Synchronizing Periodic Clocks in Kahn Networks

Albert Cohen\textsuperscript{1}, Marc Duranton\textsuperscript{2}, Christine Eisenbeis\textsuperscript{1}, Claire Pagetti\textsuperscript{1} and Marc Pouzet\textsuperscript{3}
(submitted LCTES)

\textsuperscript{1}Inria Futurs, Orsay France

\textsuperscript{2}Philips Research Laboratories, Eindhoven, The Netherlands

\textsuperscript{3}Université Pierre et Marie Curie, Paris, France

February the 3rd, 2005
Domain: real-time video processing
tera-operations per second (on pixel components)
Conception: specific hardware (ASIC)
Evolution: mixing hardware/software because of
costs, variability of supported algorithms.

Domain-specific designs: general-purpose architectures and
compilers are not suitable. Wish: higher compute
density and programmability \(\Rightarrow\) an appropriate
programming language and compiler.

Synchronous paradigm: generation of custom hardware and
software systems with *correct-by-construction*
*structural properties*, including real-time and resource
constraints.
Multiple Clock Domains

synchronous hypothesis: a common clock for all registers, and an overall predictable hardware where communications and computations can be proven to take less than a clock-cycle.

system-on-chip: is divided into multiple, asynchronous clock domains: *Globally Asynchronous Locally Synchronous* (GALS)

multiple clock domains modular designs with separate compilation phases, for a single system with multiple input/output associated with different real-time clocks;

our assumption: execution layer with a global clock.

Kahn Process Networks (KPN) model for processes communicating through unbounded FIFO buffers.
Downscaler

high definition (HD) $\rightarrow$ standard definition (SD)
$1920 \times 1080$ pixels $\rightarrow$ $720 \times 480$

horizontal filter: number of pixels in a line from 1920 pixels downto 720 pixels,
vertical filter: number of lines from 1080 downto 480
Real-Time Constraints

the input and output processes: 30Hz.

HD pixels arrive at $30 \times 1920 \times 1080 = 62,208,000$Hz

SD pixels at $30 \times 720 \times 480 = 10,368,000$Hz (6 times slower)

HF: 8:3

Reorder: stores 6 lines, transposes them by column of 6 pixels

VF: 9:4
Required Features of the Language

automatically produces an efficient code for an embedded architecture, checking that real-time constraints are satisfied and optimizing the total memory resources to store the intermediate data and the code itself.

1. a proof that, according to worst-case execution time hypotheses, the frame and pixel rate will be sustained;
2. an evaluation of the delay introduced by the downscaler in the video processing chain, i.e., the delay before the output process starts receiving pixels;
3. a proof that the system has bounded memory requirements;
4. an evaluation of memory requirements, to store data within the processes, and to buffer the stream produced by the vertical filter in front of the output process.
Outline

Review
  Introduction
  Example

Clocks as Infinite Binary Words
  Definitions
  Clock Calculus
  Extended Clock Calculus

The Programming Language
  Syntax
  Synchronous Semantics
  Relaxed Synchronous Semantics

I-O Automata Approach
infinite binary words: $(0 + 1)^\omega$

infinite periodic binary words $\Sigma_2^*$:

\[
w ::= u.(v) \\
v ::= 0 | 1 | 0.v | 1.v \\
u ::= \epsilon | 0 | 1 | 0.u | 1.u
\]

with $(v) = \lim_n v^n$ is the periodic repetition of the pattern $v$. 
$|w|$ length of $w$, $|w|_1$ number of 1, $|w|_0$ number of 0, $w[n]$ n-th letter, $w[1..n]$ prefix of length $n$. $w \sqsubseteq w' \Rightarrow \exists v$ binary word, such that $w.v = w'$

$[w]_p$ the position of the $p$-th 1. $w_1 \prec w_2$ iff $\forall p \geq 1$, $[w_1]_p \leq [w_2]_p$
Clocks

clocks

\[ \text{clk} ::= w \mid \text{clk on } w \]
\[ w \in (0 + 1)^\omega \]

on:
\[ 0.w \text{ on } w' = 0. (w \text{ on } w') \]
\[ 1.w \text{ on } 0.w' = 0. (w \text{ on } w') \]
\[ 1.w \text{ on } 1.w' = 1. (w \text{ on } w') \]

algorithm
\[ w'' = w \text{ on } w' \text{ with } \forall n \in \mathbb{N}, w''[n] = w[n] \wedge w'[|w[1..n]|_1]. \]

on-associativity
Let \( w_1, w_2 \) and \( w_3 \) be three infinite binary words. Then
\[ w_1 \text{ on } (w_2 \text{ on } w_3) = (w_1 \text{ on } w_2) \text{ on } w_3. \]
Periodic Clocks:

\[ \text{clk} ::= w \mid \text{clk on } w \]
\[ w \in \Sigma_2^* \]

