Extensions of Daedalus

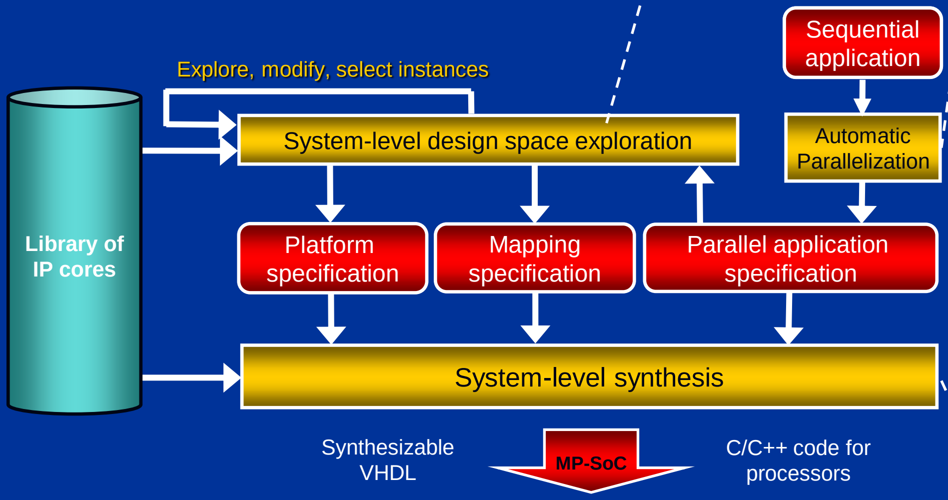
Todor Stefanov

Leiden Embedded Research Center,
Leiden Institute of Advanced Computer Science
Leiden University, The Netherlands

Overview of Extensions in Daedalus

- DSE limited to *best-effort* systems
  - Real-time behavior not guaranteed
- **Research Challenge**
  - Support for (Hard/Soft) Real-time
  - Support for Mixed-Criticality

- Parallelization limited to *static* programs
  - Program behavior known at compile time
- **Research Challenge**
  - Parallelize dynamic programs
  - Exact behavior unknown at compile time



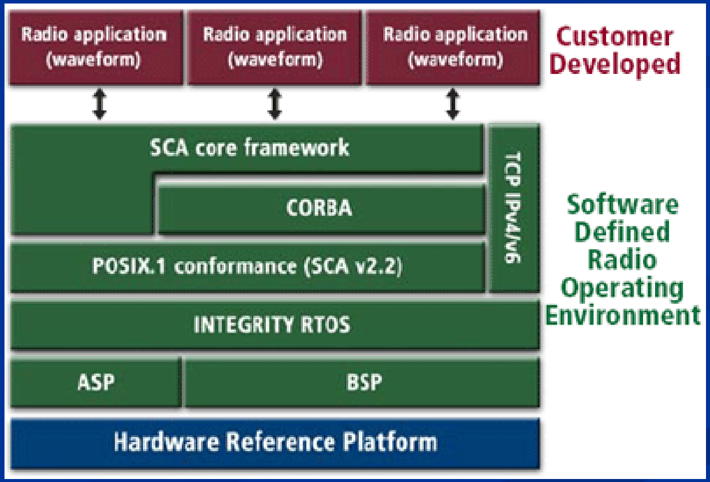
- Synthesis limited to *static* mapping
  - Task mapping fixed at design time
  - Cannot be changed at run-time
- **Research Challenge**
  - Run-time Adaptability support
  - SW/HW facilitating task migration
- Benefits:
  - Improved reliability/availability
  - Fault tolerance support
  - Efficient dynamic workload balance

Daedalus^{RT}: Automated Design of Hard-Real-Time Embedded Streaming MPSoCs

Introduction

- Complexity of modern applications is *increasing*
- This means that many systems require now:
  - Hard-real-time execution on MPSoC platforms
  - Running multiple applications on a single platform
  - Support for adding/removing applications at run-time

Interventional Radiology image filtering.
Source: Philips Healthcare



Software-Defined Radio Architecture.
Source: Green Hills Software Inc.

What is the Problem?

- How to design an embedded MPSoC that:
  - Runs multiple streaming applications simultaneously
  - Provides (Hard-) Real-time Quality of Service
    - Temporal isolation of applications
    - Strict timing deadline guarantees of tasks
  - Uses the minimum amount of resources
    - Processor
    - Memory

While minimizing the design time and effort?

