)}{\tau} \]

\( \tau \) – time, \( \sigma(\tau) \) – firing vector

- This limit exists for every timed GMG
- It is the same for all transitions!
Markov chains

\[
\begin{pmatrix}
1 & 0 & 1 \\
1 & 0 & 1 \\
0 & 1 & 1 \\
0 & 1 & 1 \\
1 & 0 & 1
\end{pmatrix}
\]

Solve Markov chain:

\[
S_2 = S_1; \quad S_3 = (1 - \alpha)S_2; \quad S_1 + S_2 + S_3 = 1;
\]

\[
\text{Th} = S_1 + (1 - \alpha)S_2
\]

State explosion problem!

Linear programming formulation

**Average marking:** \( m_p \)

\[
\text{max th}
\]

\[
\bar{m} = m_0 + t_{in} - t_{out}
\]

reachability constraints

\[
\delta(t) \cdot \text{th} \leq \min(m_{p1}, m_{p2}) \text{ for all "classical" } t
\]

\[
\delta(t) \cdot \text{th} = \alpha \bar{m}_{p1} + (1 - \alpha) \bar{m}_{p2} \text{ for all "early" } t
\]

Linear programming. Example

\[
\text{max th}
\]

\[
\begin{align*}
m_{p1} &= 1 + b - t_s \\
m_{p2} &= 0 + t_b - t_s \\
m_{p3} &= 1 + t_b - t_s \\
m_{p4} &= 0 + t_c - t_s \\
m_{p5} &= 1 + t_c - t_s
\end{align*}
\]

\[
\text{th} \leq m_{p2} \\
\text{th} \leq m_{p4} \\
\text{th} \leq m_{p5}
\]

\[
\text{th} = \alpha m_{p1} + (1 - \alpha) m_{p3}
\]

\[
\text{Th} = (2 - \alpha) / (3 - \alpha)
\]

Averaging cycle throughput or cycle times does not work

\[
\text{Averaging throughput of individual cycles}
\]

\[
\text{Th}' = \alpha 1/2 + (1 - \alpha) 2/3 = (4 - \alpha) / 6
\]

\[
\text{Averaging effective cycle times of individual cycles}
\]

\[
1/\text{Th}'' = 2\alpha + (1 - \alpha) 3/2 = (3 + \alpha) / 2 \\
\text{Th}'' = 2/(3 + \alpha)
\]

\[
\text{Th} = (2 - \alpha) / (3 - \alpha)
\]
Example

Initial marking          Average (steady state) marking

Linear programming

In general the LP yields a throughput upper bound

Particular cases of exact throughput:

- No early joins (i.e. MGs)
- All joins are early evaluation

Throughput estimation

- Throughput estimation:
  1. Up = throughput obtained with LP
  2. Low = throughput without early enabling
  3. Th = (Up + Low)/2