We can always write \( \text{clk}_1 = a.(b) \) and \( \text{clk}_2 = c.(d) \) with
\[ |a| = |c| = \max(|u|, |u'|) \]
\[ |b| = |d| = \text{lcm}(|v|, |v'|) \]
where \( \text{lcm} \) is the least common divisor.
For instance, \( 010(001100) \) and \( 10001(10) \) become \( 01000(110000) \) and \( 10001(101010) \)
Clock Signatures

A synchronous process transforms an input clock into an output clock. This transformation is encoded in the process clock signature

\[ \alpha \rightarrow \alpha \text{ on } w \]

It means that for all valuation \( w' \in \Sigma_2^* \) of the variable \( \alpha \), the output has the clock \( w' \text{ on } w \).
Downscaler Signatures

1. input process is the binary word (1)
2. HF: $\alpha \rightarrow \alpha$ on (10100100).
3. reordering process delays the output of $5 \times 720 \times 8/3 = 7680$ cycles. The clock signature $\alpha \rightarrow 0^{7680}\alpha$.
4. VF:
   $\alpha \rightarrow \alpha$ on $(1^{720}0^{720}1^{720}0^{1440}1^{720}0^{1440}1^{720})$
   
   simplification: $\alpha \rightarrow \alpha$ on (101001001).
5. output’s process clock (1000000).
Synchronizing Clocks

\[ f_1 : \alpha_1 \rightarrow C_1[\alpha_1] \text{ and } f_2 : \alpha_2 \rightarrow C_2[\alpha_2] \]

then composition \( f = f_2(f_1) : \alpha_1 \rightarrow C_2[C_1[\alpha_1]] \)

Required: \( \text{output} = \text{buffer(vert(reorder(hor(input))))} \)

- \( \text{hor(input)}: (1) \text{ on } (10100100) = (10100100). \)
- \( \text{reorder(hor(input))}: 0^{7680}(10100100). \)
- \( \text{vert(reorder(hor(input))}): 0^{7680}(10100100) \text{ on } (101001001) = 0^{7680}(1000010000000100000000100) \neq (100000). \)
Synchronizability

$clk_i$ are synchronizable, $clk_1 \lhd clk_2$, iff there exists $d, d' \in \mathbb{N}$ such that $clk_1 \prec 0^d . clk_2$ and $clk_2 \prec 0^{d'} . clk_1$.

It means that we can delay $clk_1$ by $d'$ ticks so that the 1 of $clk_2$ occur before the 1 of $clk_1$ and conversely.

1. 11(01) and (10) are synchronizable;
2. 11(0) and (0) are not synchronizable;
3. (010) and (10) are not synchronizable since there are too much reads or too much writes (infinite buffer).

$clk_1 \lhd clk_2 \Rightarrow \exists$ two synchronous processes, called buffers $b_1$ and $b_2$ such that $b_1(clk_1) = 0^d . clk_2$ and $b_2(clk_2) = 0^{d'} . clk_1$. 

Verimag Grenoble - February - 2005  ACI Alidecs
Synchronizability: Periodic Clocks

\(clk_1 \bowtie clk_2\) iff

\[
\begin{cases}
\frac{|v|_1}{|v'|_1} = \frac{|v|}{|v'|} & \text{if } |v'|_1 > 0 \\
|u| = |u|_1 \land |v|_1 = 0 & \text{otherwise}
\end{cases}
\]

The first condition means that are in average the same number of writes and reads in \((v)\) and \((v')\). The second condition deals with the particular case of finite streams where there must be precisely the same number of writes and reads.
Delaying a Clock

Delay the reads after the writes

\[ delay = \min \{ l \mid clk_1 \prec klk_2 \} \]

\( clk_1 = u(v) \) and \( clk_2 = u'(v') \) with \( |u| = |u'| \) and \( |v| = |v'| \). Then

\[ delay = \max (d', 0) \]

where

\[ d' = \max \{|w| - |w'| \mid w.1 \sqsubseteq uv, w'.1 \sqsubseteq u'v', |w'|_1 = |w|_1\} \]

For instance if \( clk_1 = 000001 \) and \( clk_2 = 001000 \), then \( d = 3 \) is reached with \( w = 000001 \) and \( w' = 00 \).

Downscaler:

\[
\begin{array}{cccccc}
100001 & 000000 & 010000 & 000100 \\
000100 & 000100 & 000100 & 000100 \\
000100 & 000100 & 000100 & 000100 \\
\end{array}
\]
Buffer Size

$clk_1 < 0^d . clk_2$. We write $clk_1 = u(v)$ and $0^d . clk_2 = u'(v')$ with $|u| = |u'|$ and $|v| = |v'|$.