Existing Solutions

- Existing design flows can be classified based on *QoS* and *multiple applications* support into:
  - Soft-real-time/Best-effort, single app/multiple apps
  - Hard-real-time, multiple apps

Use *DSE* to determine:
- min # of processors needed to schedule apps
- efficient mapping of tasks to processors

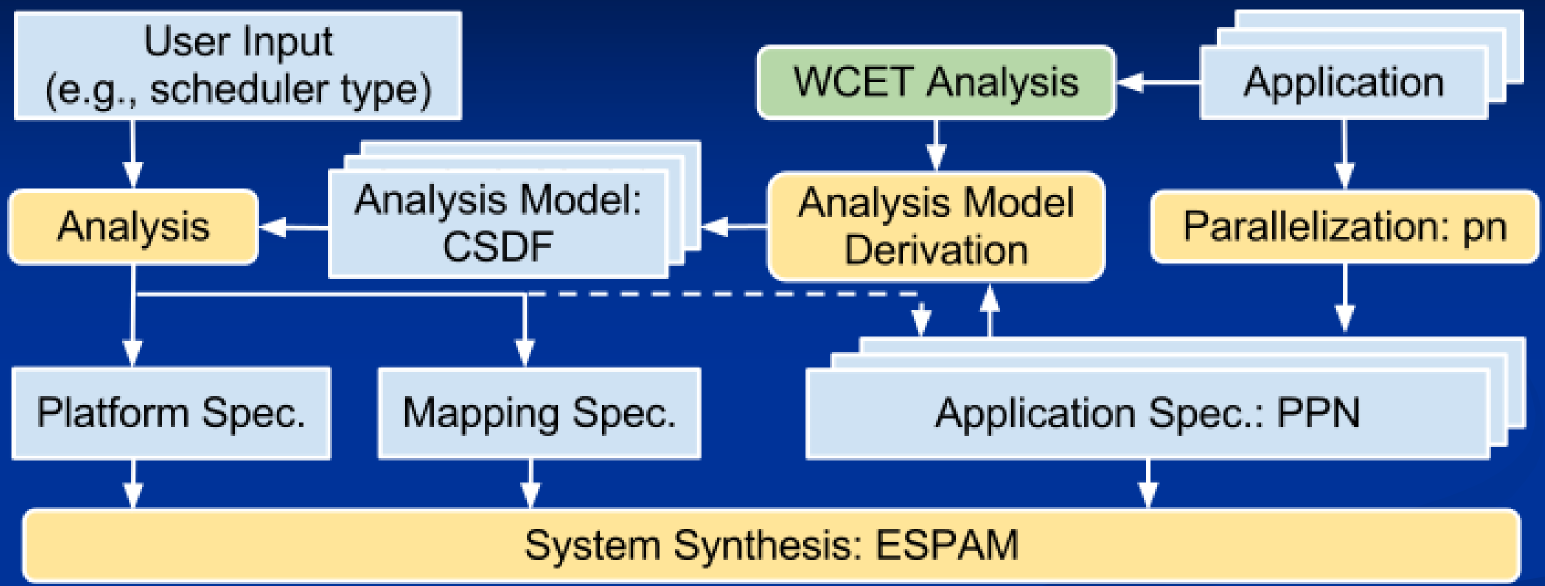
Very Complex and Time Consuming Approach!

Our Novel Answer to the Problem

- Utilize 40+ years of hard-real-time scheduling theory!
  - Bridge real-time scheduling and embedded MPSoC design
- Using hard-real-time scheduling theory and algorithms, we can:
  - Schedule apps while providing temporal isolation and hard-real-time QoS
  - Analytically determine min # of processors needed to schedule apps
  - Determine efficient mapping of tasks to processors

All of the above is achieved without performing DSE!

The Daedalus^{RT} Design Flow

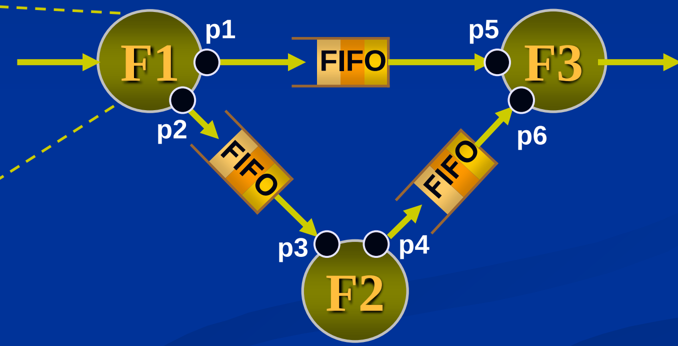


- New features compared to the initial Daedalus
  - Multi-Application Support
  - DSE replaced by
    - Analysis Model Derivation
    - Hard-Real-Time Analysis
  - Two MoCs used – PPN and CSDF! Why?

