dataflow
parallel computing with streams

Jörn W. Janneck

Dept. of Computer Science
Lund Institute of Technology (LTH)
overview

motivation
the need for a parallel programming model

dataflow programming
actors, dataflow, and the CAL actor language

dataflow perspectives
determinacy – the structure of computation

research
the once and future research agenda
motivation
the need for a parallel programming model

dataflow programming
actors, dataflow, and the CAL actor language

dataflow perspectives
determinacy – the structure of computation

research
the once and future research agenda
Moore's law

(Borrowed from Seth Goldstein.)
processor speed

Now, we need to go parallel.
programming parallel machines

C w/ “iLib”

SOPM (“ajava”)

C + VHDL

CUDA

GT200

TILE 64

Am2045

Virtex

HDL

Virtex

PC102

PC102

IntellaSys

CUDA

CUDA

CUDA

CUDA
programming sequential machines
programming sequential machines

\[\text{C, Pascal, Forth, Java, Lisp, Fortran, Haskell, ...}\]

“von Neumann” machine

- Pentium IV
- PDP-11
- Fujitsu Super SPARC
- 68020
a pivotal metaphor
programming parallel machines

- Pentium IV
- Penryn quadcore
- Am2045
- Nvidia GT200
- Virtex

- Processor 1
- Multi-core 2 ~ 10s
- MPPA, GPU 10s ~ 100s ...
- FPGA 100s ~ 1000s
portable concurrency

application
(explicit parallelism)

platform
(physical parallelism)
portable concurrency

application
(explicit parallelism)

platform
(physical parallelism)

SW w/ threads
parallelizing compilers
HDL
portable concurrency

application (explicit parallelism)

Sequentialization

Parallelization

platform (physical parallelism)
motivation
the need for a parallel programming model

**dataflow programming**
actors, dataflow, and the CAL actor language

**dataflow perspectives**
determinacy – the structure of computation

research
the once and future research agenda
parallel programming with streams

- computational kernels (aka actors)
  - provide explicit concurrency

- directed point-to-point connections
  - lossless, order-preserving
  - conceptually unbounded
  - asynchrony, abstraction from time

- communicating streams of discrete data packets (tokens)
  - no shared state
not exactly a new model...

domain expert's lunchtime rendition of a video decoder

digital signal processing
media processing
video coding
image processing / analytics
audio
networking / packet processing
...

16
data flow schemas (Dennis et al.)

INTRODUCTION

A data flow schema is a representation of the logical scheme of a program in a form in which the sequencing of function and predicate applications and the flow of values between applications are specified together. In a data flow schema, an application of a function or predicate is free to proceed as soon as the values required for its application are available and the need for application of the function or predicate is determined. Since the availability of one computed value may simultaneously enable the application of several functions or predicates, concurrency of action is an inherent aspect of a data flow schema.

We present here some basic properties of a class of data flow schemas which

networks of (potentially concurrent) functions

triggered by the availability of input data

executing in a sequence of atomic transitions

described by firing rules
**Kahn process networks** (Kahn)

In this paper, we describe a simple language for parallel programming. Its semantics is studied thoroughly. The desirable properties of this language and its deficiencies are exhibited by this theoretical study. Basic results on parallel program schemas are given. We hope in this way to make a case for a more formal (i.e., mathematical) approach to the design of languages for systems programming and the design of operating systems.

i) To every line \( e \), of type \( D_e \), associate a variable \( X \) ranging over \( D_e \).

ii) If \( X_1, X_2, \ldots, X_n \) are the variables associated to the input lines and \( i_1, \ldots, i_k \) are the sequences fed as inputs on the lines include the equations:

\[
\begin{align*}
X_1 &= i_1 \\
& \vdots \\
X_k &= i_k
\end{align*}
\]

iii) For each node \( f \), interpreted with the functions \( f_1, \ldots, f_p \), with input variables \( X_1, \ldots, X_n \)

output variables \( X'_1, \ldots, X'_p \) include \( p \) equations

\[
\begin{align*}
X'_1 &= f_1(X_1, \ldots, X_n) \\
& \vdots \\
X'_p &= f_p(X_1, \ldots, X_n)
\end{align*}
\]

networks of *stream functions* … described as communicating tasks with blocking reads

precise criterion for determinacy (*prefix-monotonicity*)

formal *fixpoint semantics*
stateful, active objects (actors)

communicating via message passing

non-determinism and partial orders

functional actors with fixpoint semantics
actors as stream operators

actors operate on infinite objects

they need not terminate
dataflow with firing

actor transition (firing) may
- consume input tokens
- produce output tokens
- modify the state