The minimal buffer size $n$ satisfies:

$$n = \max \{ |w_1| - |w'|_1 | w \sqsubseteq uv, w' \sqsubseteq u'v'; |w| = |w'| \}$$

Communication from $clk_1$ to $clk_2$ is called $n$-synchronous.

**Downscaler:** simplified version buffer size $= 1$ and general version $= 400$. 
Buffer Construction

**NOP** — $w[j] = 0$ and $w'[j] = 0$: Nothing happens in the buffer: $clk_i[j] = 0$, $w_i[j] = w_i[j - 1]$; registers $x_i$ are left unchanged.

**PUSH** — $w[j] = 1$ and $w'[j] = 0$: Some data is written into the buffer and stored in register $x_1$, all the data in the buffer being pushed from $x_i$ into $x_{i+1}$. Thus $x_i = x_{i-1}$ and $x_1 = \text{input}$, $\forall i > 2$, $w_i[j] = w_{i-1}[j - 1]$, $w_1[j] = 1$ and $clk_i[j] = 0$. 
Buffer Construction

**POP —** $w[j] = 0$ and $w'[j] = 1$: Let $p = \max \{0\} \cup \{1 \leq i \leq n | w_i[j - 1] = 1\}$. If $p$ is zero, then no register stores any data at cycle $j$: input data must be bypassed directly to the output, crossing the wire clocked by $clk_0$, setting $clk_i[j] = 0$ for $i > 0$ and $clk_0[j] = 1$, $w_i[j] = w_i[j - 1]$. Conversely, if $p > 0$, $\forall i \neq p$, $clk_i[j] = 0$, $clk_p[j] = 1$, $\forall i \neq p$, $w_i[j] = w_i[j - 1]$ and $w_p[j] = 0$. Registers $x_i$ are left unchanged (notice this is not symmetric to the PUSH operation).

**POP; PUSH —** $w[j] = 1$ and $w'[j] = 1$: A POP is performed, followed by a PUSH, as defined in the two previous cases.
A Synchronous Data-flow Kernel

\[
e ::= x \mid i \mid (e, e) \mid e \text{ where } x = e \mid e(e) \\
     \mid e \text{ fby } e \mid e \text{ when } pe \mid \text{merge } pe e e \\
     \mid \text{fst } e \mid \text{snd } e
\]

Expressions \((e)\), constants \((i)\), variables \((x)\), pairs \((e, e)\), local definitions \((e \text{ where } x = e)\), applications \((e(e))\), initialized delays \((e \text{ fby } e)\).
A Synchronous Data-flow Kernel

\[ e ::= x \mid i \mid (e, e) \mid e \text{ where } x = e \mid e(e) \]
\[ \quad \mid e \text{ fby } e \mid e \text{ when } pe \mid \text{merge } pe e e \]
\[ \quad \mid \text{fst } e \mid \text{snd } e \]

\[ d ::= \text{node } x(x) = e \mid d ; d \]

stream functions: \text{node } x(x) = e
A Synchronous Data-flow Kernel

\[
e ::= x \mid i \mid (e, e) \mid e \ where \ x = e \mid e(e) \\
| \ e \ fby \ e \mid e \ when \ pe \mid \merge \ pe \ e \ e \\
| \ \fst \ e \mid \snd \ e
\]

\[
d ::= \node \ x(x) = e \mid d; d
\]

\[
dp ::= \period \ p = pe \mid dp; dp
\]

periods: \ period \ p = pe
A Synchronous Data-flow Kernel

\[ e ::= x | i | (e, e) | e \text{ where } x = e | e(e) \\
| e \text{ fby } e | e \text{ when } pe | \text{merge } pe \ e \ e \\
| \text{fst } e | \text{snd } e \]

\[ d ::= \text{node } x(x) = e | d; d \]

\[ dp ::= \text{period } p = pe | dp; dp \]

\[ pe ::= p | w | pe \text{ on } pe | \text{not } pe | pe \text{ or } pe | pe \text{ & } pe \]
Example

\begin{verbatim}
node hf p = o where
  o1= p
  and o2= 0 fby o1
  and o3= 0 fby o2 and o4= 0 fby o3
  and o5= 0 fby o4 and o6= 0 fby o5
  and o= (o1 + o2 + o3 + o4 + o5 + o6)/6 when (10100100)

node vf (i1,i2,i3,i4,i5,i6) = o where
  o= (i1 + i2 + i3 + i4 + i5 + i6)/6 when (101001001)

node main (i : (1)) = (o : 0^7683(100000)) where
  t = fh i and (i1,i2,i3,i4,i5,i6) = buff1(t)
  and o = vf (i1,i2,i3,i4,i5,i6);;
\end{verbatim}
Clock Calculus I

Clock calculus as a type system → judgments of the form $P, H \vdash e : ct$ meaning that “the expression $e$ has clock type $ct$ in the environment of periods $P$ and the environment $H$”.