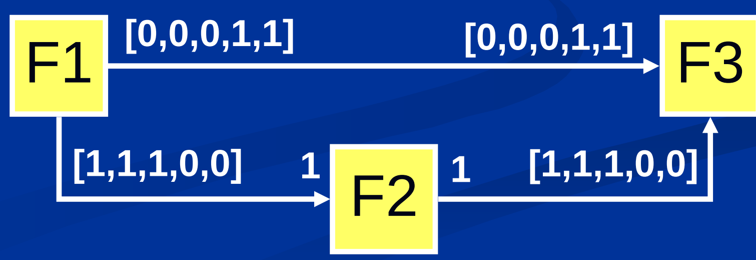
Key Ingredients – the MoCs

- *Polyhedral Process Networks (PPN)*
  - Used for code generation and optimization
  - Optimizations based on solving Integer Linear Programing Problems

```
int M = 5, P = 3;
for( j=1; j <= M; j++)
  for( i=1; i <= M; i++) {
    out = F1( j, i );
    if( i < = P)
      write( p2, out );
    else
      write( p1, out );
  }
```

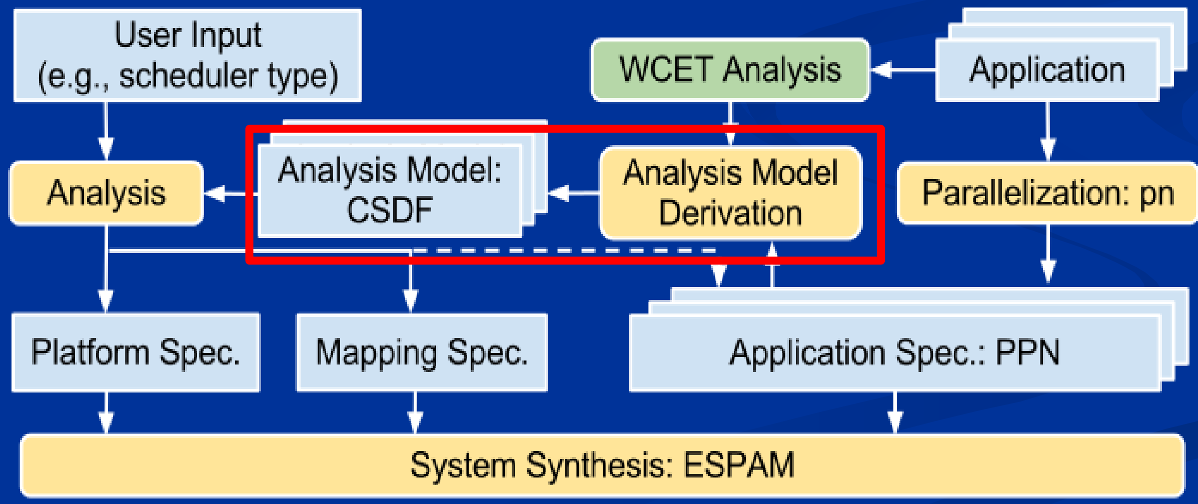


- *Cyclo-Static Dataflow (CSDF)*
  - Used for temporal analysis / performance-constrained scheduling



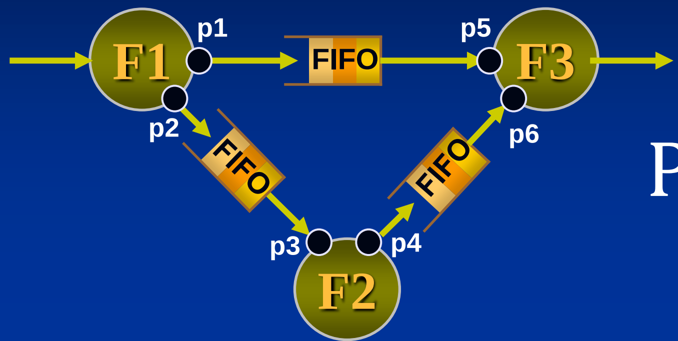
CSDF Model Derivation

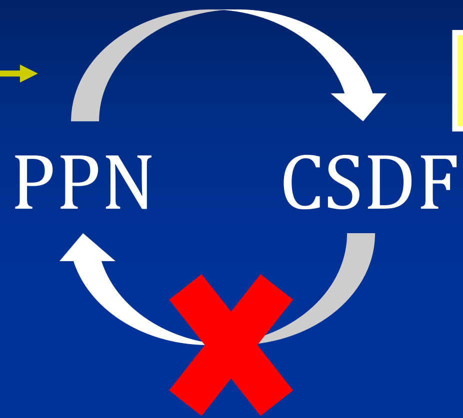
- **Input:**
  - A set of PPNs
  - Worst-case execution time (WCET) information
- **Output:**
  - A set of CSDFs annotated with WCET for each task

