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

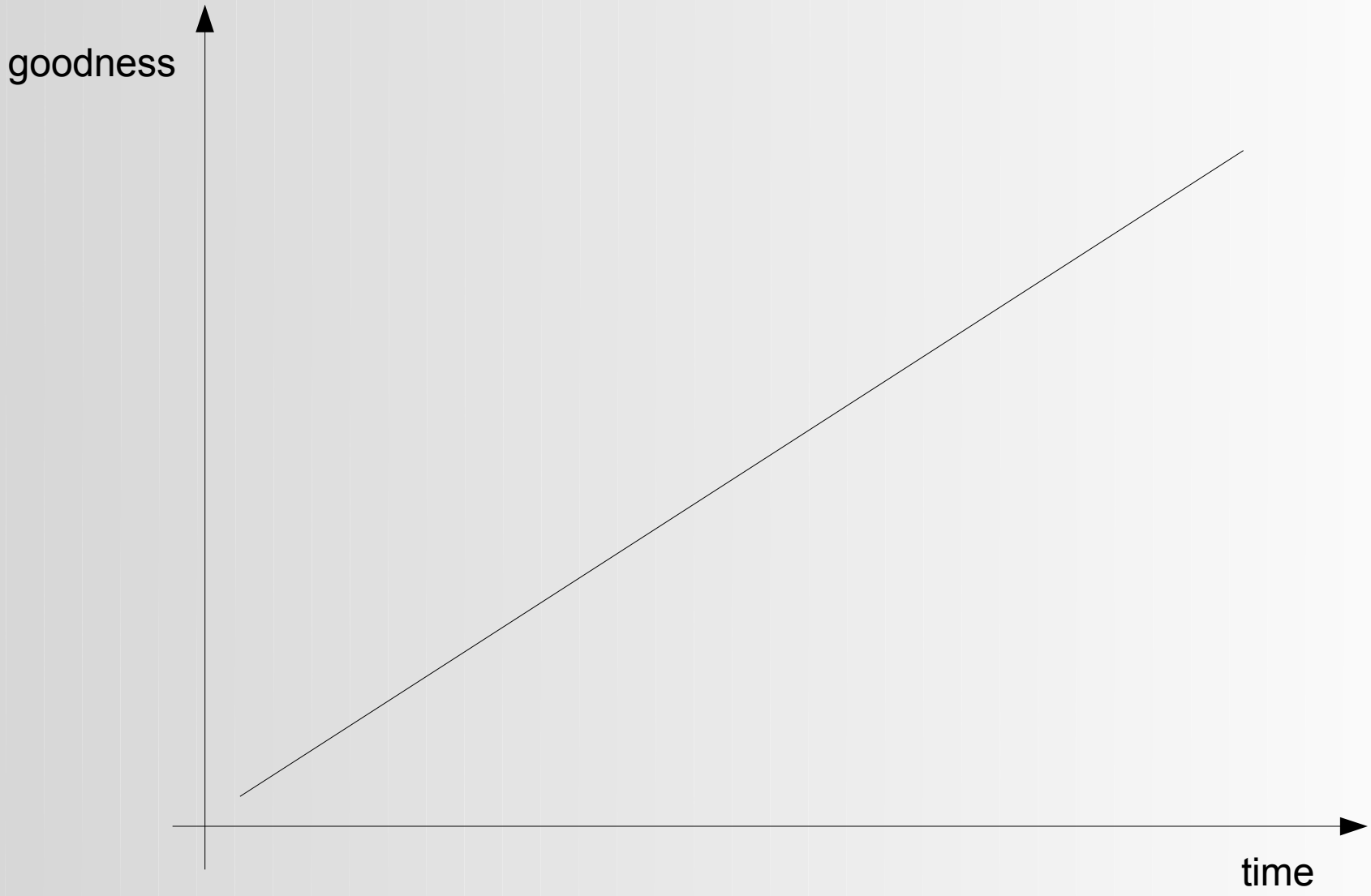
dataflow perspectives

determinacy – the structure of computation

research

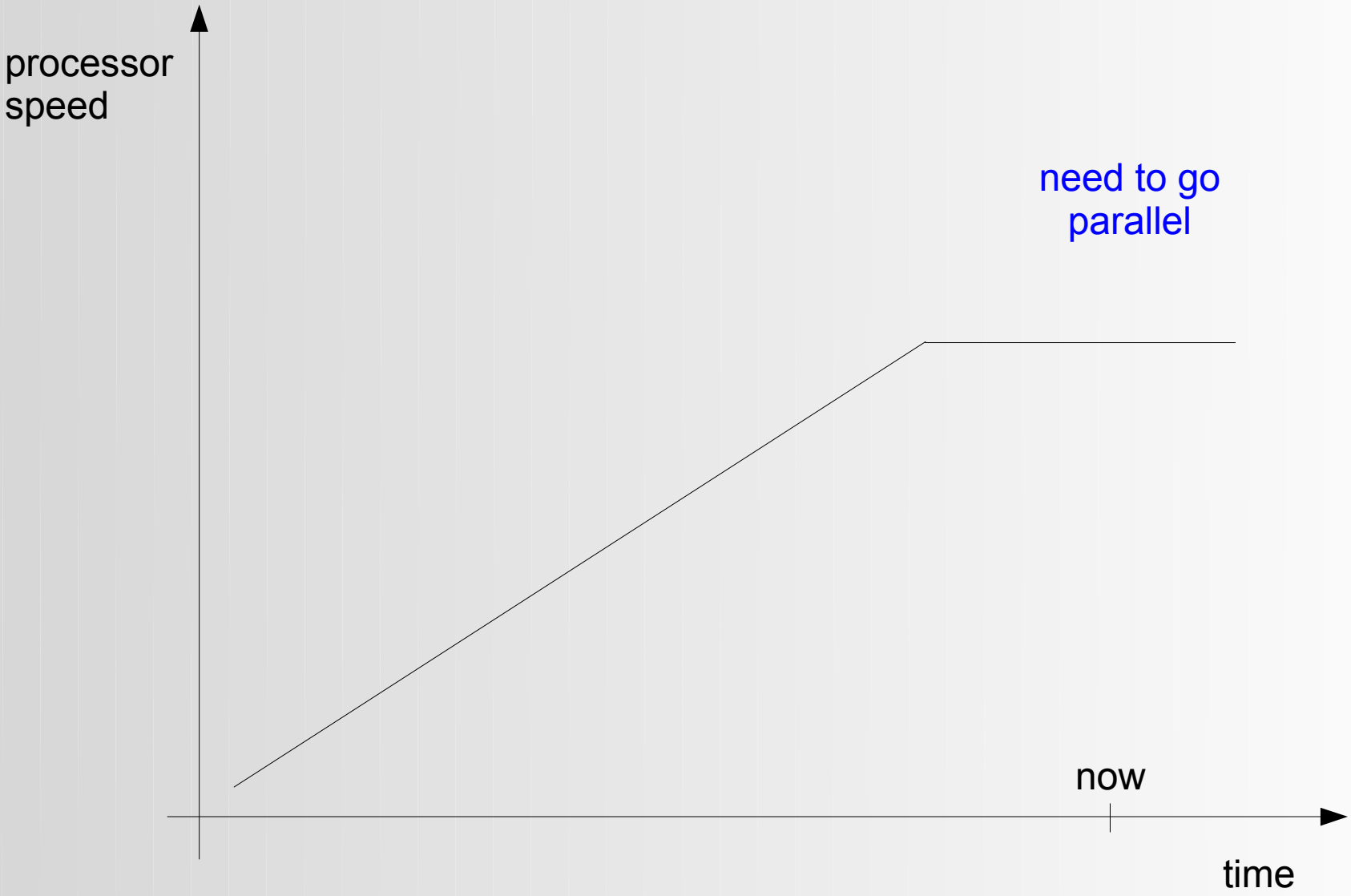
the once and future research agenda

Moore's law



(Borrowed from Seth Goldstein.)

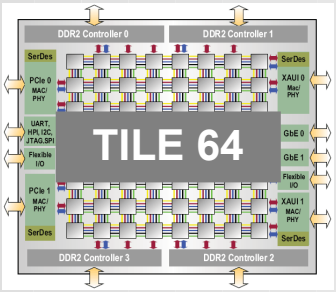
processor speed



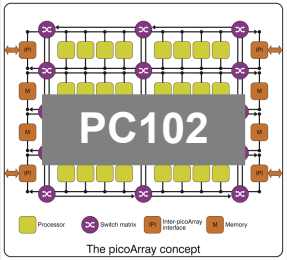
programming parallel machines

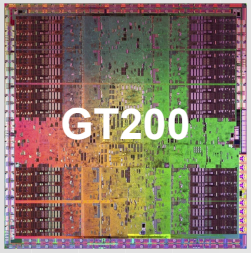
SOPM (“ajava”)

C w/ “iLib”



C + VHDL



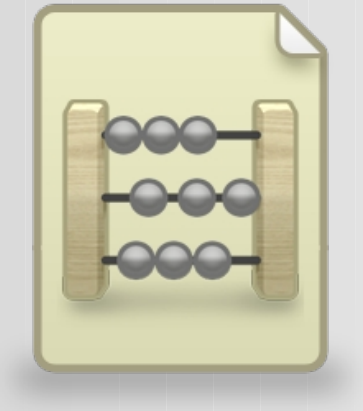


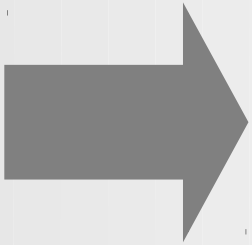
CUDA

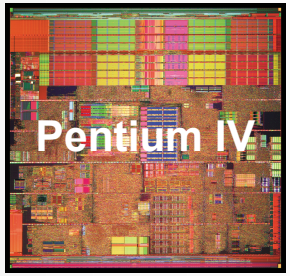
HDL

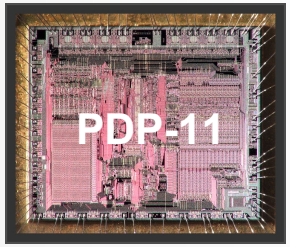
VentureForth

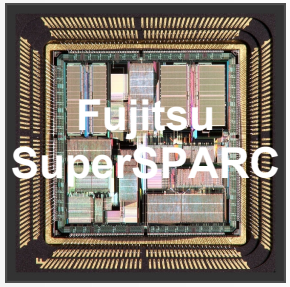
programming sequential machines

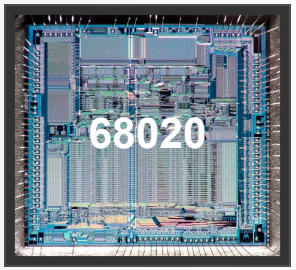












programming sequential machines

C,
Pascal,
Forth,
Java,
Lisp,
Fortran,
Haskell,
...