(input; old state) : $s_0 \rightarrow$ (output; new state) : $s_1$
examples

**example transitions**

- \((1; 2) : \circ \rightarrow 3 : \circ\)
- \((4; -9) : \circ \rightarrow -5 : \circ\)

**example runs**

\[
\begin{array}{ccccccc}
3 & 4 & -1 & 1 & 2 & \ldots \\
7 & -9 & 7 & 2 & 0 & \ldots \\
10 & -5 & 6 & 3 & 2 & \ldots \\
\end{array}
\]

\[
\begin{array}{ccccccc}
7 & -9 & 7 & 2 & 0 & \ldots \\
7 & -2 & 5 & 7 & 7 & \ldots \\
\end{array}
\]

**NDMerge**

- \((1; \epsilon) : \circ \rightarrow 1 : \circ\)
- \((\epsilon; c) : \circ \rightarrow c : \circ\)

**sum**

- \(2 : 5 \rightarrow 7 : 7\)
- \(-9 : 7 \rightarrow -2 : -2\)

**example runs**

\[
\begin{array}{ccccccc}
1 & 2 & 3 & \ldots \\
a & b & c & \ldots \\
\end{array}
\]

or

\[
\begin{array}{ccccccc}
1 & 2 & 3 & a & b & c & \ldots \\
a & b & c & 1 & 2 & 3 & \ldots \\
1 & a & 2 & b & 3 & c & \ldots \\
\ldots \\
\end{array}
\]
writing actors in CAL: actions

large number of transitions – cannot list them explicitly

transitions are of the form

\[ a : \text{sum} \rightarrow a+\text{sum} : a+\text{sum} \]

input \quad old \quad output \quad new
state \quad state

An **action** is a description of a **family of transitions**:

1. introduces variables for the input tokens
2. describes what happens to the state
3. specifies the values of output tokens

Action \( A: [a] ==> X: [\text{sum}] \)

\begin{verbatim}
  do
    sum := sum + a;
  end
\end{verbatim}
writing actors in CAL

**example transitions**

- **Add**
  - $(1; 2) : \circ \rightarrow 3 : \circ$
  - $(4; -9) : \circ \rightarrow -5 : \circ$

- **Sum**
  - $2 : 5 \rightarrow 7 : 7$
  - $-9 : 7 \rightarrow -2 : -2$

- **NDMerge**
  - $(1; \varepsilon) : \circ \rightarrow 1 : \circ$
  - $(\varepsilon; 2) : \circ \rightarrow 2 : \circ$

**code**

```plaintext
actor Add () A, B ==> X:
  action A: [a], B: [b] ==> X: [a + b] end
end

actor Sum () A ==> X:
  sum := 0;
  action A: [a] ==> X: [sum]
  do
    sum := sum + a;
  end
end

actor NDMerge () A, B ==> X:
  action A: [v] ==> X: [v] end
  action B: [v] ==> X: [v] end
end
```
What happens if both transitions can occur?  

If we care about the answer, we have to provide it.  
We do so by ordering transitions.

An actor is a partially ordered set of transitions.  
An actor description contains a partially ordered set of actions.
historical note  (CAL and Dennis data flow)

actor F () A, B ==> X:

  action A: [a], B: [b] ==> X: [f(a,b)]  end
end

actor Switch () S, T, F ==> X:

  action S:[s], T: [v] ==> X: [v]
guard s end

  action S:[s], F: [v] ==> X: [v]
guard not s end
end
actors, big and small

Compare
23 lines
(without header comments)

ParseHeaders
1320 lines
(without header comments)

http://opendf.svn.sourceforge.net/viewvc/opendf/models/MPEG4_SP_Decoder/
actors and actions

**action** – describes a family of potential actor transitions  
**actor run / execution** – a sequence of action executions  
**action selection** – the process of picking the next action to execute
actor execution model
Action selection is the focus of much of how actors are described and processed by tools.