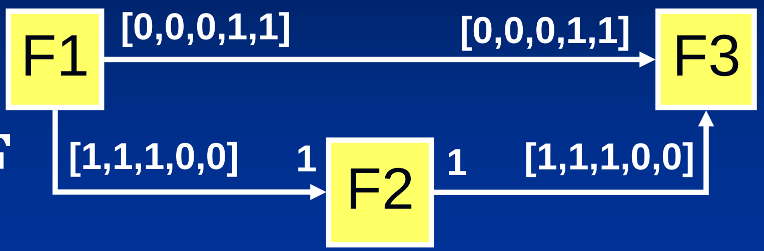


CSDF Model Derivation

- Any PPN has an equivalent CSDF graph



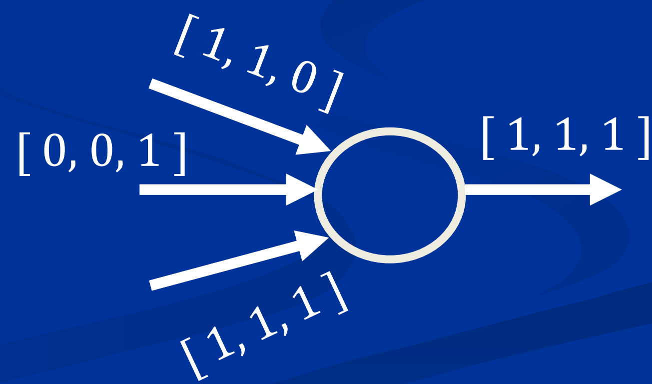




- ***How to derive production/consumption patterns?***

PPN Process

```
while(1){
 for(i=0;i<=9;i++){
  for(j=0;j<=2;j++){
   if(j<=1)
    READ(IP1,&in1);
   if(j==2)
    READ(IP2,&in1);
   READ(IP3,&in2);
   F(in1,in2,out);
   WRITE(OP3,out);
}}}
```



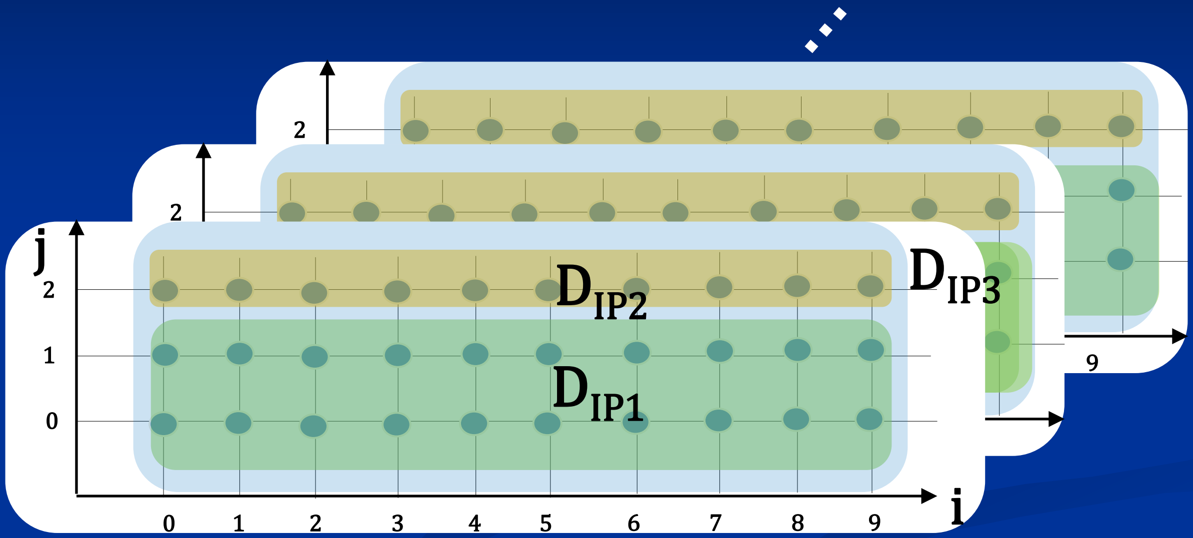
CSDF Actor

Step1: Variant Domain Extraction

- *Variant domain* is a set of process iterations where:
  - *Process accesses the same set of input/output ports*

PPN Process

```
while(1){
 for(i=0;i<=9;i++){
  for(j=0;j<=2;j++){
   if(j<=1)
    READ(IP1,&in1);
   if(j==2)
    READ(IP2,&in1);
   READ(IP3,&in2);
   F(in1,in2,out);
   WRITE(OP3,out);
}}}
```