"von Neumann" machine

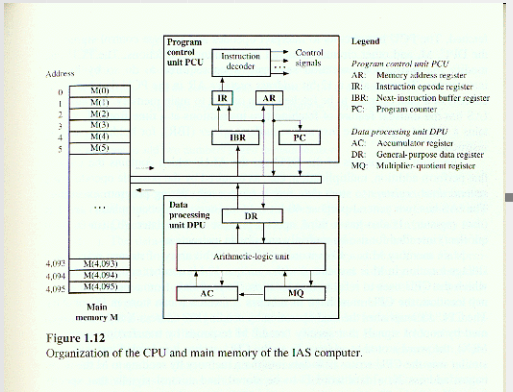
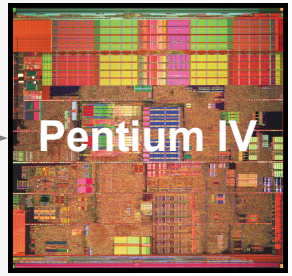
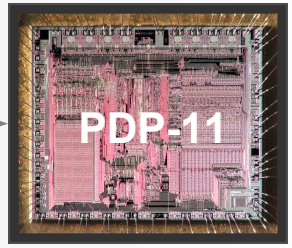
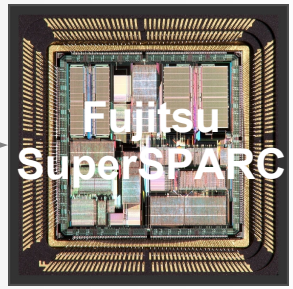
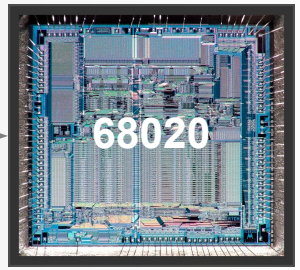


Figure 1.12
Organization of the CPU and main memory of the IAS computer.









a pivotal metaphor

programmers

compiler writers

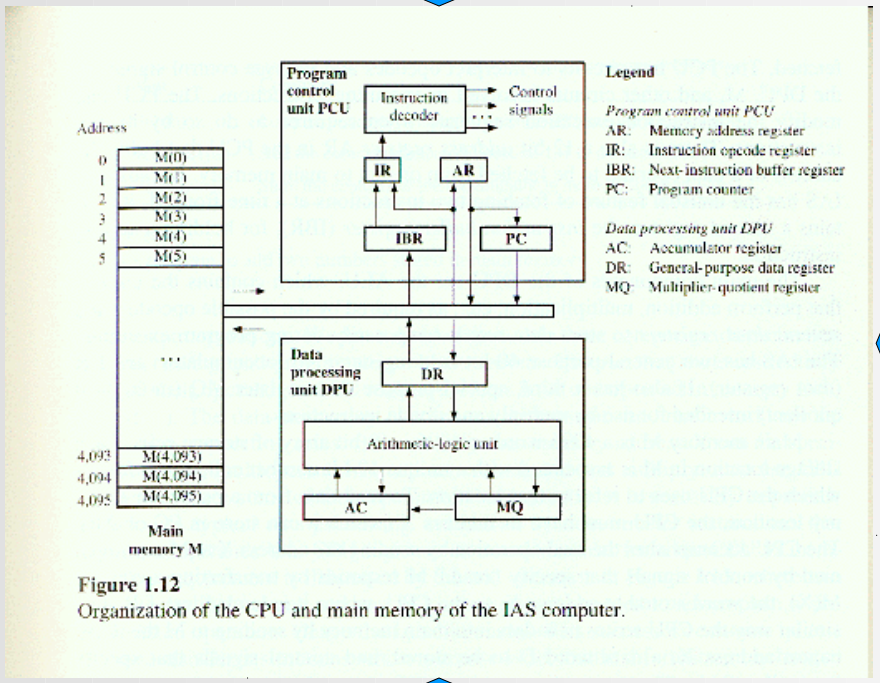
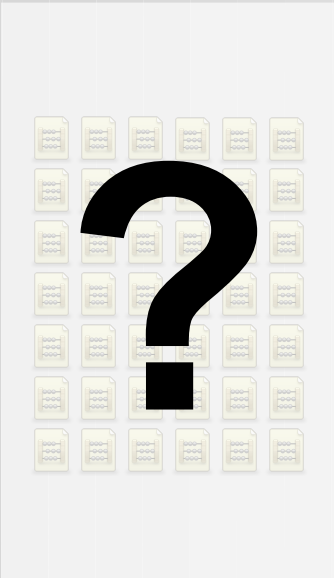


Figure 1.12
Organization of the CPU and main memory of the IAS computer.

computer architects

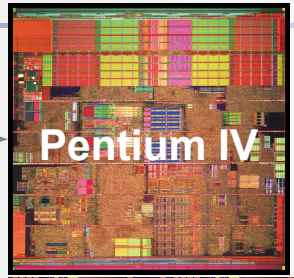
language designers

programming parallel machines

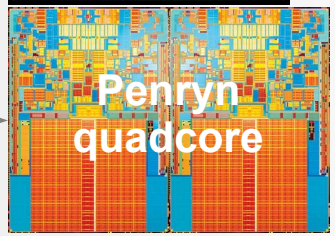


?

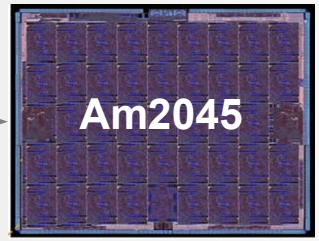
?

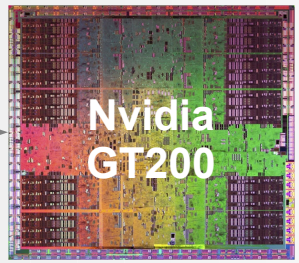


processor
1

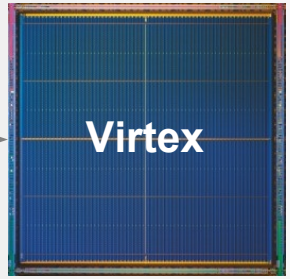


multi-core
2 ~ 10s



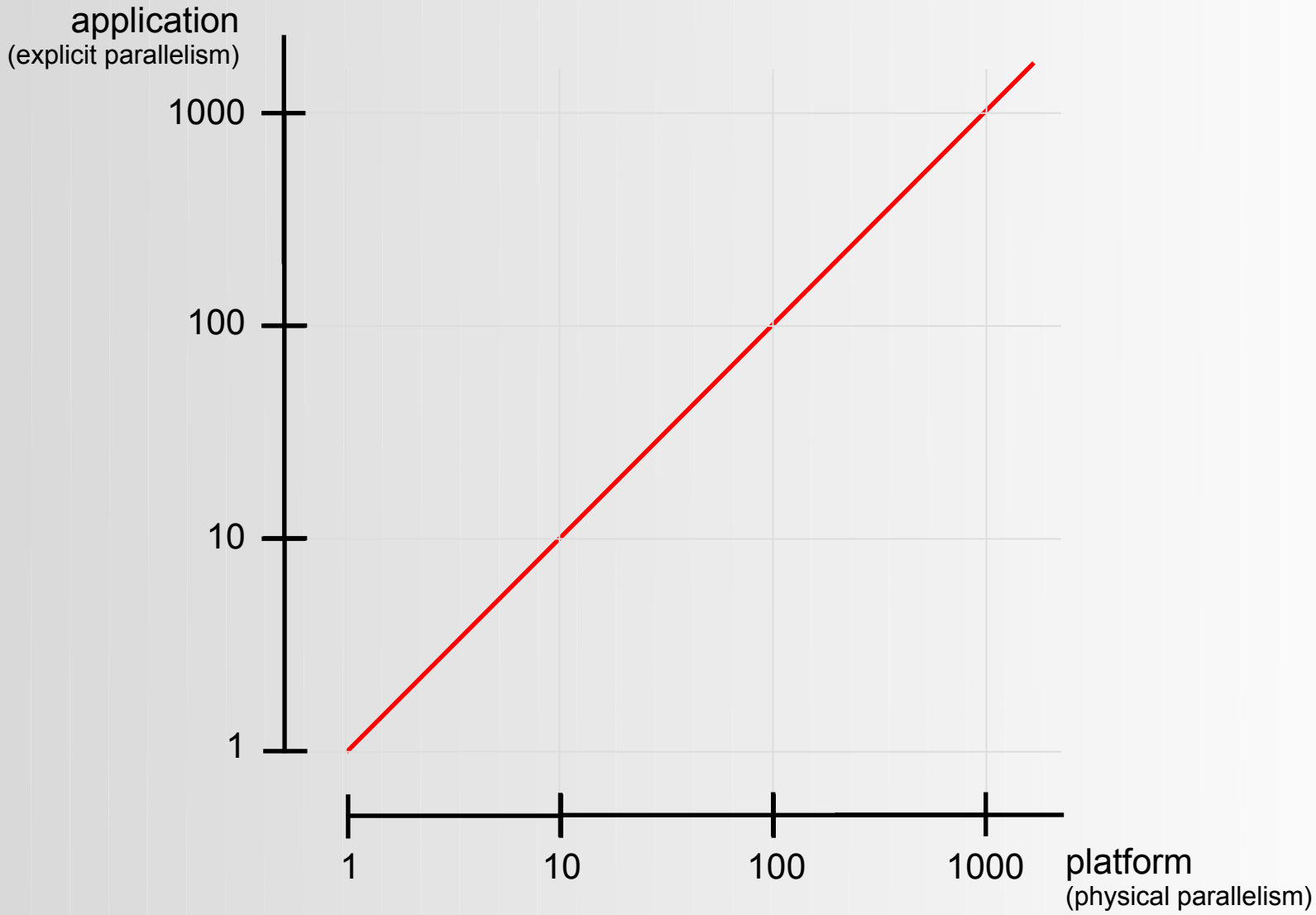


MPPA, GPU
10s ~ 100s ...