Results

<table>
<thead>
<tr>
<th>Name</th>
<th>Nodes</th>
<th>Edges</th>
<th>MG-Low</th>
<th>Real</th>
<th>Up</th>
<th>ΔTh</th>
<th>Err</th>
</tr>
</thead>
<tbody>
<tr>
<td>s27</td>
<td>2</td>
<td>1</td>
<td>0.333</td>
<td>0.333</td>
<td>0.333</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>S208</td>
<td>0.500</td>
<td>0.571</td>
<td>0.594</td>
<td>14%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s298</td>
<td>0.091</td>
<td>0.120</td>
<td>0.129</td>
<td>32%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s349</td>
<td>0.333</td>
<td>0.333</td>
<td>0.333</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s382</td>
<td>0.250</td>
<td>0.284</td>
<td>0.294</td>
<td>14%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s386</td>
<td>0.400</td>
<td>0.400</td>
<td>0.400</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s400</td>
<td>0.400</td>
<td>0.438</td>
<td>0.470</td>
<td>10%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S444</td>
<td>0.200</td>
<td>0.261</td>
<td>0.287</td>
<td>31%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S516</td>
<td>0.167</td>
<td>0.167</td>
<td>0.167</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S596</td>
<td>0.333</td>
<td>0.333</td>
<td>0.333</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S641</td>
<td>0.333</td>
<td>0.393</td>
<td>0.432</td>
<td>18%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S713</td>
<td>0.250</td>
<td>0.333</td>
<td>0.333</td>
<td>33%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S820</td>
<td>0.143</td>
<td>0.201</td>
<td>0.230</td>
<td>41%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S832</td>
<td>0.286</td>
<td>0.310</td>
<td>0.342</td>
<td>8%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S953</td>
<td>0.286</td>
<td>0.295</td>
<td>0.333</td>
<td>3%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1423</td>
<td>0.100</td>
<td>0.184</td>
<td>0.189</td>
<td>84%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1488</td>
<td>0.188</td>
<td>0.236</td>
<td>0.271</td>
<td>26%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1494</td>
<td>0.154</td>
<td>0.222</td>
<td>0.277</td>
<td>44%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S5378</td>
<td>0.235</td>
<td>0.250</td>
<td>0.250</td>
<td>6%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s8234</td>
<td>0.200</td>
<td>0.219</td>
<td>0.248</td>
<td>10%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2-input gates
- a latch for each gate
- 75% tokens, 25% bubbles
- 25% muxes

Random select probability

Circuits from MCNC

- 2-input gates
- a latch for each gate
- 75% tokens, 25% bubbles
- 25% muxes
Summary

- Early evaluation to improve system throughput
  - Evaluate expressions as soon as possible
  - Generate antitokens to erase “don’t care” bits

- Analytical model to estimate the throughput
  - Useful for architectural exploration
  - Which muxes must have early evaluation?
  - Where do we put our by-passes?
  - Faster than simulation
  - Simulation can be used at later design stages

III

- Optimization
  - Slack matching and buffer sizing
  - Retiming and recycling
  - Clustering controllers

- Correctness
  - Theory of elastic machines
  - Formal verification

Buffer sizing

(joint work with Dmitry Bufistov)

How many bubbles do we need?

$$O(n^2) \text{ cycles}$$

$$O(n) \text{ cycles}$$
Throughput of an $n$-stage ring

- Number of tokens on a loop cannot be easily changed (inherent to the computation)
- Bubbles can always be added (as many as necessary), but may decrease throughput
- Buffer sizes can always be increased (provided forward latency of the buffer does not change)
- Tokens determine the maximum achievable throughput (assuming infinite buffer sizes)

Optimization techniques

- Buffer sizing: select optimal capacity of elastic buffers without increasing forward latency for propagating data-tokens
- Slack matching: insert additional empty elastic buffers
  - increases buffer capacity
  - but, typically increases forward latency as well
  - also called recycling in the context of synchronous elastic (latency-tolerant) designs

Buffer optimization

- Slack matching: insert additional empty elastic buffers
- Th = 5/8 = 0.625
Buffer optimization

Forward (Valid)

Backward (Stop)

Th = 5 / 8 = 0.625

Th = 3 / 5 = 0.6

Th = 4 / 7 = 0.57

Why?

Traffic jams in short branches of fork/join structures

Solution 1: slack matching/recycling

Make non-critical branches longer (be careful, forward latency increases)
Solution 2: increase buffer size

FIFO: n master + 1 slave latches

Slack matching vs. buffer capacity
- Not equivalent (slack matching cannot always achieve the forward latency, while buffer capacity can)
- Slack matching is a well-studied problem in asynchronous design
- Slack matching = inc buffer capacity + split

Buffer optimization
- Increase buffer capacity = Put token in backward edge
Buffer sizing

- Find min possible increase in buffer sizes such that the throughput is equal to the throughput of a system with infinite size buffers
- Combinatorial problem
- We found an exact ILP formulation, but ...
- ILP is exponential
- Can we do better (polynomial time)?