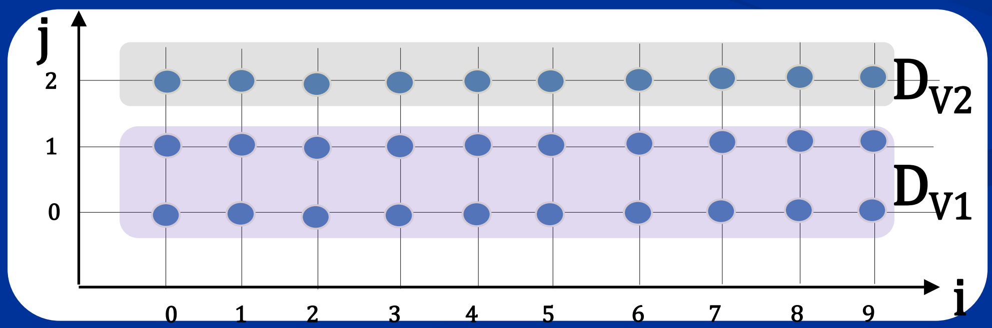


Intersection of Polyhedrons

Variant Domains:

V2 = { IP2, IP3, OP3}

V1 = { IP1, IP3, OP3}



Step2: Variant Domain Traversal

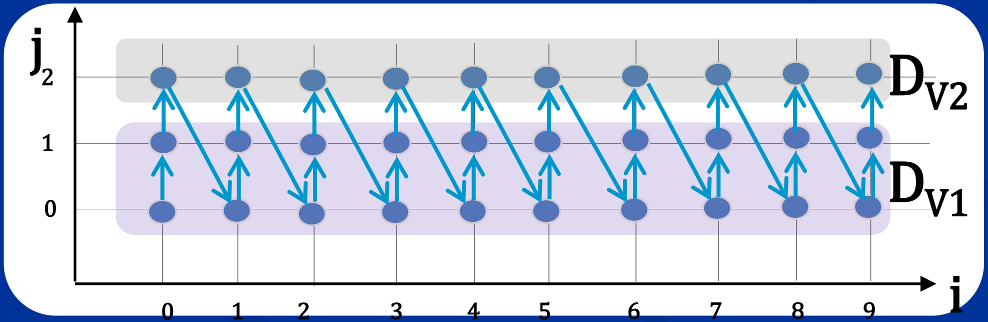
- Traverse variant domains according to loops order to
  - *Express behavior of a process as sequence of variants*

```
while(1){
 for(i=0;i<=9;i++){
  for(j=0;j<=2;j++){
   if(j<=1)
    READ(IP1,&in1);
   if(j==2)
    READ(IP2,&in1);
   READ(IP3,&in2);
   F(in1,in2,out);
   WRITE(OP3,out);
}}}
```

Variant Domains: $V2 = \{ IP2, IP3, OP3\}$

$V1 = \{ IP1, IP3, OP3\}$

Variant Domains Traversal:



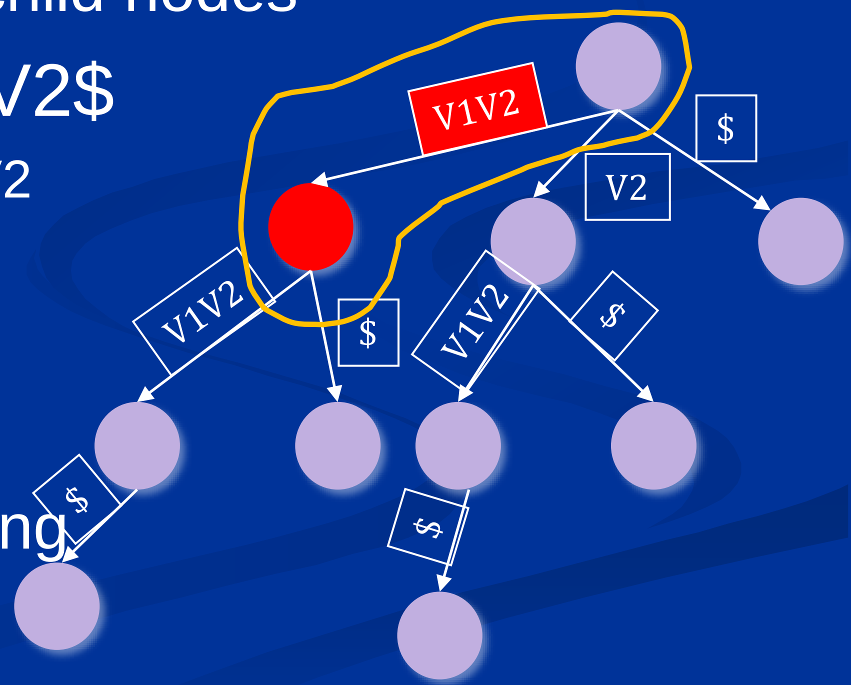
- The traversal results in a sequence (string) *S*

