The 14th IEEE Real-Time and Embedded Technology and Applications Symposium, April 2008

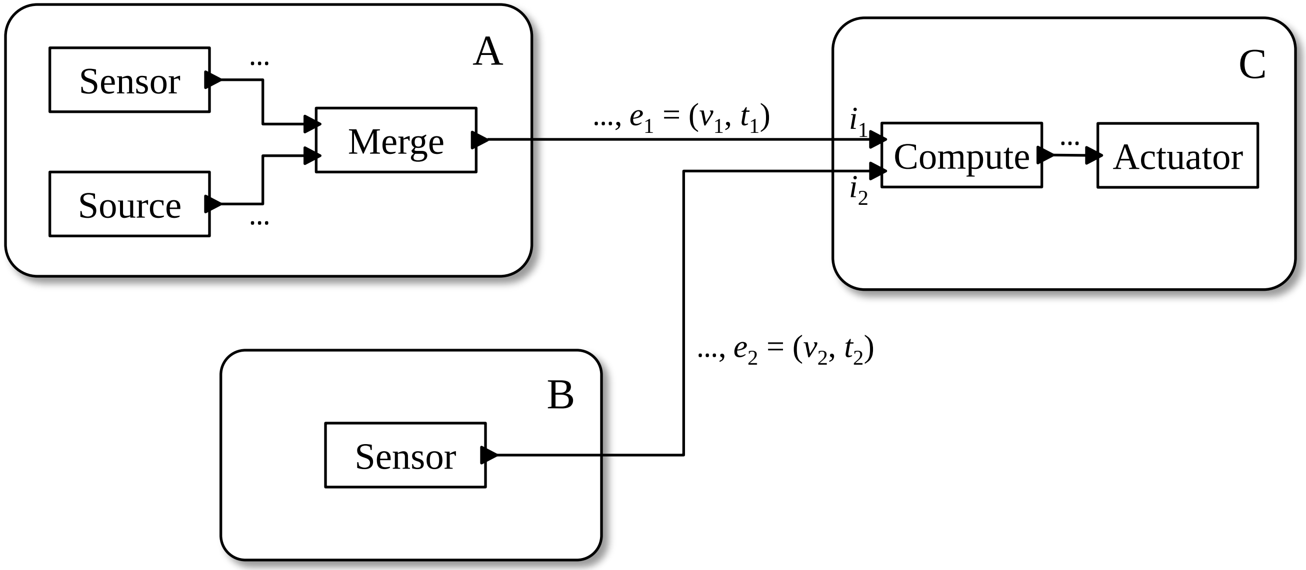
Real-Time Distributed Discrete-Event Execution with Fault Tolerance

Thomas Huining Feng and Edward A. Lee

Center for Hybrid and Embedded Software Systems
EECS, UC Berkeley

{tfeng, eal}@eecs.berkeley.edu

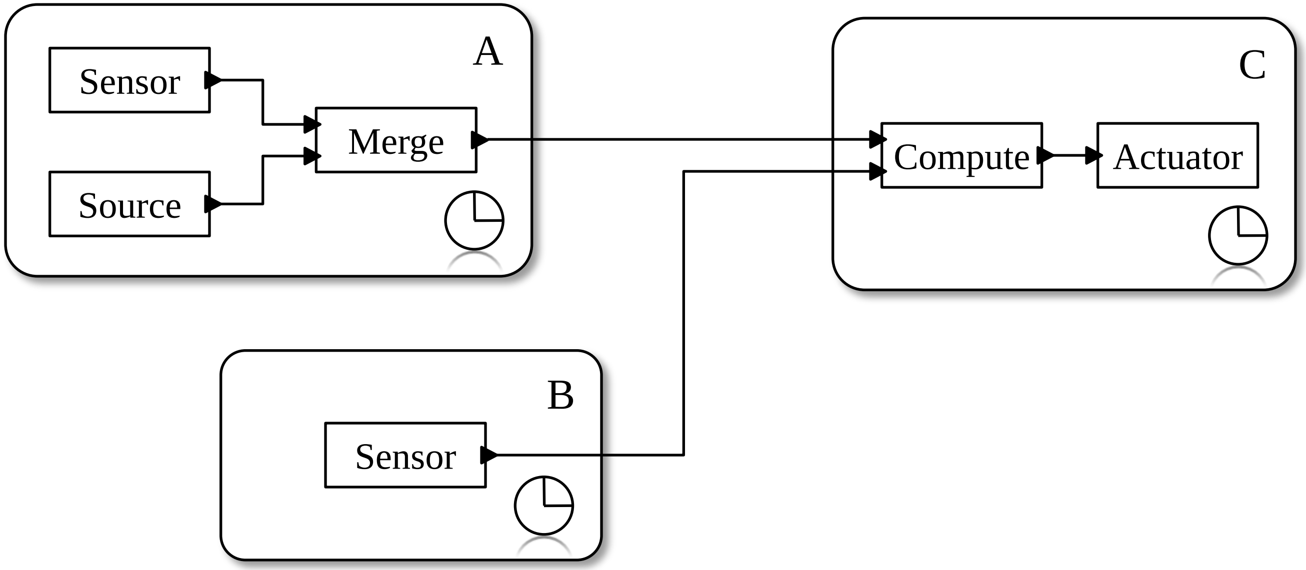
Distributed Discrete-Event Execution Strategy