Buffer sizing is NP-complete

- NP-hardness: reduction of “min edges that cut all cycles in a digraph” to buffer sizing
- NP: Checking validity of solution can be done in polynomial time (e.g. Karp’s algorithm)
- Therefore,
  
  No polynomial algorithm exists, unless P = NP

LP performance model

(only forward (“Valid”) edges)

max achievable throughput (infinite-size buffers)

ILP model for buffer sizing

throughput with infinite buffers

extra capacity

\[ t_{in} \rightarrow p \rightarrow t_{out} \]

\[ t_{in} \rightarrow p \rightarrow p' \rightarrow t_{out} \]
### Table of results

<table>
<thead>
<tr>
<th>Circuit</th>
<th>V</th>
<th>E</th>
<th>Th</th>
<th>Max Th</th>
<th>ΔTok</th>
<th>CPU (sec)</th>
<th>Mem (Mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1423</td>
<td>484</td>
<td>942</td>
<td>0.33</td>
<td>0.33</td>
<td>0</td>
<td>&lt;1</td>
<td>81</td>
</tr>
<tr>
<td>s1144</td>
<td>321</td>
<td>1662</td>
<td>0.5</td>
<td>0.5</td>
<td>0</td>
<td>&lt;1</td>
<td>95</td>
</tr>
<tr>
<td>s1494</td>
<td>341</td>
<td>1775</td>
<td>0.5</td>
<td>0.5</td>
<td>0</td>
<td>1</td>
<td>108</td>
</tr>
<tr>
<td>s208</td>
<td>16</td>
<td>106</td>
<td>0.5</td>
<td>0.5</td>
<td>1</td>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>s27</td>
<td>31</td>
<td>78</td>
<td>0.5</td>
<td>0.5</td>
<td>1</td>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>s1494</td>
<td>823</td>
<td>7154</td>
<td>0.5</td>
<td>0.5</td>
<td>0</td>
<td>5</td>
<td>946</td>
</tr>
<tr>
<td>s249</td>
<td>199</td>
<td>241</td>
<td>0.6</td>
<td>0.6</td>
<td>3</td>
<td>&lt;1</td>
<td>7</td>
</tr>
<tr>
<td>s326</td>
<td>115</td>
<td>119</td>
<td>0.5</td>
<td>0.5</td>
<td>0</td>
<td>&lt;1</td>
<td>7</td>
</tr>
<tr>
<td>s400</td>
<td>165</td>
<td>273</td>
<td>0.33</td>
<td>0.33</td>
<td>0</td>
<td>&lt;1</td>
<td>7</td>
</tr>
<tr>
<td>s444</td>
<td>152</td>
<td>298</td>
<td>0.33</td>
<td>0.33</td>
<td>0</td>
<td>&lt;1</td>
<td>8</td>
</tr>
<tr>
<td>s510</td>
<td>149</td>
<td>571</td>
<td>0.5</td>
<td>0.5</td>
<td>0</td>
<td>&lt;1</td>
<td>16</td>
</tr>
<tr>
<td>s566</td>
<td>352</td>
<td>119</td>
<td>0.5</td>
<td>0.5</td>
<td>0</td>
<td>&lt;1</td>
<td>11</td>
</tr>
<tr>
<td>s537</td>
<td>113</td>
<td>2484</td>
<td>0.42</td>
<td>0.42</td>
<td>50</td>
<td>4708</td>
<td>500</td>
</tr>
<tr>
<td>s601</td>
<td>182</td>
<td>298</td>
<td>0.5</td>
<td>0.67</td>
<td>6</td>
<td>&lt;1</td>
<td>11</td>
</tr>
<tr>
<td>s713</td>
<td>208</td>
<td>350</td>
<td>0.42</td>
<td>0.5</td>
<td>1</td>
<td>&lt;1</td>
<td>15</td>
</tr>
<tr>
<td>s820</td>
<td>185</td>
<td>919</td>
<td>0.5</td>
<td>0.5</td>
<td>0</td>
<td>&lt;1</td>
<td>31</td>
</tr>
<tr>
<td>s872</td>
<td>191</td>
<td>972</td>
<td>0.5</td>
<td>0.5</td>
<td>0</td>
<td>&lt;1</td>
<td>34</td>
</tr>
<tr>
<td>s9234</td>
<td>1023</td>
<td>1992</td>
<td>0.25</td>
<td>0.25</td>
<td>0</td>
<td>&lt;1</td>
<td>350</td>
</tr>
<tr>
<td>s953</td>
<td>373</td>
<td>704</td>
<td>0.43</td>
<td>0.64</td>
<td>10*</td>
<td>&gt;21600</td>
<td>60</td>
</tr>
<tr>
<td>s38417</td>
<td>8315</td>
<td>16440</td>
<td>0.25</td>
<td>0.33</td>
<td>-</td>
<td>-</td>
<td>&gt;20Gb</td>
</tr>
</tbody>
</table>