$S = V1V1V2V1V1V2V1V1V2V1V1V2V1V1V2V1V1V2V1V1V2V1V1V2V1V1V2V1V1V2$

- *S* can be very long!
  - At the same time, it might consist of a repeating sub-string

Step3: Find Repeating Sub-string

- Use Suffix Tree representing string S
  - Path from root to any internal node represents repetitive sub-string
  - For any internal node:
    #occurrence of sub-string = #child nodes
- Example: S = V1V2V1V2V1V2$
  - ROOT-to-RED node represents V1V2
  - # Children of RED node = 3
  - V1V2 occurs 3 times in S
- Search Suffix Tree
  - For shortest repeating sub-string
  - Covering entire string S



Step4: Production/Consumption Rates Generation

S = V1V1V2V1V1V2V1V1V2V1V1V2V1V1V2V1V1V2V1V1V2V1V1V2V1V1V2V1V1V2

Repeating Pattern in *S* is: *V1V1V2*

Variant Domains are:

V1 = { IP1, IP3, OP3}

V2 = { IP2, IP3, OP3}

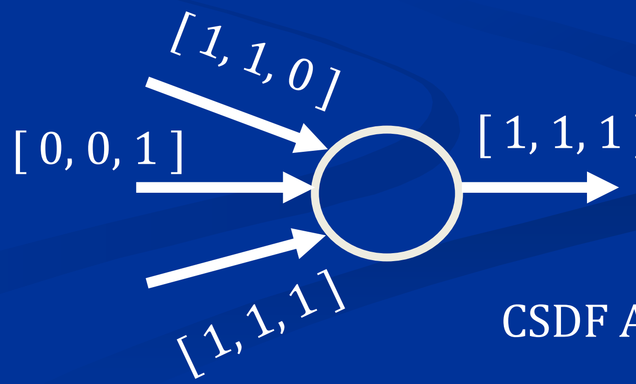
Build a Table

	IP1	IP2	IP3	OP3
V1	1	0	1	1
V1	1	0	1	1
V2	0	1	1	1

PPN Process

```
while(1){
 for(i=0;i<=9;i++){
  for(j=0;j<=2;j++){
   if(j<=1)
    READ(IP1,&in1);
   if(j==2)
    READ(IP2,&in1);
   READ(IP3,&in2);
   F(in1,in2,out);
   WRITE(OP3,out);
}}}
```

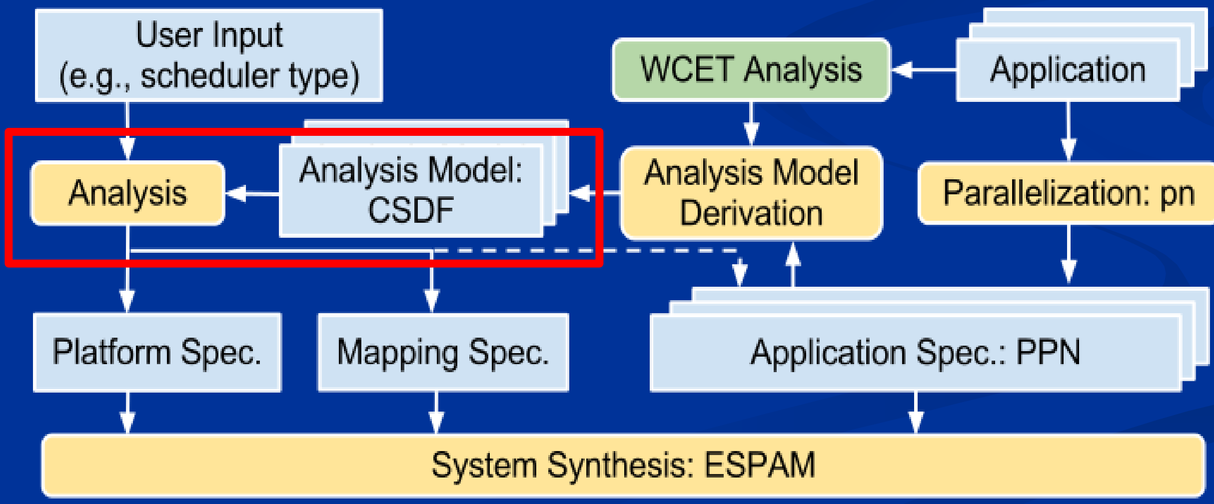
Every column is Rate Pattern!