- Execution strategy decides whether/when it is *safe to process* an input event.
- Conventional: Compute can process top event e_1 if e_2 has a greater time stamp.
- Null message $(null, t_2)$
  Cons: overhead, sensitive to faults, lack of real-time property

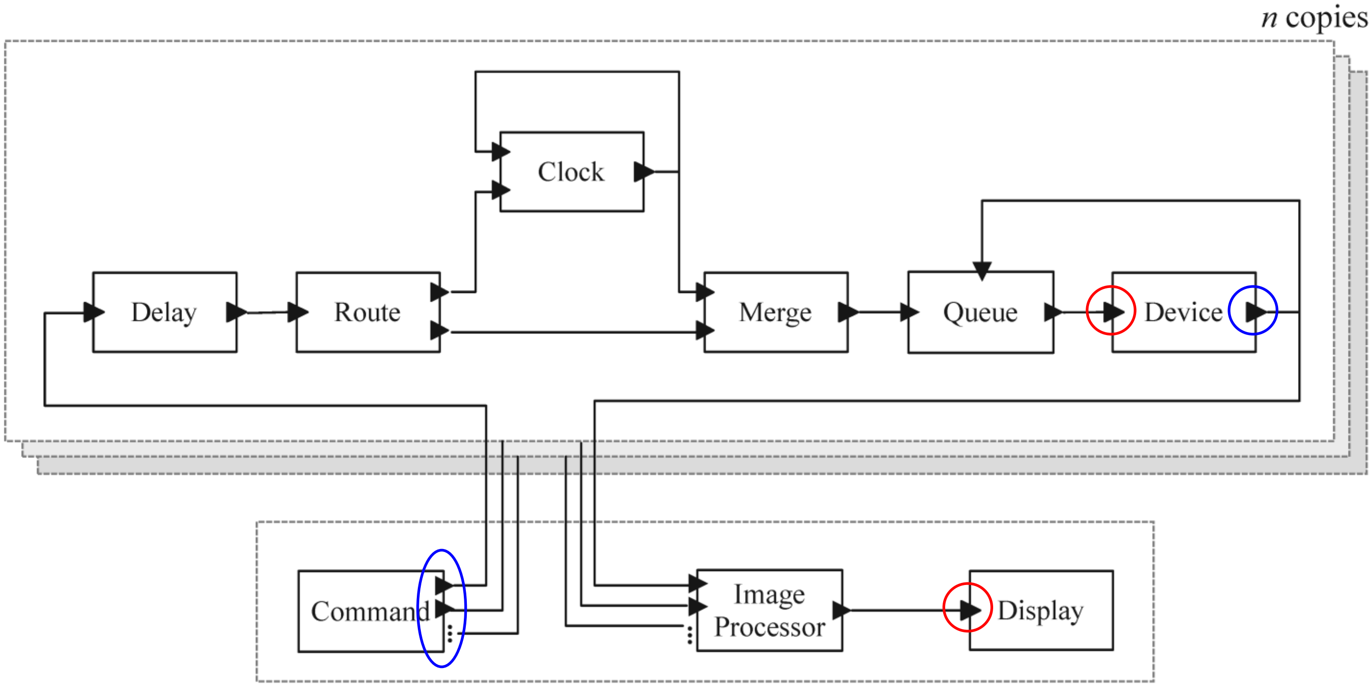
Overview of Our Approach

- Leverage time-synchronized platforms
- Eliminate null messages
- Potentially improves concurrency
- Decompose assertions of real-time properties
- Recover software components from faults