1. Compute the **enabling conditions** for all actions.

2. An action is **enabled** if all its enabling conditions are met.

3. Pick one of the enabled actions, if any.
enabled actions

1. its input patterns are satisfied

```
actor Add () A, B ==> X:
    action A: [a], B: [b] ==> X: [a + b] end
end
```

2. its guards are true

```
actor Split () A ==> P, N:
    action A: [v] ==> P: [v]
    guard v >= 0
    end

    action A: [v] ==> N: [v]
    guard v < 0
    end
end
```

3. no higher-priority action is enabled

```
actor BiasedMerge () A, B ==> X:
    CopyA: action A: [v] ==> X: [v] end
    CopyB: action B: [v] ==> X: [v] end
    priority CopyA > CopyB; end
end
```
motivation
the need for a parallel programming model

dataflow programming
actors, dataflow, and the CAL actor language

dataflow perspectives
determinacy – the structure of computation

research
the once and future research agenda
the classic case

In the domain of threads, determinacy is *not compositional*.

What would be a non-trivial condition for a composition of threads to be determinate?
actor determinism

If there is always at most one enabled action, the actor is determinate. Otherwise, it may not be.

Some simple (sufficient) conditions for determinism:

There is only one action.

Actions are totally priority-ordered.

```plaintext
actor Add () A, B ==> X:
    action A: [a], B: [b] ==> X: [a + b] end
end

actor BiasedMerge () A, B ==> X:
    CopyA: action A: [v] ==> X: [v] end
    CopyB: action B: [v] ==> X: [v] end
    priority CopyA > CopyB; end
end
```
**dataflow program determinism**

In dataflow, too, determinacy is not compositional.

Can we find a property of actors that is compositional, and that entails determinacy?
We can predict (determinate) actor behavior for each input sequence.

Unfortunately, *we cannot predict the input sequence* at each point in time.

Can we characterize actors that produce the same output irrespective of the *timing of token arrival*?
An actor is timing-independent\(^1\) if *additional input tokens never disable an action*. 

\(^1\) aka “prefix-monotonic”
How can additional input *disable* an action?

It would have to “unsatisfy” a previously satisfied enabling condition.

- action A: [a], B: [b] ==> ...
  
  - 1. all input patterns are satisfied
  - 2. all guards are true
  - 3. no higher-priority action is enabled

CopyA: action A: [v] ==> X: [v] end
CopyB: action B: [v] ==> X: [v] end

priority CopyA > CopyB; end

The additional input tokens would have to *enable a higher-priority action*. 
Timing independence

How would additional tokens enable a higher-priority action?

The higher-priority action would have to require at least one token not needed by the lower-priority action.

```plaintext
actor BiasedMerge () A, B ==> X:

CopyA: action A: [v] ==> X: [v] end
CopyB: action B: [v] ==> X: [v] end

priority CopyA > CopyB; end
end
```
Looking at the input patterns and the priority order, we can derive guarantees about timing-independence of each actor.

At compile time.

Any composition of timing-independent actors is determinate.
It is also timing-independent itself.
useful timing dependence

Why not just disallow timing-dependent behavior?

```plaintext
actor Amp () In, Vol ==> Out:

  vol := 1;

  action Vol: [newV] ==> do
    vol := newV;
  end

  action In: [d] ==> Out: [d * v] end
end

actor BrokenAmp () In, Vol ==> Out:

  vol := 1;

  action Vol: [newV] ==> do
    vol := newV;
  end

  action In: [d] ==> Out: [d * v] end
end

priority V > D; end
```
a taxonomy of actors

timing-independent
(prefix monotonic)

actor Add () A, B ==> X:

  action A: [a], B: [b] ==> X: [a + b] end
end

actor Merge () A, B ==> X:

  action A: [v] ==> [v] end
  action B: [v] ==> [v] end
end

actor OddActor () A ==> X:

  High:
  action [a1, a2] ==> [a1+a2] end
  action B: [v] ==> [v] end

  Low:
  action [a] ==> [a] end
  action [a] ==> [-a] end

  priority High > Low; end
end

actor Split () Input ==> Output1, Output2:

  action [x] ==> Output1: [x] end
  action [x] ==> Output2: [x] end
end

timing-dependent

actor BiasedMerge () A, B ==> X:

  CopyA: action A: [v] ==> [v] end
  CopyB: action B: [v] ==> [v] end

  priority CopyA > CopyB; end
end
motivation
the need for a parallel programming model

dataflow programming
actors, dataflow, and the CAL actor language

dataflow perspectives
determinacy – the structure of computation

research
the once and future research agenda
structure of a computation

actor A () ==> X:
    s := 0;
    action ==> X: [s]
do
    s := s + 1;
end
end

actor B () A ==> X:
    action A: [a] ==> X: [f(a)] end
end

actor C () A ==>:
    action A: [a] ==> end
end

trace is a DAG
vertices are action firings
edges are dependencies

dependencies mediated by
tokens
state variables
(ports)
traces of a decoder

parsing

parsing, end of header

IDCT
trace analysis

...0111010010010

input/output tagging
trace metrics (examples)

**input**
(bytes)

**cost**
(Σ area * time)

**latency**
(incremental)

**parallelizability**
(cost/latency)
motivation
the need for a parallel programming model

dataflow programming
actors, dataflow, and the CAL actor language

dataflow perspectives
determinacy – the structure of computation

research
the once and future research agenda
Cal Actor Language history

2002
- CAL
- CAL interpreter (Ptolemy)

2003
- CLR 1.0

2004
- MPEG-4 decoder...

2005
- ... in hardware
- OpenDF (SourceForge)

2006
- ... in software
- Eclipse plugin

2007
- Graphiti

2008
- ORCC

2009
- Open HDL codegenerator

2010
- RVC-CAL

SystemBuilder

MPEG RVC

ACTORS project

Open Dataflow

first hardware
1. Expressiveness: Can we build interesting things, easier/better/...?

   - MPEG video library
   - TCP/IP stack
   - image processing
   - LTE inner receiver
   - channel estimator

2. Implementability: Can we generate efficient implementations?

   - outperformed hand designs on FPGA in size, throughput, productivity
   - outperformed hand design for ASIC in power, throughput, productivity
   - significant improvements on multicores, but still work to be done

3. Adoption: Can/will people use it?

   - MPEG/ISO standardization, consistent use in MPEG-RVC community
   - some experimental use in industry and academia
   - small community of steady contributors
     EPFL, LTH Control/CS/EIT, INSA Rennes, Ericsson, ...
open problems

• implementation tools
  – efficient code generation for processors and multicores
  – implementation on other targets: GPUs, processor arrays, heterogeneous targets
  – runtime, virtual machine

• design tools
  – partitioning, profiling, early performance prediction
  – trace analysis
  – refactoring and code transformation (e.g. unrolling, folding, multi-channelization)
  – debugging, deadlock/bottleneck detection and tracking
open problems, cont'd

- **foundations**
  - program analysis, partial static scheduling, theory, machine model

- **applications**
  - streaming applications (graphics, packet processing, ...)
  - other application areas (HPC, processor design, ...)