CSDF Actor

Hard-Real-Time Analysis and Scheduling

- **Input:**
  - A set of CSDFs annotated with WCET for each task
  - User may specify the Hard-Real-Time schedule type to be used
- **Output:**
  - Platform Specifications
  - Mapping Specifications



Exisitng Analysis and Scheduling Approaches

- Self-timed scheduling (STS)
  - Proven to achieve the maximum throughput and minimum latency
  - BUT no temporal isolation! ☹
  - Complex and time consuming DSE needed to find the minimum number of processors! ☹
- Time Division Multiplexing (TDM)
  - Provides temporal isolation ☺
  - BUT complex and time consuming DSE needed to find the minimum number of processors! ☹

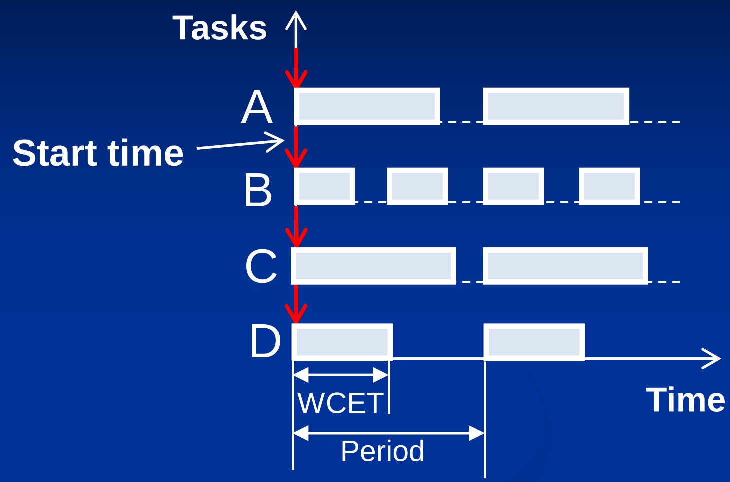
Our Analysis and Scheduling Approaches

- Use hard-real-time multiprocessor scheduling theory
  - Proven timing guarantees
    - enables Hard-Real-Time execution
  - Temporal isolation
    - enables multiple apps + add/remove of apps @ runtime
  - Fast schedulability analysis
    - enables fast admission control + platform sizing

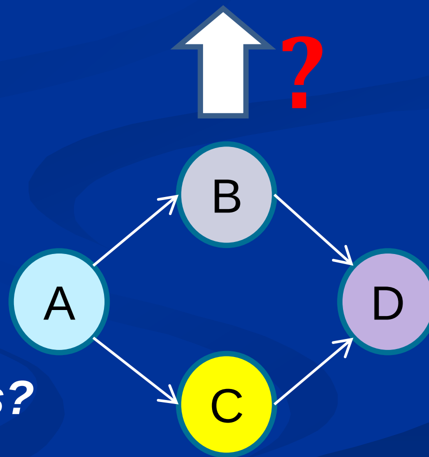
- This theory received little attention in the MPSoC design community! ☹ Why?

The Problem is …

- Most hard-real-time scheduling algorithms assume that:
  - Applications are represented as ***Independent*** periodic or sporadic tasks
  - Each task is characterized by:
    - Start time ***s***
    - Worst-Case Execution Time ***μ***
    - Period ***λ*** (assumed as an implicit deadline)



- In contrast, MPSoC methodologies assume:
  - Applications are represented as tasks/actors with ***Data dependencies*** (in our case CSDF model is used)



- **<u>Problem Statement</u>**
  - ***Can we represent CSDF actors as strictly periodic tasks?***
    - Find a minimum period ***λ*** for each actor
    - Find a start time ***s*** for each actor
    - ***s*** and ***λ*** must satisfy the data dependencies

Our Answer to the Problem is

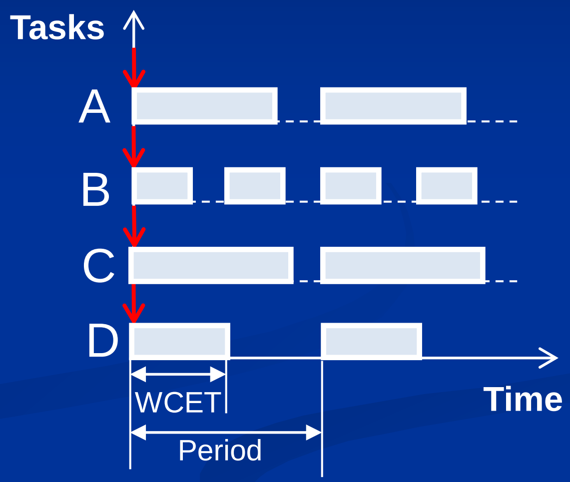
<u>Formally we have proven the following:</u>