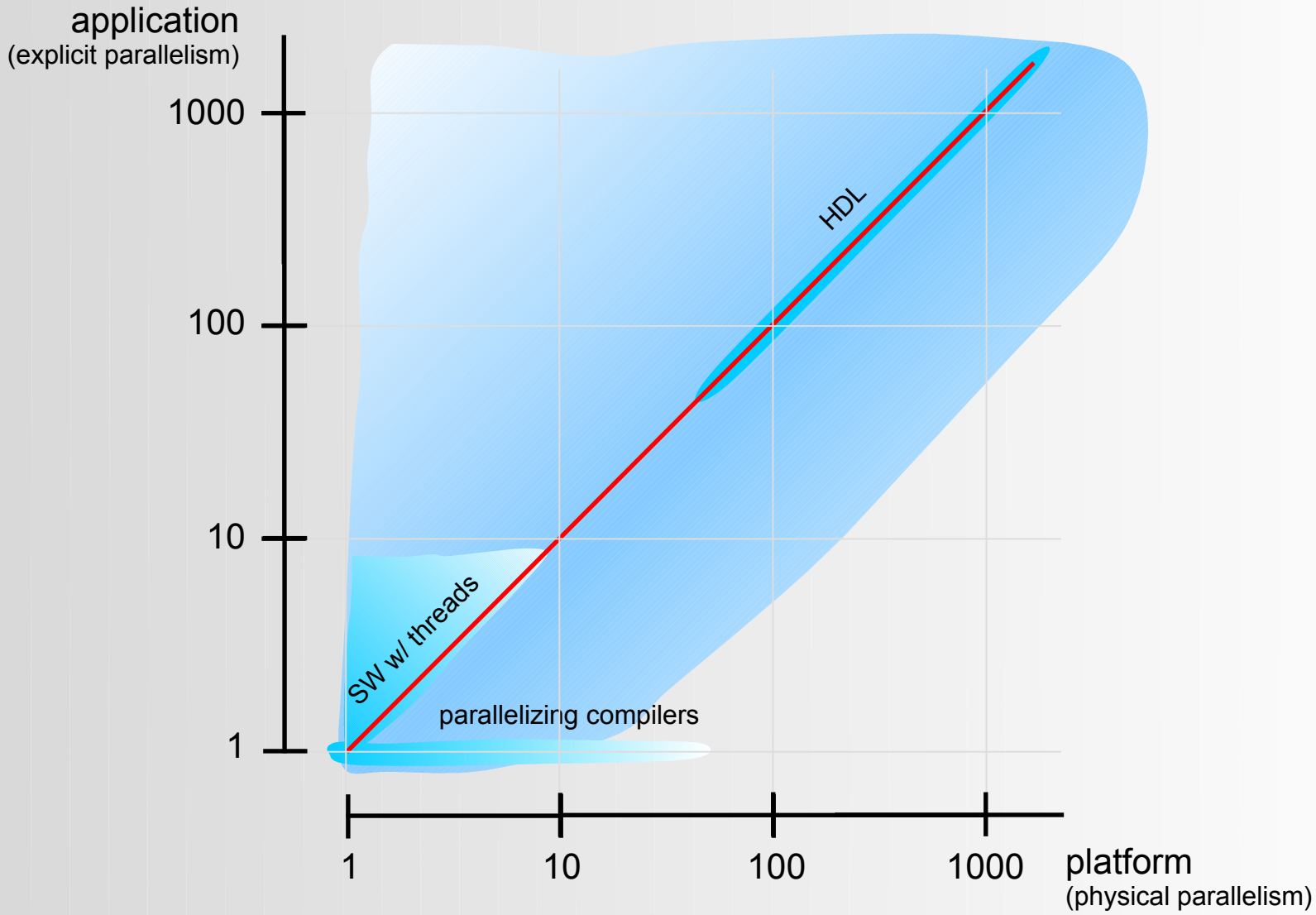


FPGA
100s ~ 1000s

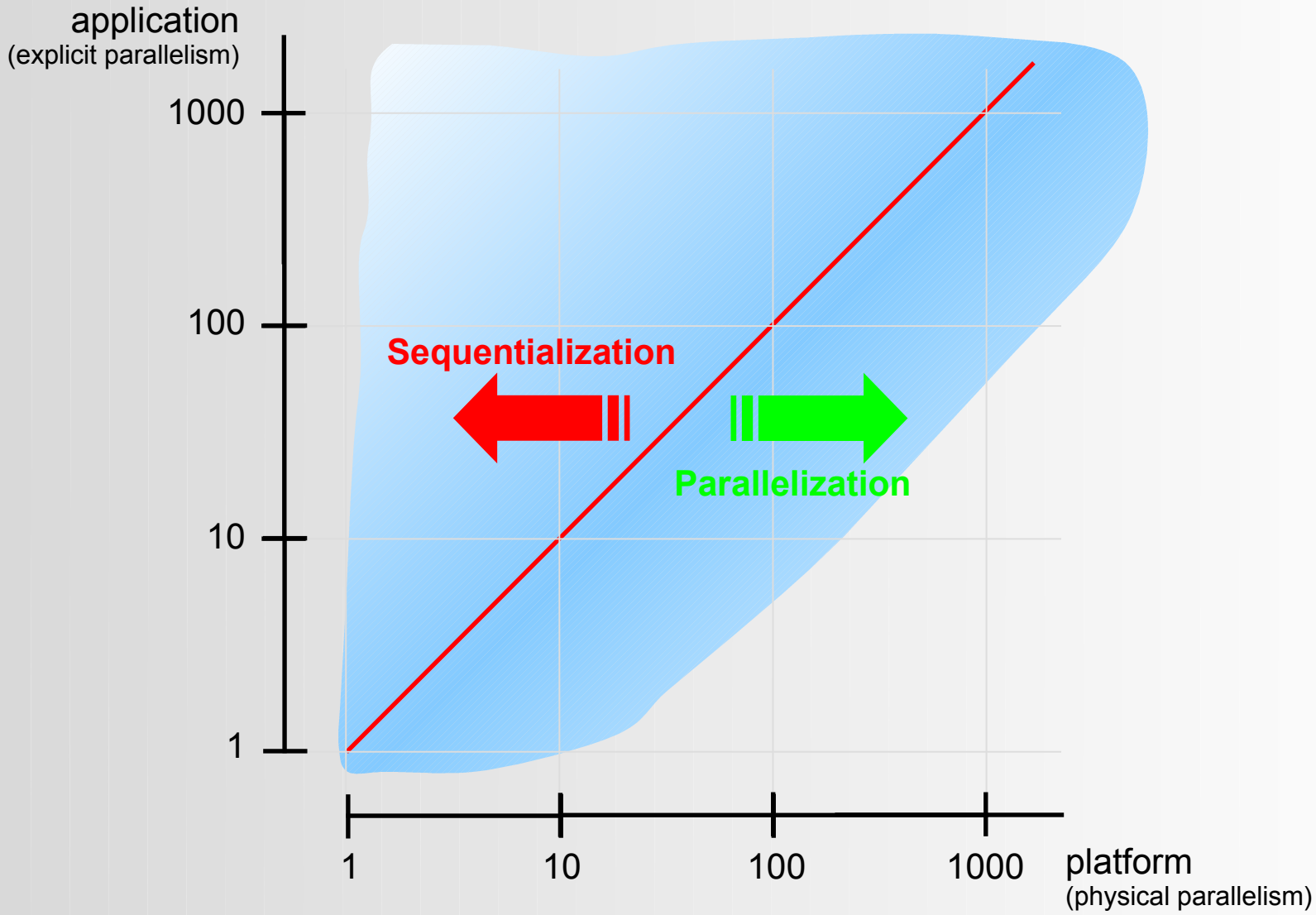
portable concurrency



portable concurrency



portable concurrency



motivation

the need for a parallel programming model

dataflow programming

actors, dataflow, and the CAL actor language

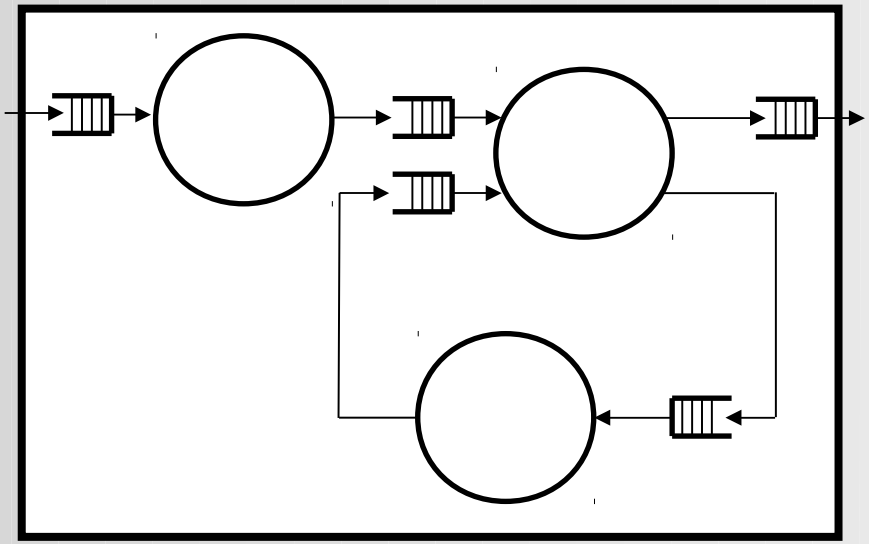
dataflow perspectives

determinacy – the structure of computation

research

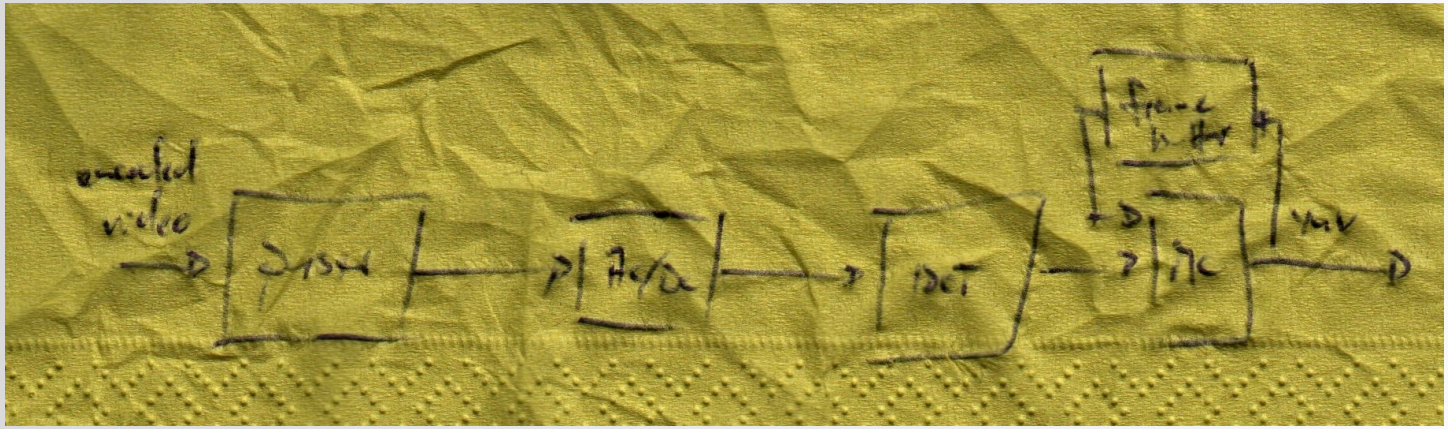
the once and future research agenda

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

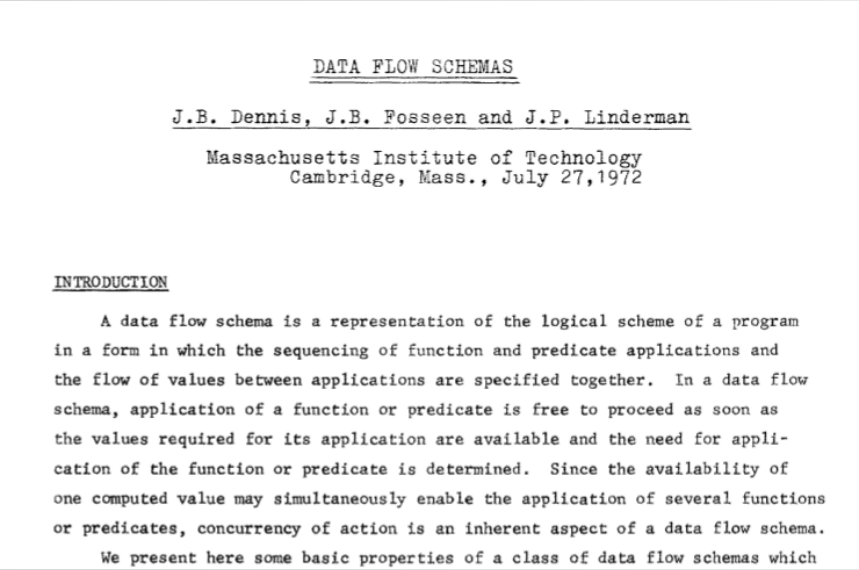
not exactly a new model...



domain expert's lunchtime rendition of a video decoder

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

data flow schemas (Dennis et al.)