- **architecture**
  - building dataflow hardware
credits & resources


opendf.org
Open Dataflow portal

opendf.sf.net
open source tool suite

orcc.sf.net
Open RVC-CAL Compiler

www.actors-project.eu
ACTORS Project

mpeg.chiariglione.org
MPEG

jwj@cs.lth.se
BACKUP
<table>
<thead>
<tr>
<th>Name</th>
<th>TPCI</th>
<th>TPCI cum.</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>17.66%</td>
<td>17.66%</td>
<td>1973</td>
</tr>
<tr>
<td>C++</td>
<td>11.06%</td>
<td>28.73%</td>
<td>1985</td>
</tr>
<tr>
<td>Perl</td>
<td>5.48%</td>
<td>34.20%</td>
<td>1987</td>
</tr>
<tr>
<td>Python</td>
<td>3.47%</td>
<td>37.67%</td>
<td>1990</td>
</tr>
<tr>
<td>VB</td>
<td>9.73%</td>
<td>47.40%</td>
<td>1991</td>
</tr>
<tr>
<td>Delphi</td>
<td>2.15%</td>
<td>49.54%</td>
<td>1994</td>
</tr>
<tr>
<td>Java</td>
<td>21.17%</td>
<td>70.72%</td>
<td>1995</td>
</tr>
<tr>
<td>PHP</td>
<td>9.86%</td>
<td>80.58%</td>
<td>1995</td>
</tr>
<tr>
<td>JavaScript</td>
<td>2.20%</td>
<td>82.78%</td>
<td>1995</td>
</tr>
<tr>
<td>C#</td>
<td>3.07%</td>
<td>85.85%</td>
<td>2002</td>
</tr>
</tbody>
</table>

Cumulative TCPI by language creation date (for top 10 languages)

building (& specifying) a video decoder

```c
int main(int argc, char **argv)
{
    int iRet;
    DecodedPicList *pDecPicList;
    int hFileDecOutput0=-1, hFileDecOutput1=-1;
    int iFramesOutput=0, iFramesDecoded=0;
    InputParameters InputParams;
#if DECOUTPUT_TEST
    hFileDecOutput0 = open(DECOUTPUT_VIEW0_FILENAME, OPENFLAGS_WRITE, OPEN_PERMISSIONS);
    fprintf(stdout, "Decoder output view0: %s\n", DECOUTPUT_VIEW0_FILENAME);
    hFileDecOutput1 = open(DECOUTPUT_VIEW1_FILENAME, OPENFLAGS_WRITE, OPEN_PERMISSIONS);
    fprintf(stdout, "Decoder output view1: %s\n", DECOUTPUT_VIEW1_FILENAME);
#endif
    //get input parameters;
    Configure(&InputParams, argc, argv);
    //open decoder;
    iRet = OpenDecoder(&InputParams);
    if(iRet != DEC_OPEN_NOERR)
    {
        fprintf(stderr, "Open encoder failed: 0x%x!\n", iRet);
        return -1; //failed;
    }
    //decoding;
    do
    {
        iRet = DecodeOneFrame(&pDecPicList);
        if(iRet==DEC_EOS || iRet==DEC_SUCCEED)
        {
            //process the decoded picture, output or display;
            iFramesOutput += WriteOneFrame(pDecPicList, hFileDecOutput0, hFileDecOutput1, 0);
            iFramesDecoded++;
        }
        else
        {
            //error handling;
            fprintf(stderr, "Error in decoding process: 0x%x\n", iRet);
        }
    }
    while((iRet == DEC_SUCCEED) && ((p_Dec->p_Inp->iDecFrmNum==0) || (iFramesDecoded<p_Dec->p_Inp->iDecFrmNum)));
    iRet = FinitDecoder(&pDecPicList);
    iFramesOutput += WriteOneFrame(pDecPicList, hFileDecOutput0, hFileDecOutput1, 1);
    iRet = CloseDecoder();
    //quit;
    if(hFileDecOutput0>=0)
    {
        close(hFileDecOutput0);
    }
    if(hFileDecOutput1>=0)
    {
        close(hFileDecOutput1);
    }
    //printf("%d frames are decoded.\n", iFramesDecoded);
    //printf("%d frames are decoded, %d frames output.\n", iFramesDecoded, iFramesOutput);
    return 0;
}
```

The input is an encoded bitstream as a series of octets. The decoder index output is in
the usual format specified. There are also connections to an external HDMI and multi-port
controller (see diagram): data and address - MIO, MCA; The HDMI output carries information about each macroblock
line, as well as the start of frame and frame sync to the video display.

---

56