* - Non-optimal integral solution with time limit 120 seconds

---

**Retiming and Recycling**

(joint work with Dmitry Bufistov and Sachin Sapatnekar)

---

**Retiming**

- Retiming: moving registers across combinational blocks or (equivalently) moving combinational blocks across registers
  - forward retiming
  - backward retiming

- Retiming in elastic systems
  - all registers participating in the retiming move should be labeled with the same number of data-tokens
  - use of negative tokens can remove the above constraint (will not discuss here)

**Recycling**

- Recycling: insert (or remove) empty elastic buffers (empty registers for short) on any edge
  - possible only in elastic systems
- We will ignore initialization and consider only steady state behavior
  - Initialization to an equivalent state almost always possible, but may require extra logic
Effective Cycle Time

- Cycle time: $c = \max$ (path delay between registers)
- Throughput: $\Theta = \min$ (tokens/cycle)
- Effective cycle: $C = c / \Theta$

\[
\begin{align*}
c &= 12 \\ \Theta &= 4/5 \\ C &= 15
\end{align*}
\]

R&R graph (RRG)

- Combinational block with a delay of 10 units
- Register (EB) with one data token
- Empty register (EB with no data tokens)

R&R is more powerful than retiming

Min delay retiming

\[
\begin{align*}
c &= 16
\end{align*}
\]

Min delay R&R

\[
\begin{align*}
c &= 15
\end{align*}
\]

Analogy between circuit retiming and reachability in MGs

- Retiming graph of a circuit = MG:
  - Combinational block = node
  - Connection = edge
  - Register = token
  - Firing rules = backward retiming rules: each time a node is retimed, registers are removed from the input edges and added to the output edges

- MGs: A live marking $M$ of an SCMG is reachable iff $M(\phi) = M(\phi)$ for every cycle $\phi$. Retiming interpretation:
  - $\Rightarrow$ valid retiming preserves the number of registers at each cycle
  - $\Leftarrow$ If an assignment of registers has the same number of registers at each cycle as the initial circuit, then the assignment is a valid retiming.
Analogy between circuit retiming and reachability in MGs

- Non-negative marking $M$ is reachable iff the marking equation holds:
  $$M = M_0 + A \times \sigma$$

- Retiming interpretation:
  - $M_0$ - initial assignment of registers to edges
  - $M$ - assignment after retiming
  - $A$ - retiming matrix
  - $\sigma$ - retiming vector

- Rename $M$ to $R$: $R = R_0 + A \times \sigma$

- ILP formulation ($A$ is totally unimodular. Polynomial problem)

Example of marking equation

$$e_{12} \begin{bmatrix} 0 \\ 1 \\ 2 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$M = M_0 + A \times \sigma$$