DATA FLOW SCHEMAS

J.B. Dennis, J.B. Fosseen and J.P. Linderman

Massachusetts Institute of Technology
Cambridge, Mass., July 27,1972

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, 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

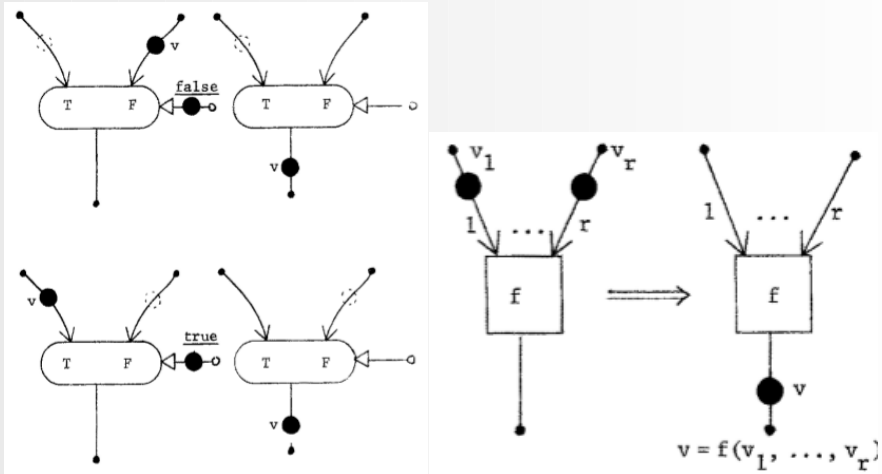


Figure 4. Firing rules for data flow schemas.

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)

INFORMATION PROCESSING 74 – NORTH-HOLLAND PUBLISHING COMPANY (1974)

THE SEMANTICS OF A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING

Gilles KAHN

IRIA-Laboria, Domaine de Voluceau, 78150 Rocquencourt, France

and

Commissariat à l'Energie Atomique, France

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 schemata 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.

```
 Begin
(1) Integer channel X, Y, Z,
(2) Process f(integer  in U,V
     Begin integer I ; logica
           B := true ;
           Repeat Begin
(4)          I := if B then w
(7)          print (I) ;
(5)          send I on W ;
             B := ¬B ;
             end ;
     End ;
   Process g(integer in U ; i
     Begin integer I ; logica
       B := true ;
       Repeat Begin
         I := wait (U) ;
         if B then send I on
         B := ¬B ;
         End ;
     End ;
(3) Process h(integer in U;integer out V; integer INIT);
     Begin integer I ;
       send INIT on V ;
       Repeat Begin
         I := wait(U) ;
         send I on V ;
         End ;
     End ;
  Comment : body of mainprogram ;
(6) f(Y,Z,X) par g(X,T1,T2) par h(T1,Y,0) par h(T2,Z,1
  End ;
```

Fig.1. Sample parallel program S.

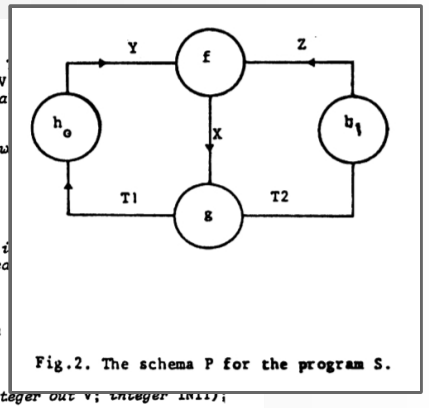


Fig.2. The schema P for the program S.

i) To every line e, of type D_e, associate a variable X_e 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 :

$$\begin{cases} X_1 = i_1 \\ \vdots \\ X_k = i_k \end{cases}$$

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 in Σ_p :

$$\begin{cases} X'_1 = f_1(X_1 \ldots X_n) \\ \vdots \\ X'_p = f_p(X_1 \ldots X_n) \end{cases}$$

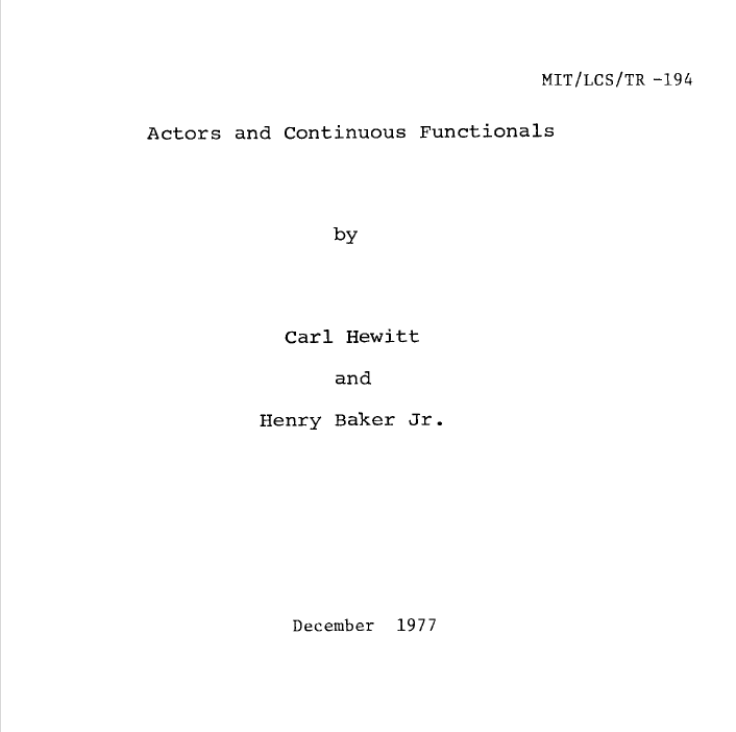
networks of *stream functions*

… described as communicating tasks with blocking reads

precise criterion for determinacy (*prefix-monotonicity*)

formal *fixpoint semantics*

actors (Hewitt et al.)



MIT/LCS/TR -194

Actors and Continuous Functionals

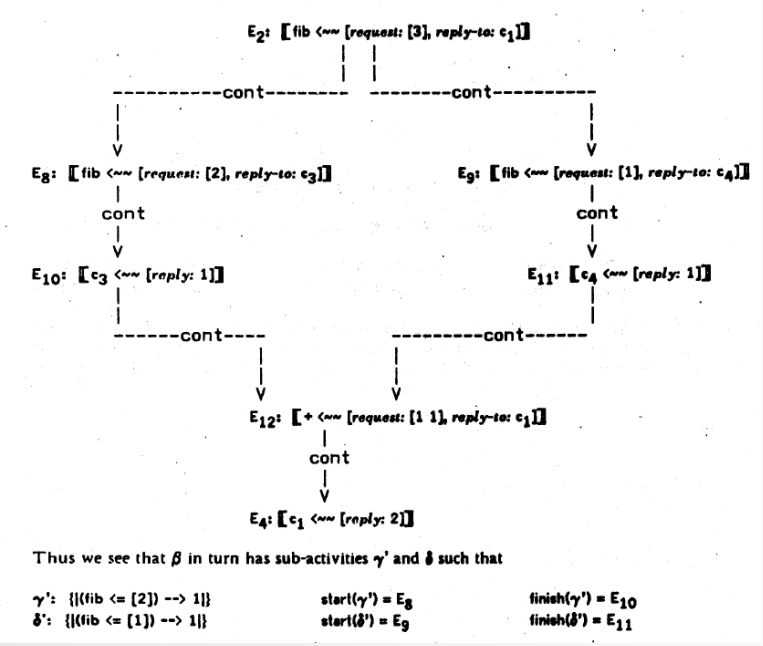
by

Carl Hewitt

and

Henry Baker Jr.

December 1977



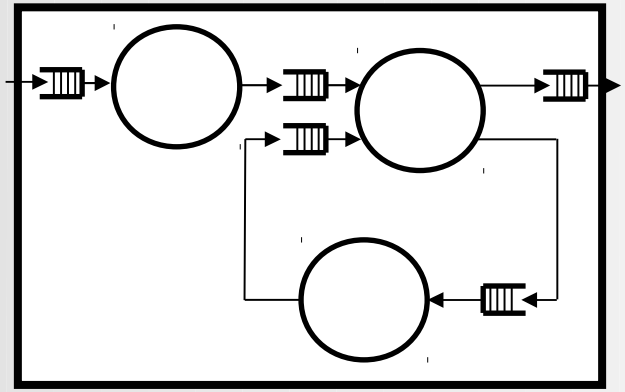
stateful, active objects (actors)