Reference Application: Distributed Cameras

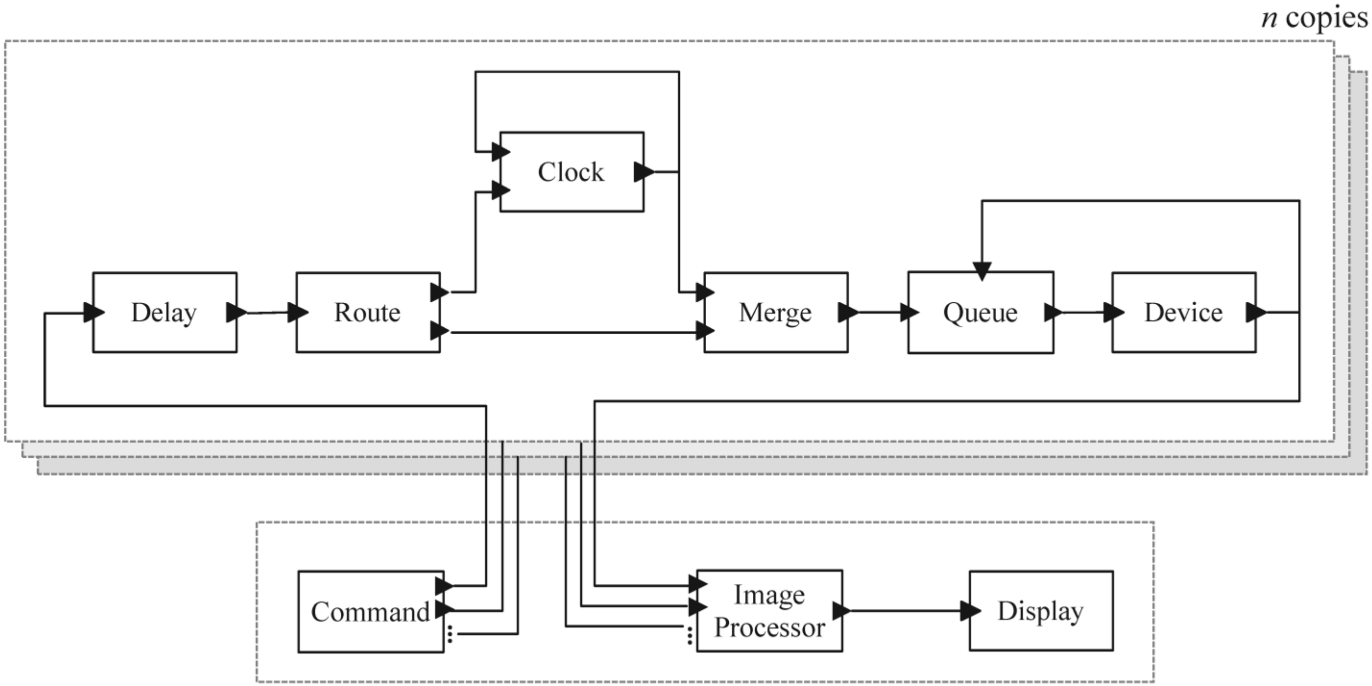
- *n* cameras located around a football field, all connected to a central computer.
- Events at blue ports satisfy $t \leq \tau$
  (t – time stamp of any event; τ – real time)
- Events at red ports satisfy $t \geq \tau$



Reference Application: Distributed Cameras

Problems to solve:

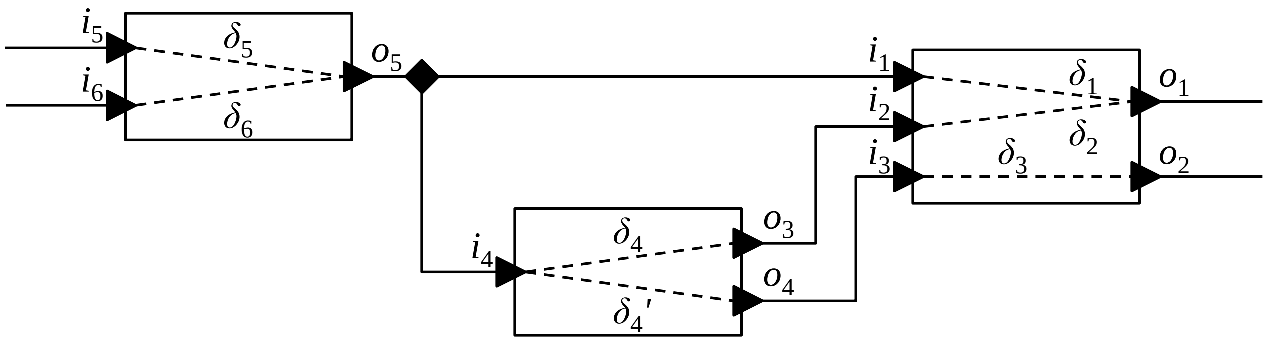
- Make event-processing decisions locally
- Guarantee timely command delivery to the Devices
- Guarantee real-time update at the Display
- Tolerate images loss or corruption at Image Processor



Minimum Model-Time Delay δ

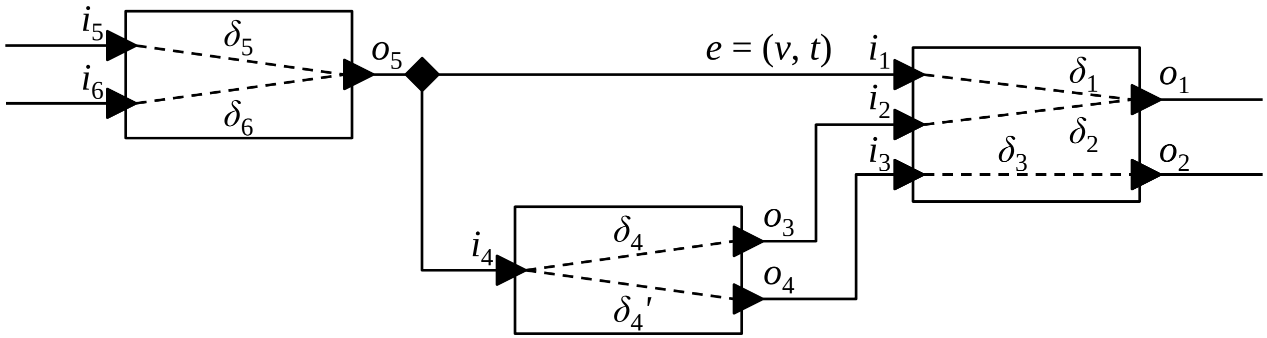
$\delta : P \times P \to R^+ \cup \{\infty\}$ returns the minimum model-time delay between any two ports.

(P – set of ports; R^+ – set of non-negative reals.)



Example: $\delta(i_5, o_1) = \min\{\delta_5+\delta_1, \delta_5+\delta_4+\delta_2\}$, where $\delta_1, \ldots, \delta_6 \in R^+$ are pre-defined.