Valid R&R solutions

Any integer solution for $R$ and $R'$:

$$R \geq R' = R_0 + A \times \sigma$$

is a valid R&R solution

- $R'$ - retiming subset (registers with data-tokens)
- $R$ represents the R&R solution (registers with data-tokens or bubbles)
- $(R - R')$ registers with bubbles (recycling)

Combinational path constraints

$$\text{tin}(e) \geq \text{tout}(e') + \delta(u), \forall e' = (w, u)$$
$$\text{tout}(e) \geq \text{tin}(e) - \tau \times R(e)$$
$$\text{tout} \geq 0$$
$$\text{tin}(e) \leq \tau$$

$\tau$ – desired cycle time
$\tau'$ – original cycle time or any other constant $> \tau$
$\delta(u)$ – node $u$ delay
$R(e)$ – number of register on edge $e = (u, v)$
Register delays can be taken into account
Throughput constraints

Let $R$ be a valid R&R register assignment. There is a nonnegative real vector $\sigma$ that fulfills the above inequality iff $\Theta(R) > \Theta$.

$$R \leq (R_0 + A \times \sigma) / \Theta$$

ILP formulation for R&R

Given a cycle time $\tau$ and a throughput $\Theta$, $R$ is a valid R&R register assignment of an RRG $(N, E, R_0, d)$ with $\tau(R) < \tau$ and $\Theta(R) > \Theta$ iff there exists a feasible solution of the above ILP.

$$RR(\tau, \Theta) = \begin{cases} R \geq R_0 + A \times \sigma_1 \geq 0 \\ R \leq (R_0 + A \times \sigma_2) / \Theta \\ Path \_ constr(R, \tau) \\ R, \sigma_1 \in \text{int} \end{cases}$$

Min period R&R

Given an RRG and a throughput $\Theta > 0$, find a register assignment $R$ that minimizes the cycle time $\tau$ and has throughput $\Theta(R) \Rightarrow \Theta$.

$$\text{MIN \_ PER}(\Theta) = \min_{\text{subject to } RR(\tau, \Theta)} \tau$$

Max throughput R&R

Given an RRG and a cycle period $\tau$, find a register assignment $R$ with $\tau(R) \leq \tau$ that maximizes the throughput $\Theta(R)$.

$$\text{MAX \_ THR \_ NL}(\tau) = \max_{\text{subject to } RR(\tau, \Theta)} \Theta$$

This problem is not linear (and not convex): $\Theta$ is a variable of the model and the second constraint of $RR(\tau, \Theta)$ is not linear.

Use binary search on different $\Theta$. 
Size of the interval for binary search

- Binary search explores $[\Theta_L, \Theta_U]$.
  - $\Theta_L$ has feasible R&R solution,
  - $\Theta_U$ – does not
- What is the size of the interval not to miss an optimal solution?

$$|\Theta_L - \Theta_U| \geq 1 / (|R_0| + |E|)^2$$

$|R_0|$ – number of initial registers
$|E|$ – number of edges

---

Search for min effective cycle

Initialization: $\Theta(R_{\text{min}}) = d_{\text{max}} / \tau_{\text{rt}}$, where $d_{\text{max}}$ is the maximum delay of a node and $\tau_{\text{rt}}$ is the cycle period obtained by min-delay retiming.

MIN EFFECTIVE CYCLE TIME R&R

Given an RRG, find a register assignment $R_{\text{min}}$ with a minimal effective cycle time $C(R_{\text{min}})$.