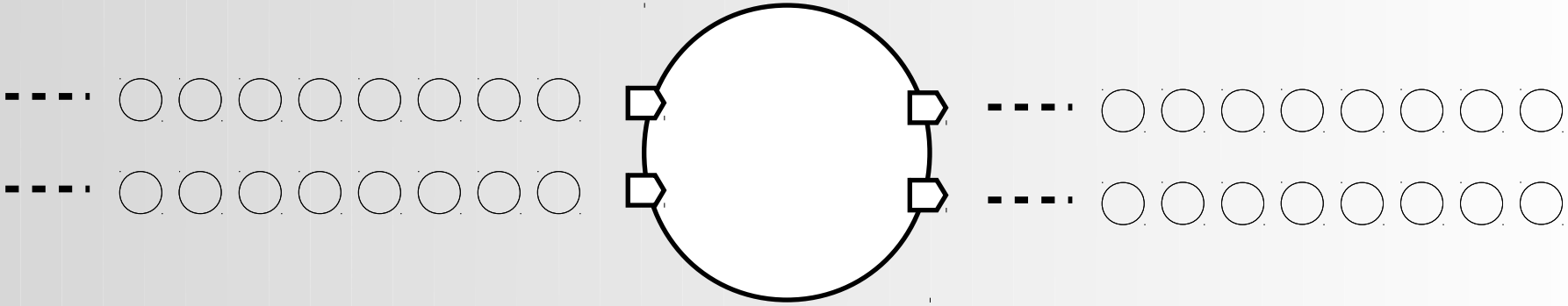
communicating via message passing

non-determinism and partial orders

functional actors with fixpoint semantics

actors as stream operators

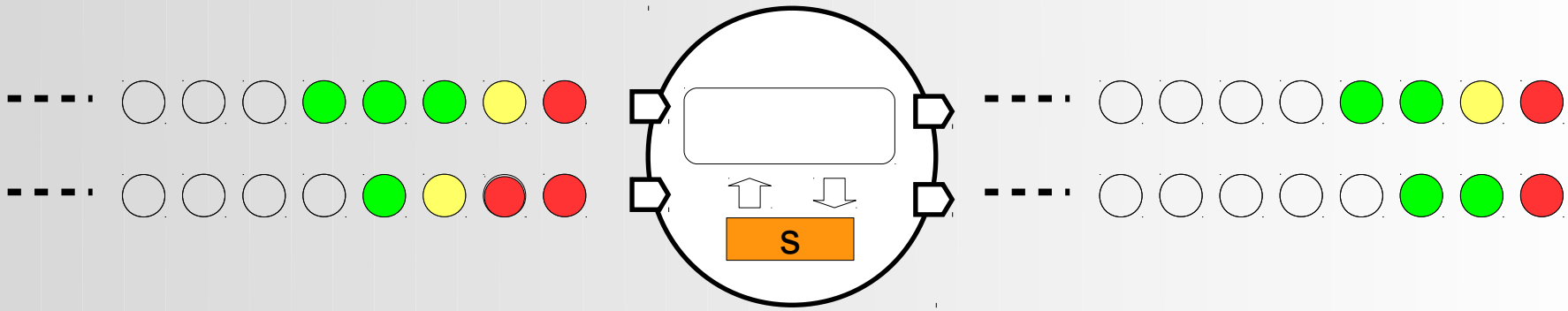


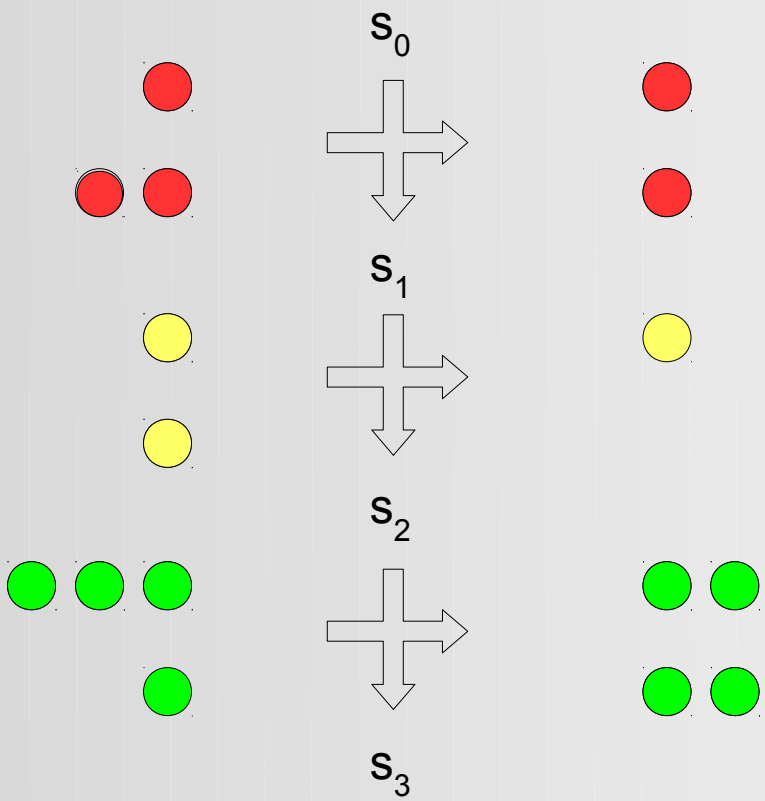


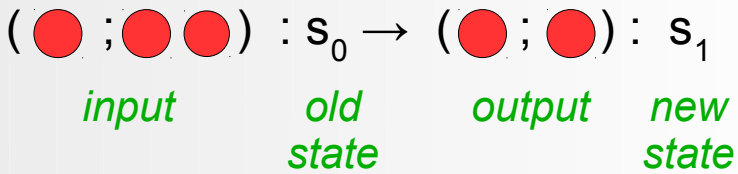
actors operate on **infinite** objects

they need not terminate

dataflow with firing

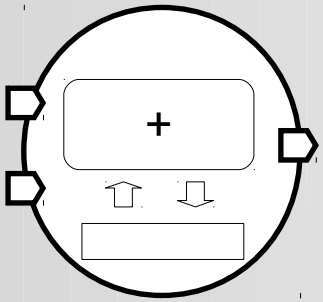
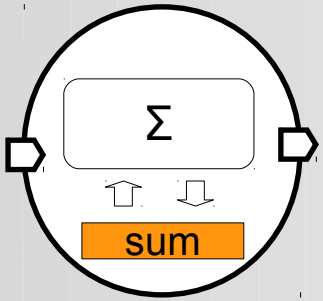
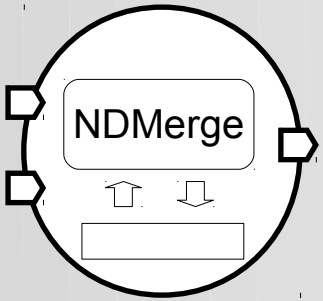






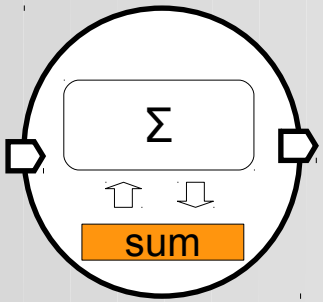
actor transition (**firing**) may
- consume input tokens
- produce output tokens
- modify the state

examples

	example transitions	*example runs*
	$(1\,;2):\circ \;\rightarrow\; 3:\circ$ $(4\,;-9):\circ \;\rightarrow\; -5:\circ$	3 4 -1 1 2 ... 7 -9 7 2 0 ... 10 -5 6 3 2 ...
	$2:5 \;\rightarrow\; 7:7$ $-9:7 \;\rightarrow\; -2:-2$	7 -9 7 2 0 ... 7 -2 5 7 7 ...
	$(1;\varepsilon):\circ \;\rightarrow\; 1:\circ$ $(\varepsilon\,;c):\circ \;\rightarrow\; c:\circ$	1 2 3 ... a b c ... 1 2 3 a b c ... or a b c 1 2 3 ... or 1 a 2 b 3 c ... or ...

writing actors in CAL: actions

large number of transitions – cannot list them explicitly



2 : 5 → 7 : 7

-9 : 7 → -2 : -2

transitions are of the form

a : sum → a+sum : a+sum

input *old state* *output* *new state*

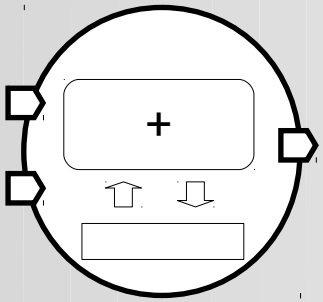
An **action** is a description of a *family of transitions*:

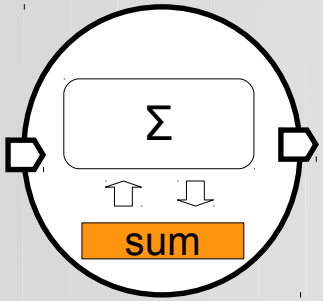
```
action A: [a] ==> X: [sum]
do
    sum := sum + a;
end
```

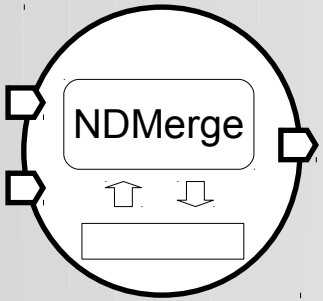
1. introduces variables for the input tokens
2. describes what happens to the state
3. specifies the values of output tokens

writing actors in CAL

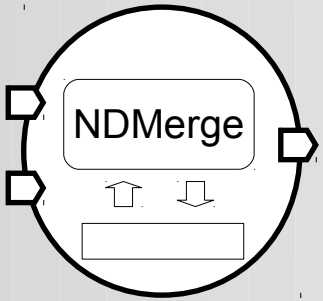
	example transitions	*code*

	$(1\,;2):\circ \;\rightarrow\; 3:\circ$
	$(4\,;-9):\circ \;\rightarrow\; -5:\circ$
		`actor Add () A, B ==> X:`
		`  action A: [a], B: [b] ==> X: [a + b] end`
		`end`

	$2:5 \;\rightarrow\; 7:7$
	$-9:7 \;\rightarrow\; -2:-2$
		`actor Sum () A ==> X:`
		`  sum := 0;`
		`  action A: [a] ==> X: [sum]`
		`  do`
		`  sum := sum + a;`
		`  end`
		`end`

	$(1;\varepsilon):\circ \;\rightarrow\; 1:\circ$
	$(\varepsilon\,;2):\circ \;\rightarrow\; 2:\circ$
		`actor NDMerge () A, B ==> X:`
		`  action A: [v] ==> X: [v] end`
		`  action B: [v] ==> X: [v] end`
		`end`

priorities



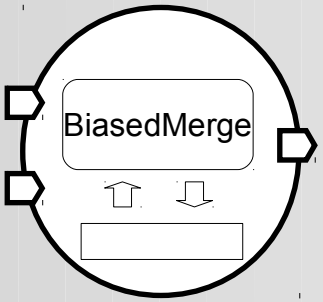
$(1; \varepsilon) : \circ \;\rightarrow\; 1 : \circ$

$(\varepsilon ; 2) : \circ \;\rightarrow\; 2 : \circ$

```
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.



$(1; \varepsilon) : \circ \;\rightarrow\; 1 : \circ$

$>$

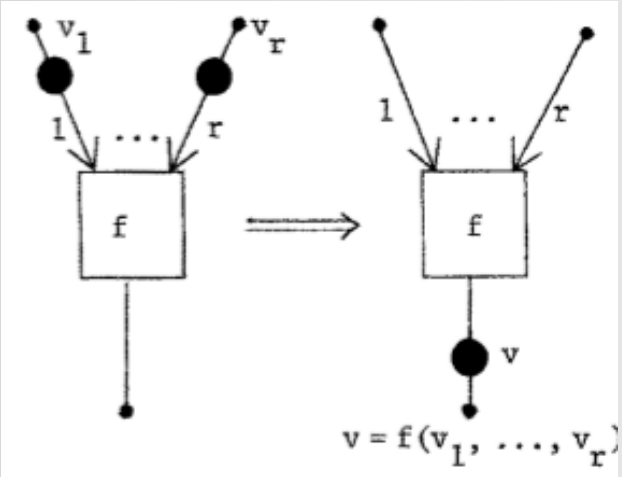
$(\varepsilon ; 2) : \circ \;\rightarrow\; 2 : \circ$

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

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

    priority CopyA > CopyB; end
end
```

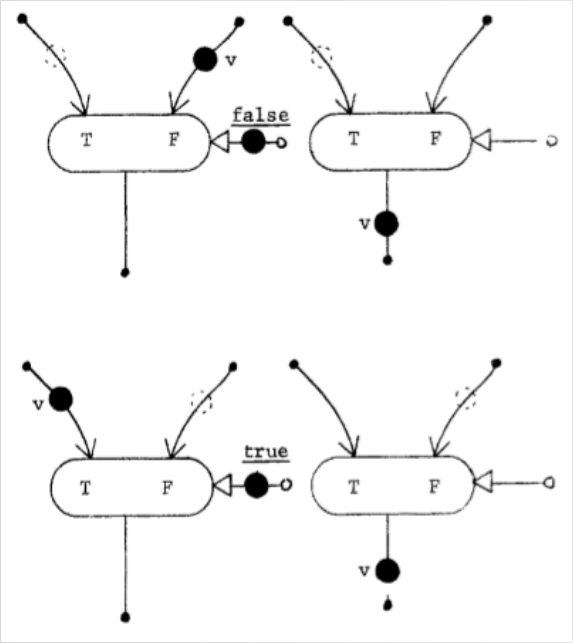
An actor is a *partially ordered* set of transitions.