$$\sigma ::= \forall \alpha_1, \ldots, \alpha_m.ct$$
$$ct ::= ct \to ct \mid ct \times ct \mid ck$$
$$ck ::= \text{base} \mid \text{ck on } pe \mid \alpha$$

$$H ::= [x_1 : \sigma_1, \ldots, x_m : \sigma_m]$$
$$P ::= [p_1 : pe_1, \ldots, p_n : pe_n]$$

clock schemes ($\sigma$), unquantified clock types ($ct$), clock type variables ($\alpha$), functional clock types ($ct \to ct$), products ($ct \times ct$), or stream clocks ($ck$), base clock ($\text{base}$), sampled clock ($ck \text{ on } pe$), clock variable ($\alpha$).
0-Synchrony Compilation

node t i =

... clock calculus
rules (OP) 

Compilation

prog (caml ..)

accepted

rejected
Relaxed Clock Calculus

\(n\)-synchronous programs \(\leftrightarrow\) programs which can be executed using buffers of size at most \(n\).

Kahn networks \(\leftrightarrow\) \(\infty\)-synchronous programs.

Synchronous programs \(\leftrightarrow\) 0-synchronous programs.

Extension of the previous clock calculus with a sub-typing rule:

\[
\begin{array}{c}
\frac{P, H \vdash_s e : \text{ck on } p_{e_1} \quad p_{e_1} \prec p_{e_2}}{P, H \vdash_s e : \text{ck on } p_{e_2}} \\
\text{(SUB)}
\end{array}
\]
Example

node \( f(x) = y \) where \( y = (x \text{ when } (01)) + (x \text{ when } 1(10)) \)

(01) and 1(10) can be synchronized using a buffer of size 1. We can apply the rule:

\[
P, H \vdash_s x \text{ when } 1(10) : \alpha \text{ on } 1(10) \quad 1(10) \prec (01)
\]

and then, classical rules apply and we get the final signature:

\[
f : \forall \alpha. \alpha \rightarrow \alpha \text{ on } (01)
\]
Translation Semantics

programs accepted with the relaxed clock calculus $\rightarrow$ synchronous programs which are accepted by the original clock calculus through a program transformation which insert a buffer every time the (SUB) is applied.
Relaxed Synchrony Compilation

node \( t \) \( i = \)

- extended clock calculus
  - rules (OP) ....
  - \text{SUB}

- accepted

transformation

- \( \text{ck on } p \leftrightarrow \text{ck on } p' \)
- insert
- \( \text{buff}(p/p') \)

Compilation

prog (caml ..)
Buffer in Lucid size 1

```
let buffer1 (push, pop, i) = (empty, o) where
 rec o = if pempty then i
     else pmemo
 and memo = if push then i else pmemo
 and pmemo = 0 fby memo
 and empty = if push then if pop then pempty
             else false
             else if pop then true
             else pempty
 and pempty = true fby empty
```
Alphabet

\[ \Sigma = \{i/o, i/\bar{o}, \bar{i}/o, \bar{i}, /\bar{o}\}. \]

- the symbol \( i \) stands for an input occurs,
- the symbol \( o \) stands for an output occurs, so that for instance the event \( i/o \) means that an input and an output occur simultaneously,
- \( \bar{i} \) stands for no input occurs and the symbol \( \bar{o} \) stands for no output occurs.
Example

$A_0 = \text{input}$ \hspace{1cm} $A_1 = \text{horizontal filter}$

$L \subseteq \Sigma^*$ can be seen as $L_1 \times L_2 \subseteq \{0, 1\}^* \times \{0, 1\}^*$ with $l \leftrightarrow 1$ and $\bar{l} \leftrightarrow 0$ with $l = i, o$
Synchronization

$L(A_1) \equiv L_1^1 \times L_2^1$ and $L(A_2) \equiv L_1^2 \times L_2^2$. Then $A = A_1 \parallel A_2$ recognizes $L(A) \equiv L_1 \times L_2$. We have $L_1 \rightarrow L_1$ on $L_2 = L_1^1 \rightarrow (L_1^1$ on $L_2^1$ on $(L_1^2$ on $L_2^2))$. This means that $L_1 = L_1^1$ and $L_2 = L_2^1$ on $(L_1^2$ on $L_2^2)$. 
Conclusion

- design of real-time applications: strong correctness requirements, decomposition into modular components communicating thanks to a buffering mechanism;
- global system is synchronous but hard by hand;
- extended synchronous framework: automatic generation of the synchronous buffers which are semantically (as defined by Kahn) guaranteed correct.
  - periodic clocks;
  - synchronous functional programming language.
Future Work

- algebraic characterization of clocks (diadic numbers);
- connection between retiming and delay insertion;
- towards a criterion of optimization of the buffers (here we choose a particular solution but no unicity);
- delays in the language.