Min effective cycle time R&R

Min effective cycle time R&R

Initialization:
- $\Theta := d_{\text{max}} / \tau_{\text{rt}}$
- $\epsilon := 1 / (|R_0| + |E|)$

while $\Theta < 1$
- $R := \text{MAX } \text{ECYC } \text{STEP}(\Theta)$
- if $C(R) < C$ then $C := C(R)$
- $\Theta := \Theta(R) + \epsilon$

return $C$

---

Results

- R&R can provide better effective cycle time than regular retiming if
  - long cycles and
  - unbalanced delays
- Useful for micro-architectural problems, not for well balanced circuit graphs
Clustering controllers

(joint work with Josep Carmona and Jorge Júlvez)

Sharing controllers and elastic buffers

Merging nodes in elastic MG

Mergeable transitions

- **Definition:** Transitions \( t_i \) and \( t_j \) of EMG \( G \) are called mergeable if \( \text{Th}(G) = \text{Th}(G') \), where \( G' \) obtained by merging \( t_i \) and \( t_j \) in \( G \)

- **Idea:** Merge transitions with the same critical average marking at their input arcs
  - If transitions are not critical, then explore slack at the non-critical input arcs: check if the same throughput can be achieved with critical average marking
Correctness and Verification

- Developed theory of elastic machines
- Verify correctness of any elastic implementation = check conformance with the definition of elastic machine
- All SELF controllers are verified for conformance
- Elasticization is correct-by-construction

Correctness (long story) = theory of Elastic Machines

(joint work with Sava Krstic and John O'Leary)

Systems

- A stream $a$ over a set $A$ is an infinite sequence $a[0], a[1], \ldots$ of elements of $A$
- $a \sim_{n} b$ iff $a$ and $b$ have a common prefix of length $n$
- $W$ is a set of wires
- $W$-behavior $\sigma$: a stream $\sigma.w$ for each $w \in W$
- $[W]$ = the set of all $W$-behaviors
- $\sim_{n}$ extends to $[W]$: $\sigma \sim_{n} \tau$ iff $(\forall w \in W) \sigma.w \sim_{n} \tau.w$
- A $W$-system is a subset of $[W]$
- Example: $\text{Conn}(X,Y) = \{ \sigma | \sigma.X = \sigma.Y \} \subseteq [(X,Y)]$
Operations

* Projection $\sigma \rightarrow \pi.V : [W] \rightarrow [V]$ defined for $V \subseteq W$
* $\text{hide}_V(S) = \{\sigma.(W-V) | \sigma \in S\} \subseteq [W-V]$
* $S_1 \cup S_2 = \{\sigma | \sigma.W_1 \in S_1 \land \sigma.W_2 \in S_2\} \subseteq [W_1 \cup W_2]$
* Networks of systems:

$$\langle S_1, \ldots, S_m | u_1 = v_2, \ldots, u_n = v_n \rangle = \text{hide}_{\langle u_1, \ldots, u_n, v_1, \ldots, v_n \rangle}(S_1 \cup \cdots \cup S_m \cup \text{Conn}(u_1, v_1) \cup \cdots \cup \text{Conn}(u_n, v_n))$$

$$= (A, B, C, D | 2 = 5, 4 = 7, 10 = 11, 8 = 9, 12 = 1)$$

Machines = abstract circuits

Definition An $(I, O)$-machine is an $(I \cup O)$-system given by a function $F: [I] \rightarrow [O]$ satisfying the causality property

$$(\forall \sigma, \sigma' \in [I]) (\forall k \geq 0) \quad \sigma \sim_k \sigma' \implies F(\sigma) \sim_k F(\sigma')$$

Outputs at the first $k$ cycles are determined by inputs at the first $k$ cycles.

When is a network a machine?

* Feedback

$$S : \begin{array}{ccc}
\text{I} & \text{z} & \text{W} \\
\text{V} & \\ & \text{W} \\
\end{array} \quad \langle S | v = w \rangle : \begin{array}{ccc}
\text{V} & \text{z} & \text{W} \\
\text{V} & \\ & \text{W} \\
\end{array}$$

* A network is a multiple feedback

* Must preclude “combinational cycles”—but what are they?