An *actor description* contains a partially ordered set of *actions*.

historical note (CAL and Dennis data flow)



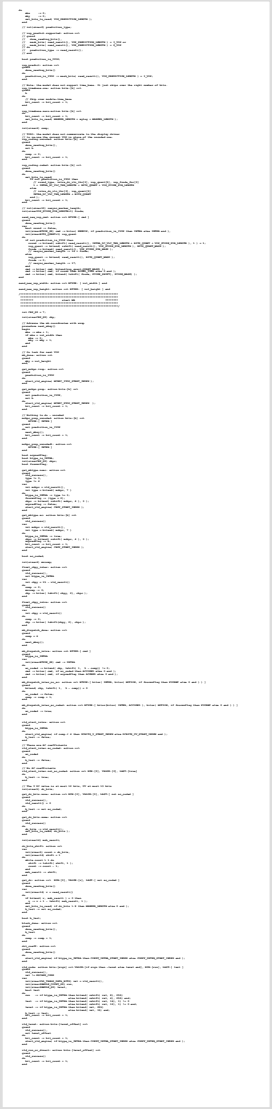

```
actor F () A, B ==> X:

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

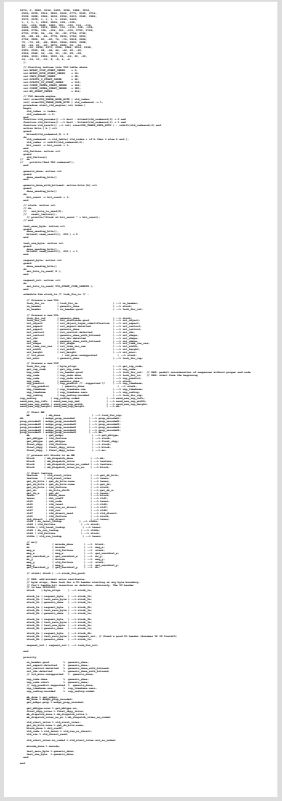



```
actor Switch () S, T, F ==> X:

    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)

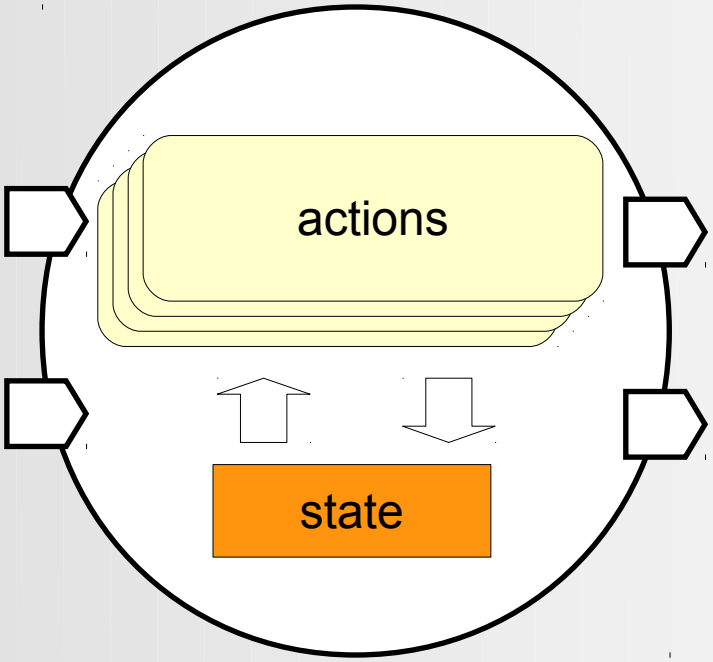




ParseHeaders
1320 lines
(without header comments)

http://opendf.svn.sourceforge.net/viewvc/opendf/models/MPEG4_SP_Decoder/

actors and actions

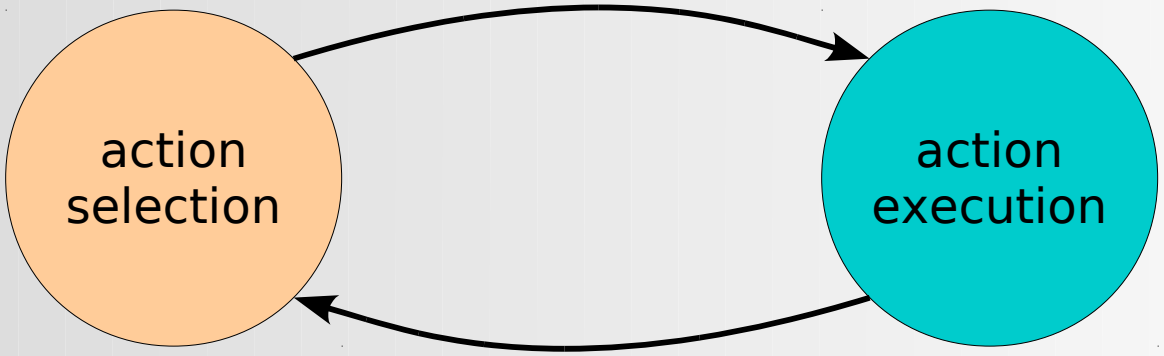


action – describes a family of potential actor transitions

actor run / **execution** – a sequence of action executions

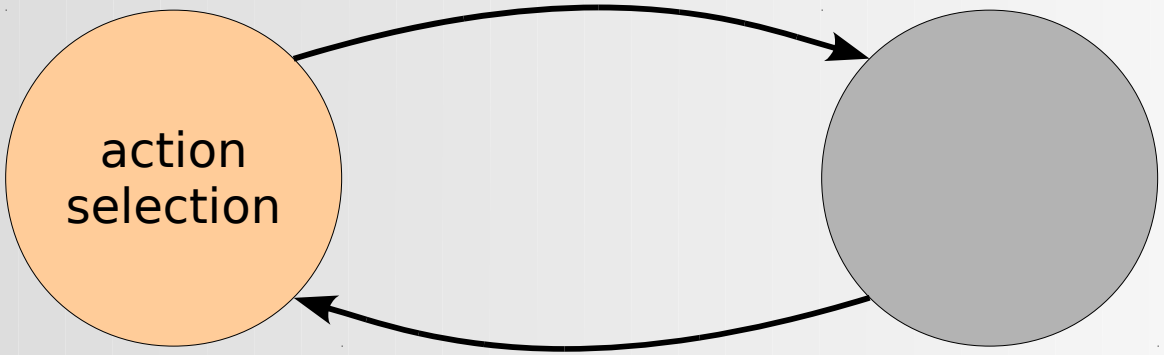
action selection – the process of picking the next action to execute

actor execution model



action selection

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

```
n = 0;
```

Thread 1:

```
t1 = n;
n = t1 + 1;
```

Thread 2:

```
t2 = n;
n = t2 + 1;
```

thread race on shared state

Each thread is determinate.

Their composition is not.

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:

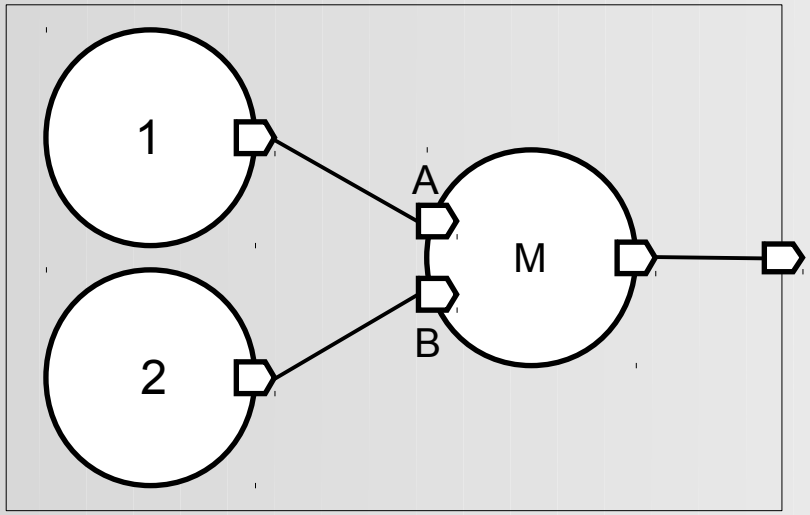
There is only one action.

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

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

Actions are totally priority-ordered.

```
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




```
actor Constant1 () ==> X:

    action ==> X: [1] end
end
```

```
actor Constant2 () ==> X:

    action ==> X: [2] end
end
```

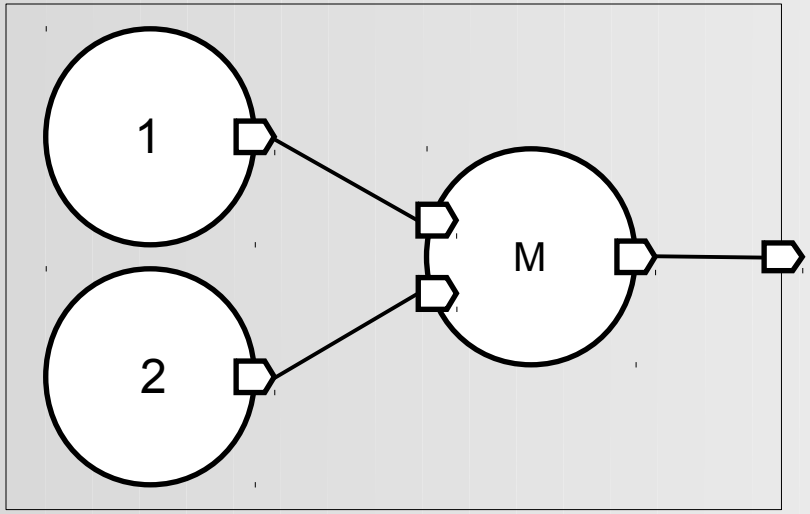
```
actor BiasedMerge () A, B ==> X:

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

    priority CopyA > CopyB; end
end
```

firing sequence	**output**
2, 1, M, M, ...	1, 2, ...
2, M, 1, M, ...	2, 1, ...

In dataflow, too, determinacy is not compositional.

Can we find a property of actors that **is** compositional, and that entails determinacy?

source of nondeterminism



firing sequence	output
2, 1, M, M, ...	1, 2, ...
2, M, 1, M, ...	2, 1, ...

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*?

timing independence

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

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

If the B-token is available, but the A-token is late, nothing happens:

The actor just waits for the A-token to arrive.

Here, if the A-token is late, the actor *does something else* instead. (In this case, the `CopyB` action.)

An *earlier arrival* of the A-token would have *disabled that action*.

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

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

    priority CopyA > CopyB; end
end
```

An actor is timing-independent[1] if *additional input tokens never disable an action*.

[1] aka “prefix-monotonic”

timing independence

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

```
action A: [v] ==> P: [v]
guard v >= 0
```

2. all guards are true

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

priority CopyA > CopyB; end
```

3. no higher-priority action is enabled

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.

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

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

    priority CopyA > CopyB; end
end
```

timing independence

timing-independent

```
actor TI () A ==> X:

    Hi: action A: [v] ==> X: [1] end
    Lo: action A: [v, w] ==> X: [0] end

    priority Hi > Lo; end
end
```

timing-dependent

```
actor TD () A ==> X:

    Hi: action A: [v, w] ==> X: [1] end
    Lo: action A: [v] ==> X: [0] end

    priority Hi > Lo; end
end
```

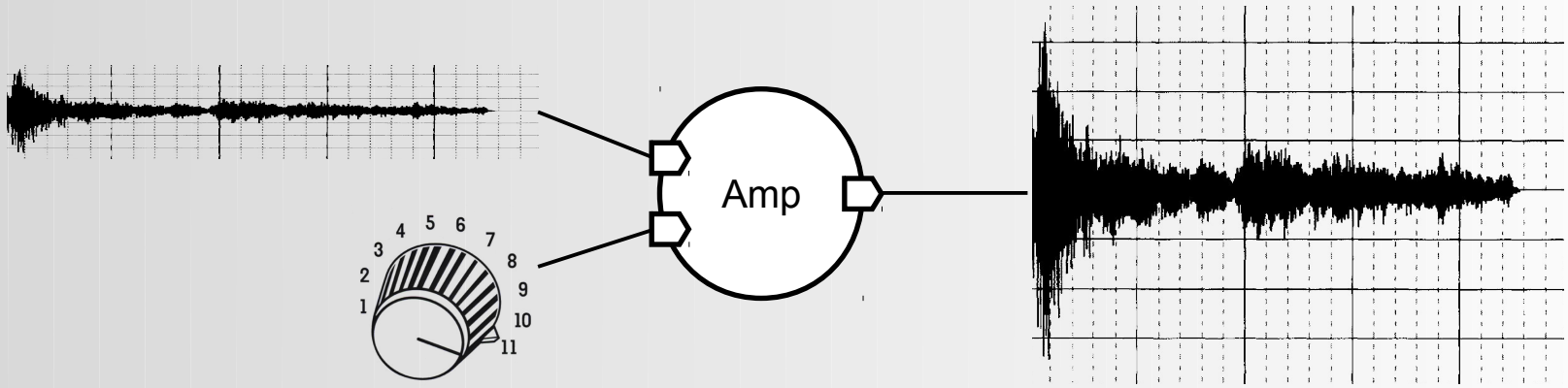
Looking at the input patterns and the priority order, we can derive guarantees about timing-independence of each actor.