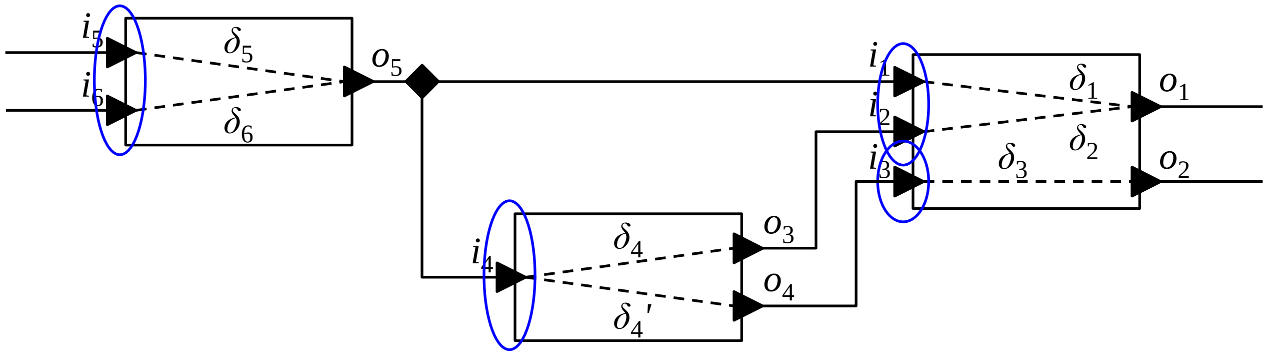
Intuition of Execution Strategy



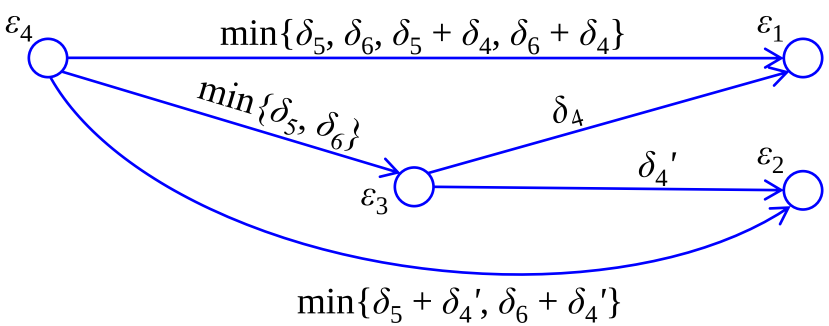
When is it safe to process $e = (v, t)$ at i_1?

1. future events at i_1, i_2 and i_3 have time stamps $\geq t$ (conventional), or
2. future events at i_1 and i_2 have time stamps $\geq t$, or
3. future events at i_1 have time stamps $\geq t$, and
   future events at i_2 depend on events at i_4 with time stamps $\geq t - \delta_4$, or
4. future events at i_1 and i_2 depend on events at i_5 and i_6 with time stamps $\geq t - \min\{\delta_5, \delta_6, \delta_5 + \delta_4, \delta_6 + \delta_4\}$.

Relevant Dependency



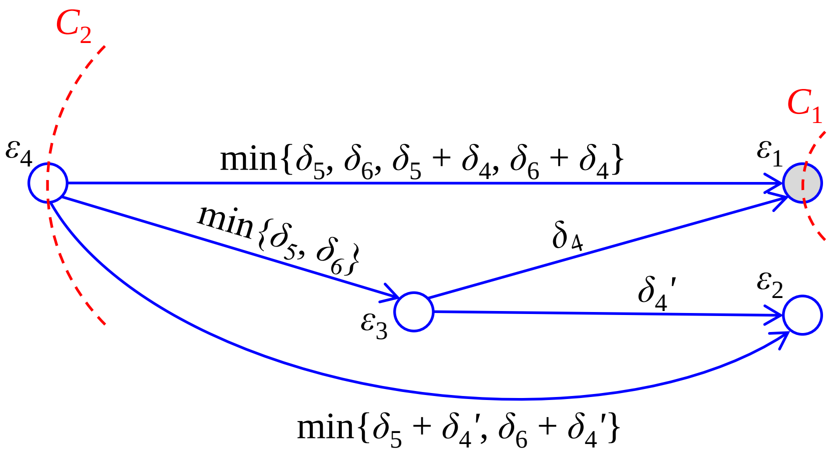
$i \sim i'$ iff they are input of the same actor and affect a common output. An *equivalence class* is a transitive closure of ~.