Sequential and combinational dependency

Definition An input-output pair $(u, v)$ is sequential if

$$(\forall \sigma, \sigma' \in [I]) \quad (\forall k \geq 0) \quad \sigma, u \sim_k \sigma', u \land (\forall x \neq u) \sigma, x \sim_k \sigma', x \implies F(\sigma).v \sim_k F(\sigma').v$$

Feedback Lemma If $(u, v)$ is sequential for a machine $S$, then $\langle S | u = v \rangle$ is a machine.
Detecting combinational loops

**Definition** $\Gamma(N)$: Vertices are wires of $N$; directed edges drawn for non-sequential wire pairs.

**Theorem** If $\Gamma(N)$ is acyclic, then $N$ is a machine.

---

**Liveness**

- Liveness

  $(\forall Y \in O) \quad S \models G(\text{min}_{tctY} \geq tctY \land tctY > tctY \Rightarrow F \text{valid}_Y)$

  $(\forall X \in I) \quad S \models G(\text{min}_{tctX} \geq tctX \Rightarrow F \text{stop}_X)$

- Serve only the most hungry channels:

  ![Diagram of serve only the most hungry channels]

- Liveness guarantees that all transfer behaviors $\omega.T.Z$ are infinite (in an "elastic environment")

  $\therefore$ The transfer system $[S^T] = \{ \omega.T | \omega \in S \cup \text{Env}_{I,O} \}$

---

**Elastic machine**

- **Input-output structure**
  - inputs: $I \cup \{ \text{valid}_X | X \in I \} \cup \{ \text{stop}_Y | Y \in O \}$
  - outputs: $O \cup \{ \text{valid}_Y | Y \in O \} \cup \{ \text{stop}_X | X \in I \}$

- **Persistence**
  - $S \models G(\text{valid}_Y \land \text{stop}_Y \Rightarrow (\text{valid}_Y)^+) \quad \text{for every } Y \in O$

- **Determinism**
  
  \( \forall \omega_1, \omega_2 \in S \) \quad $\omega_1^T I = \omega_2^T I \Rightarrow \omega_1^T O = \omega_2^T O$

**Definition** $S$ is an $[I,O]$-elastic machine if it has the input-output structure as described, and satisfies the persistence, liveness, and determinism conditions.

**Theorem** If $S$ is a $[I,O]$-elastic machine, then $S^T$ is an $(I,O)$-machine.

- $S$ is an elasticization of $M$ when $M = S^T$
Elastic networks

Suppose $S_1, \ldots, S_m$ are elastic machines.

\[ N = \langle S_1, \ldots, S_m \mid X_1 = Y_1, \ldots, X_n = Y_n \rangle \]

\[ \Delta \]

\[ (S_1, \ldots, S_m \mid X_i = Y_i, \text{valid}_i, \text{stop}_i = \text{stop}_i (1 \leq i \leq n)) \]

* Is $N$ an elastic machine?
* Do we have $N^T = \langle S^T_1, \ldots, S^T_m \mid X_1 = Y_1, \ldots, X_n = Y_n \rangle$?

---

Elastic feedback

\[ F = \langle S \parallel P = Q \rangle = \langle S \mid P = Q, \text{valid}_P = \text{valid}_Q, \text{stop}_P = \text{stop}_Q \rangle \]

**Definition** An i/o channel pair $(P, Q)$ sequential for $S$ if

\[ S \models G (\min_{tct_{i \in IO}} \geq \min_{tct_{j \in IO}} tct \land \min_{tct_{j \in IO}} \geq tct \Rightarrow F \text{valid}_Q) \]

and the graph $I'F$ is acyclic.

**Theorem** If the channel pair $(P, Q)$ is sequential for $F$, then

* the wire pair $(P, Q)$ is sequential for $S^T$
* $F$ is an elastic machine
* $F^T = \langle S^T \mid P = Q \rangle$

---

Elastic network theorem

* $N = \langle S_1, \ldots, S_m \mid X_1 = Y_1, \ldots, X_n = Y_n \rangle$

* $\delta_i$ : a sequentionality interface for $S_i$ ($\delta_i(Z)$ is a set of input wires “jointly sequential” wrt $Z$)

**Definition** $[\Delta(N)]$: Vertices are channels of $N$ (: $X_j$ and $Y_j$ are identified); a directed edge drawn for each pair $(P, Q) \in I_i \times O_i$ such that $P \notin \delta_i(Q)$.