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?




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

    vol := 1;

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

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

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

    vol := 1;

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

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

    priority V > D; end
end
```

a taxonomy of actors

timing-independent
(prefix monotonic)

timing-dependent

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

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

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

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

  priority CopyA > CopyB; end
end
```

determinate
(functional)

nondeterminate
(non-functional)

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

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

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

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

```
actor OddActor () A ==> X:

  High: action [a1, a2] ==> [a1+a2] end
  Low: action [a] ==> [a] end
  Low: action [a] ==> [-a] end

  priority High > Low; 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

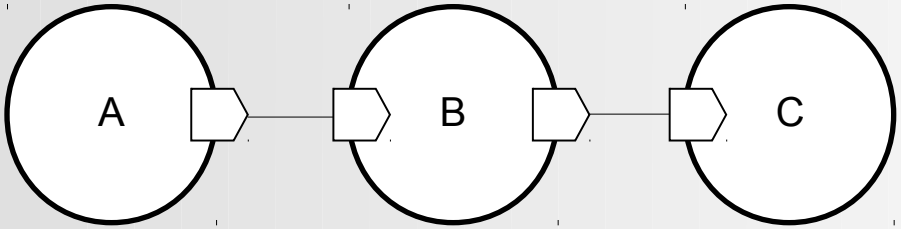
the once and future research agenda

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*

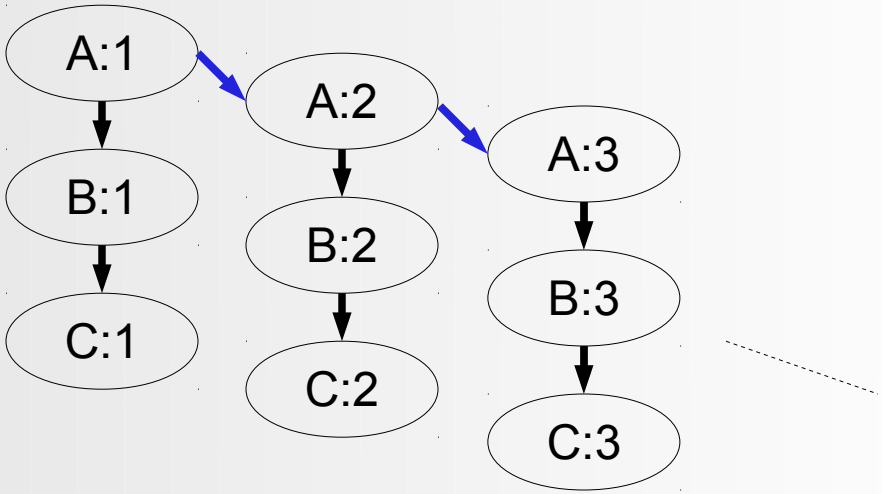
edges are *dependencies*

dependencies mediated by

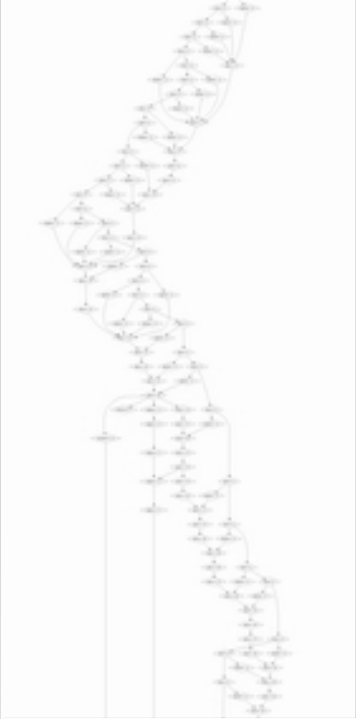
tokens

state variables

(ports)

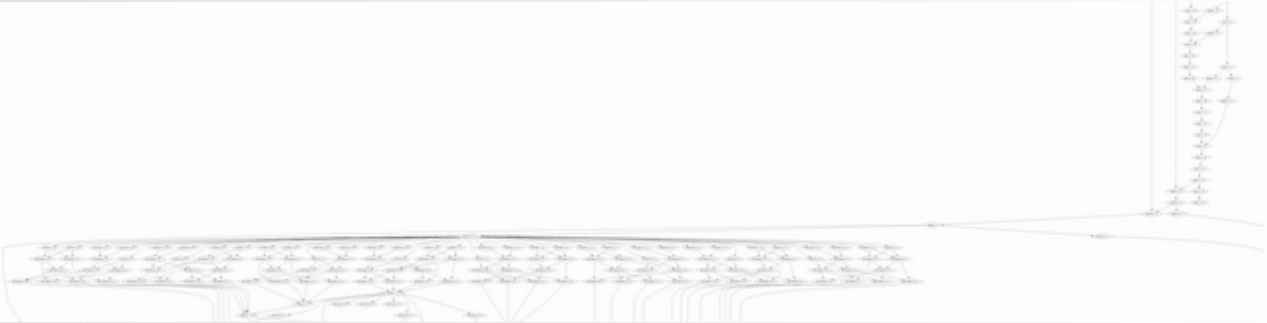


traces of a decoder

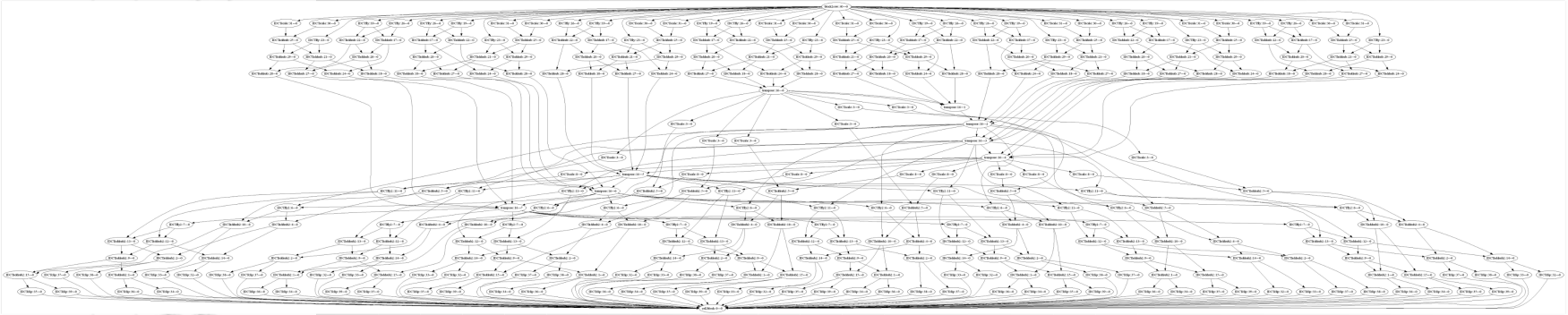


parsing

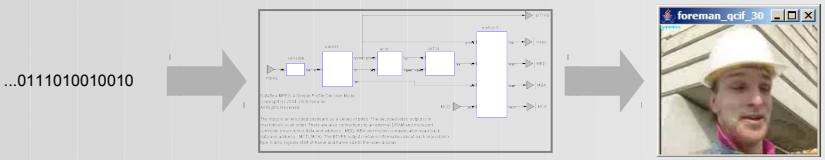
parsing, end of header

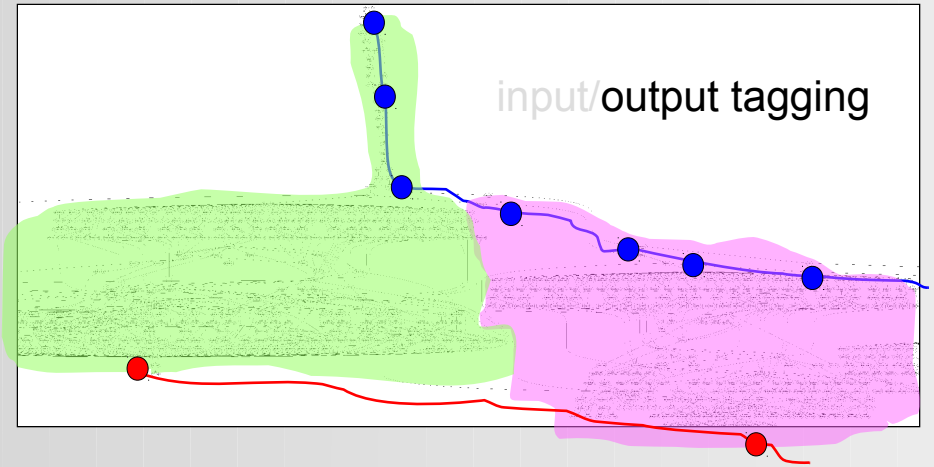


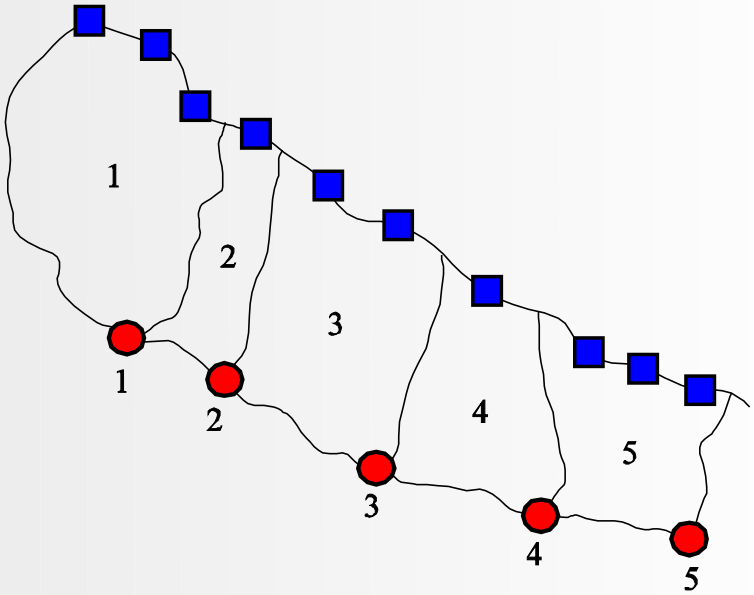
IDCT



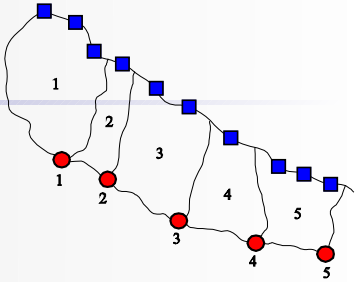
trace analysis



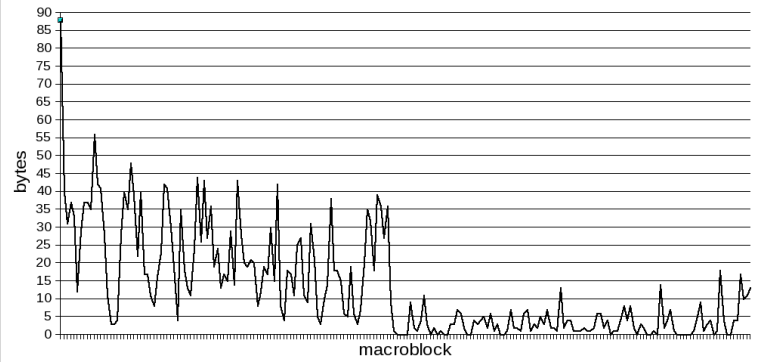




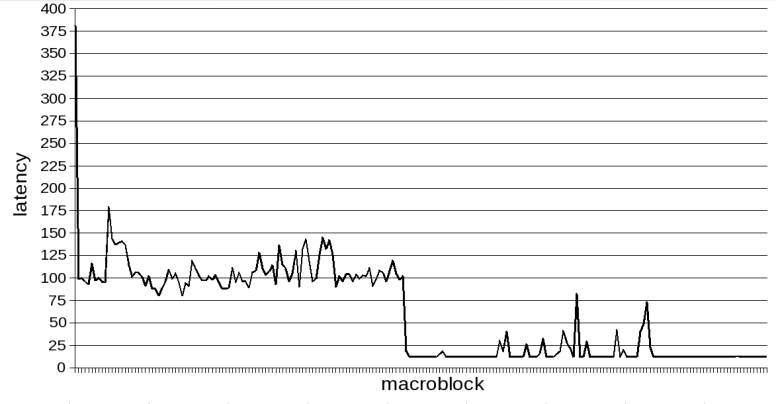
trace metrics (examples)