Construct a collapsed graph, and compute *relevant dependency* between equivalence classes.

$$d(\varepsilon', \varepsilon) = \min_{i' \in \varepsilon',\, i \in \varepsilon} \{\delta(i', i)\}$$

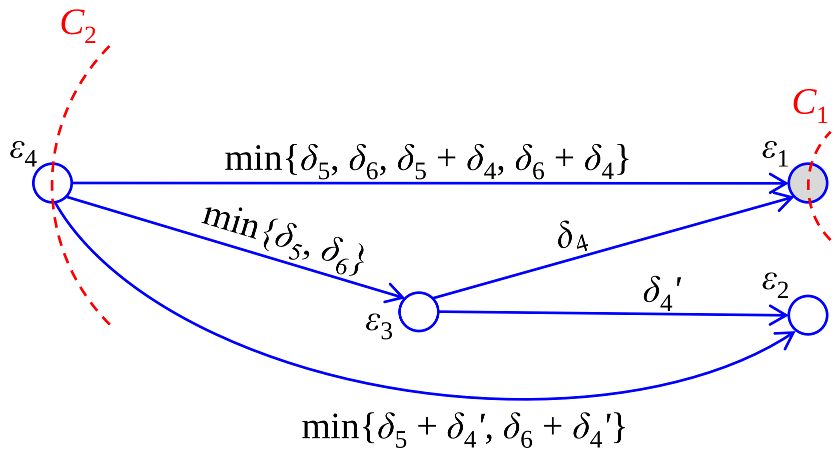
Dependency Cut



A *dependency cut for* ε is a minimal but complete set of equivalence classes that needs to be considered to process an event at ε.

Example: C_1 and C_2 are both dependency cuts for ε_1.