* $N' = \langle S^T_1, \ldots, S^T_m \mid X_1 = Y_1, \ldots, X_n = Y_n \rangle$

**Theorem** If $\Delta(N)$ is acyclic, then $N$ is an elastic machine, $N'$ is a machine, and $N^T = N'$.

---

Inserting empty buffers

**Theorem** Suppose $N_1$ and $N_2$ are elastic networks obtainable from each other by insertion and deletion of empty elastic buffers. If one of $\Delta(N_1)$, $\Delta(N_2)$ is acyclic, then the other is acyclic too, and one has $N^T_1 = N^T_2$. 
Verification of elastic systems

(joint work with Syed Suhaib and Sava Krstic)

Implementation of Elastic Module

1. Verify Properties of Elastic Machine
2. Verify Data Transfer Properties

Problem

- Infinite domain transfer counters (tct)
- Model checking requires finite domain counters

Finite domain is sufficient

- Any implementation has finite sequential depth between any input channel and any output channel
- Model the tct variable as integers modulo \( (k+1) \) in some finite range \([0,k]\) sufficient to cover maximal sequential depth
- Reset the tct when range reaches \( k \), and restart from 0

How do you compute \( k \)?
Synchronous Slack

- Capacity: \( C(i,j) \) is defined as the maximum number of data storage elements between channels \( i \) and \( j \), where \( i \in I \) and \( j \in O \)

- For an \([I,O]\)-system \( S \), its synchronous slack \( \varepsilon = \min_{\{i,j\}} \{k: G(k \geq \max(|tct_i - tct_j|))\} \)

- \( \varepsilon \leq \max_{\{i,j\}} C(i,j) \)

Validating Data Correctness

- Symbolic terms and uninterpreted function
  - Proposed by Burch and Dill '94

- We employ similar procedure
  - Encode all possible terms
  - Combinational logic modeled as a single function

Modeling of Counters

- Use uninterpreted function

- Consider, e.g., a two input uninterpreted function
Model checking

- **SPIN Model Checker [Bell Labs]**
- **NuSMV Model Checker [IRST and CMU]**

Summary

- SELF gives a low cost implementation of elastic machines
- Functionality correct when latencies change
- New micro-architectural opportunities
- Compositional theory proving correctness
- Early evaluation - mechanism for performance and power optimization
- Retiming and recycling, buffer optimization and other optimization opportunities
- To read on this work: see list of references

Research directions

- How to specify elastic machines
  - Asynchronous specification (e.g., CSP) discretized asynchrony view
  - Elastic synchronous specification (extend Esterel, Lustre, PBS with controlled asynchrony)
- Compilers
- Improve bounds on analytical perf. analysis for early evaluation
- Formal methods for micro-architectural optimization
- R&R and buffer optimization for systems with early evaluation
- More on optimization for elastic machines

Some of Related work

- **Async**
  - Rings (T. Williams, J.Sparso)
  - Caltech CHP and stack-elasticity (A. Martin, S.Burns, R.Manohar et al.)
  - Micropipelines (J. Sutherland)
  - Many others
- **Latency insensitive design**
  - L. Carloni and a few follow-ups (large overhead)
  - C. Svensson (Linköping U.) - via-pipelining
- **Interlock pipelines**
  - H. Jacobson et al.
- **Desynchronization**
  - J. Cortadella et al.
  - V. Varshavsky
- **Performance analysis**
  - S.Burns
  - H. Hulgaard
  - C.Neelsen/M.Kahnevsky, etc.
- **Synchronous implementation of CSP**
  - J. O'Leary et al.
  - A. Peeters et al.
- **Telescopic units** – Benini et al.
- See a list of references