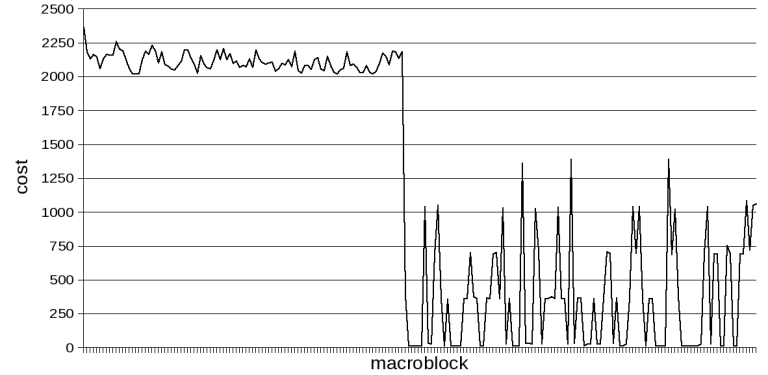
input
(bytes)



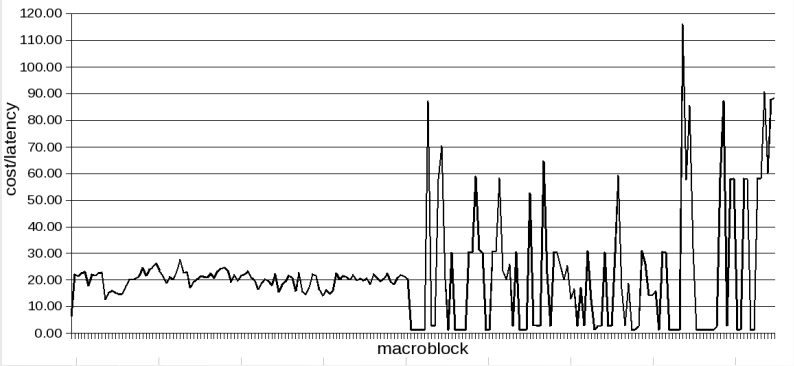
latency
(incremental)



cost
(Σ area * time)



parallelizability
(cost/latency)



motivation

the need for a parallel programming model

dataflow programming

actors, dataflow, and the CAL actor language

dataflow perspectives

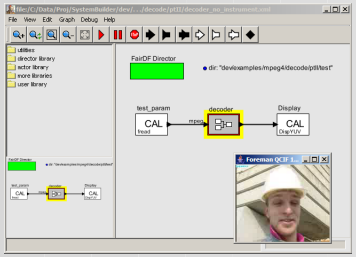
determinacy – the structure of computation

research

the once and future research agenda

Cal Actor Language history

MPEG RVC



SystemBuilder

ACTORS project

first hardware

MPEG-4 decoder...

... in hardware

... in software

RVC-CAL

2002	2003	2004	2005	2006	2007	2008	2009	2010

CAL

interpreter (Ptolemy)

CLR 1.0

OpenDF (SourceForge)

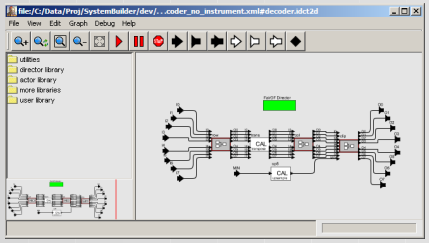
Eclipse plugin

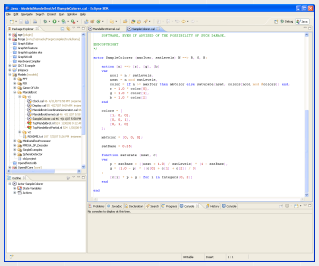
Graphiti

ORCC

open HDL codegenerator

Open Dataflow





basic questions (and current answers)

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

Johan Eker Ericsson Edward A. Lee UC Berkeley Marco Mattavelli EPFL Carl von Platen Ericsson Dave B. Parlour Tabula Ian Miller Overture Mickaël Raulet INSA Rennes Mathieu Wipliez INSA Rennes Christophe Lucarz EPFL Shuvra S Bhattacharyya U Maryland Karl-Erik Arzen LTH Ruirui Gu Microsoft Ghislain Roquier EPFL

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

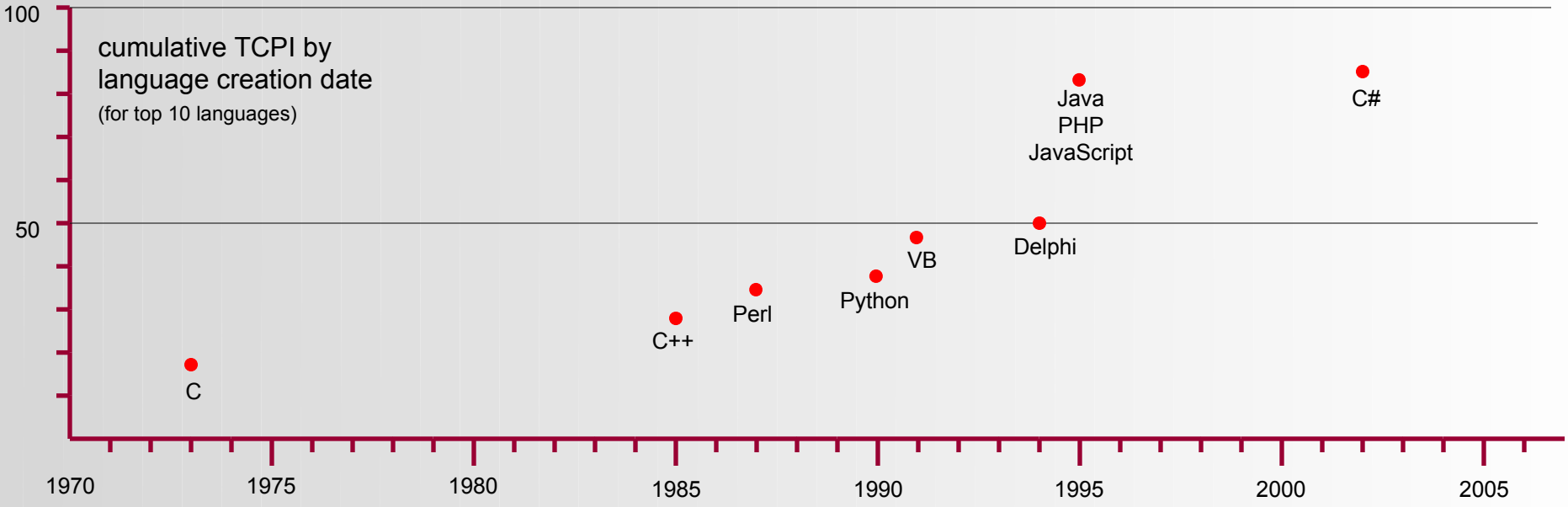
mpeg.chiariglione.org
MPEG

jwj@cs.lth.se

BACKUP

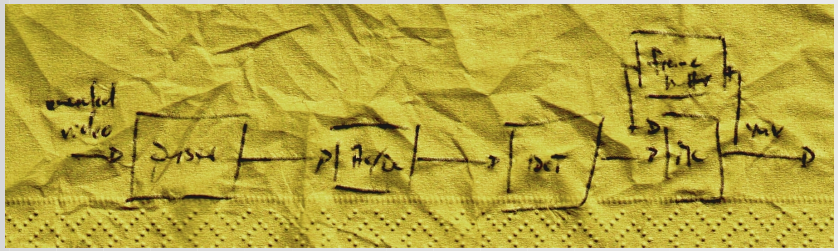
programming language adoption

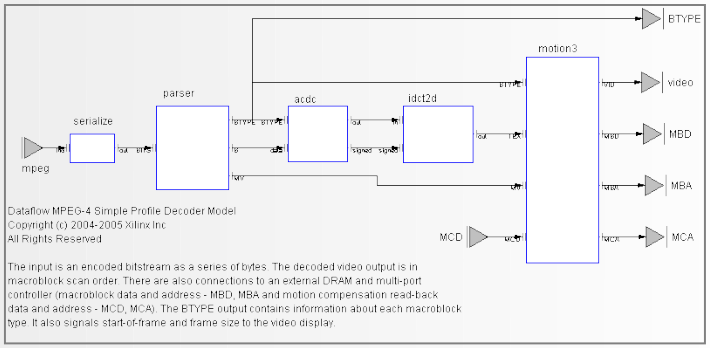
Name	TPCI	TPCI cum.	Year
C	17.66%	17.66%	1973
C++	11.06%	28.73%	1985
Perl	5.48%	34.20%	1987
Python	3.47%	37.67%	1990
VB	9.73%	47.40%	1991
Delphi	2.15%	49.54%	1994
Java	21.17%	70.72%	1995
PHP	9.86%	80.58%	1995
JavaScript	2.20%	82.78%	1995
C#	3.07%	85.85%	2002



source: TIOBE Programming Community Index, TPCI, October 2006, `http://www.tiobe.com/tpci.htm`

building (& specifying) a video decoder






```
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;
}
```