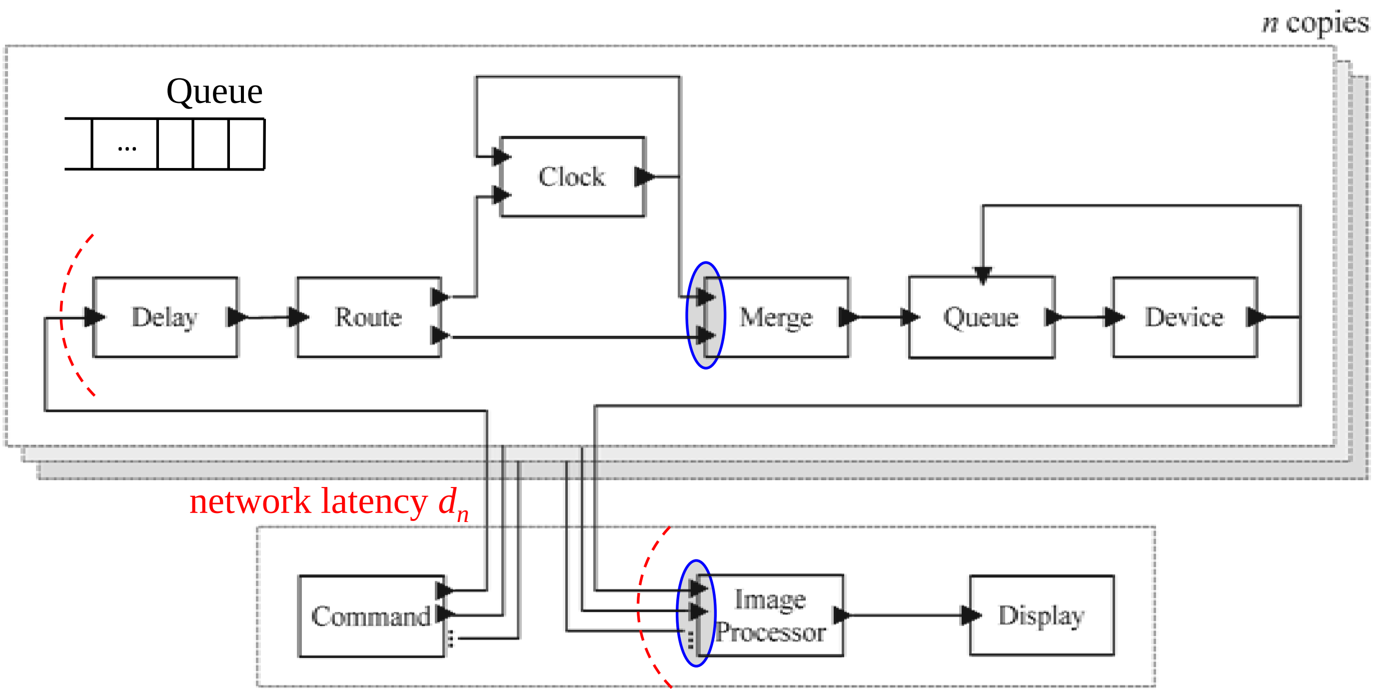
Execution Strategy



Determine top event $e = (v, t)$ at ε_1 safe to process

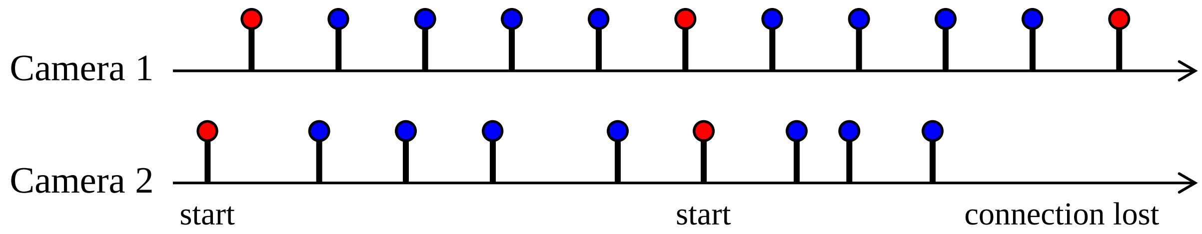
- If we choose C_1: future events at ε_1 have time stamps $\geq t$.
- If we choose C_2: for any $\varepsilon \in C_2$, future events at in ε_1 depend on events at ε with time stamps $\geq t - d(\varepsilon, \varepsilon_1)$.
- In general, we can freely choose any dependency cut.

Implementation of the Execution Strategy



- $n + 1$ platforms with synchronized clocks (IEEE 1588).
- Choose dependency cuts at platform boundary.
- A queue stores events local to the platform.
- At real time τ, future events have time stamps $\geq \tau - d_n$.

Tolerating Loss of Images



- Start the composition as soon as the starting packets are received.

- Create a checkpoint at the beginning (small constant overhead)
- Backtrack when fault is detected (linear in memory locations)
- In most cases, discard the checkpoint (garbage collection)

A Program Transformation Approach

Before Transformation

```
int s;
void f(int i) {
  s = i;
}
```

After Transformation

```
int s;
void f(int i) {
  $ASSIGN$s(i);
}
```

An assignment is transformed into a function call to record the old value:

```
private final int $ASSIGN$s(int newValue) {
  if ($CHECKPOINT != null && $CHECKPOINT.getTimestamp() > 0) {
    $RECORD$s.add(null, s, $CHECKPOINT.getTimestamp());
  }
  return s = newValue;
}
```

This incurs a constant overhead.

A Program Transformation Approach

<table>
<tr><th>Before Transformation</th><th>After Transformation</th></tr>
<tr><td><pre>int s;
void f(int i) {
  s = i;
}</pre></td><td><pre>int s;
void f(int i) {
  $ASSIGN$s(i);
}</pre></td></tr>
<tr><td><pre>Image img;
int partNum;
void consume(Packet p1, Packet p2) {
  if (img == null) {
    img = new Image();
    partNum = 0;
  }
  img.parts[partNum] = compose(p1,
      p2);
  partNum++;
}</pre></td><td><pre>Image img;
int partNum;
void consume(Packet p1, Packet p2) {
  if (img == null) {
    $ASSIGN$img(new Image());
    $ASSIGN$partNum(0);
  }
  img.$ASSIGN$parts(partNum,
      compose(p1, p2));
  $ASSIGN$SPECIAL$partNum(11, -1);
}</pre></td></tr>
</table>

```
0:  +=
1:  -=
    ...
11: ++
12: --
```

Value, not used for ++.

Observation:

The overhead for each basic operation is constant.

Conclusion and Future Work

- Advantages
  - Eliminate null messages
  - Decompose real-time schedulability analysis
  - Advance the system even when some platforms fail
  - Tolerate faults without sacrificing real-time properties
- Future Work
  - Examine different choices of dependency cuts
  - Develop static WCET (worst-case execution time) analysis to guarantee real-time properties on each platform
  - Build an implementation to support a variety of real applications
  - Exploit parallelism with multi-core platforms