- **Actors in any *acyclic* CSDF graph can be scheduled as a set of strictly periodic tasks with**
  - Periods $\boldsymbol{\lambda}$ given by the solution to

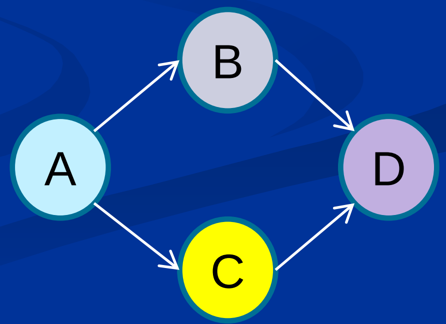
$$q_1 * \lambda_1 = q_2 * \lambda_2 = \ldots = q_N * \lambda_N$$

  - Starting times $\boldsymbol{s}$ proportional to

$$\alpha = q_i * \lambda_i$$



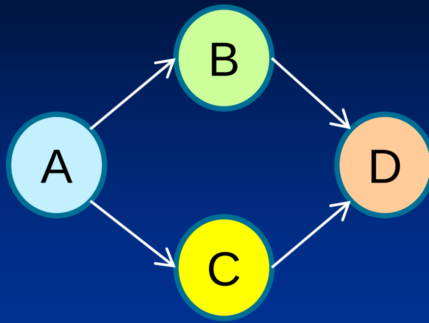
YES



Our *Proof* enables applying classical Hard-Real-Time scheduling theory to embedded streaming applications modeled as acyclic CSDF graphs! ☺

Finding Task Periods

- Example of *acyclic* CSDF graph with
  - Four actors { A, B, C, D }
  - Repetition vector $q = [\ q_A, q_B, q_C, q_D\ ] = [2, 2, 4, 2]$
  - WCET vector $\mu = [\ \mu_A, \mu_B, \mu_C, \mu_D\ ] = [2, 4, 1, 3]$



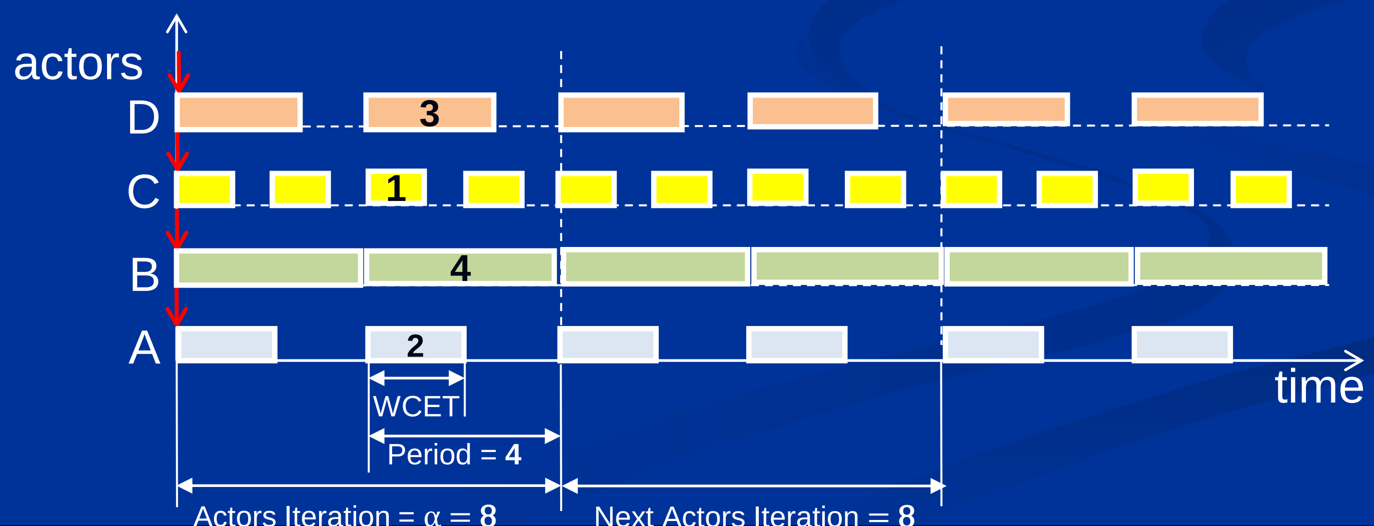
- **Equalize time needed to complete actor iteration** for all actors in order to find the minimum periods of actors:

$2^* \lambda_A = 2^* \lambda_B = 4^* \lambda_C = 2^* \lambda_D$

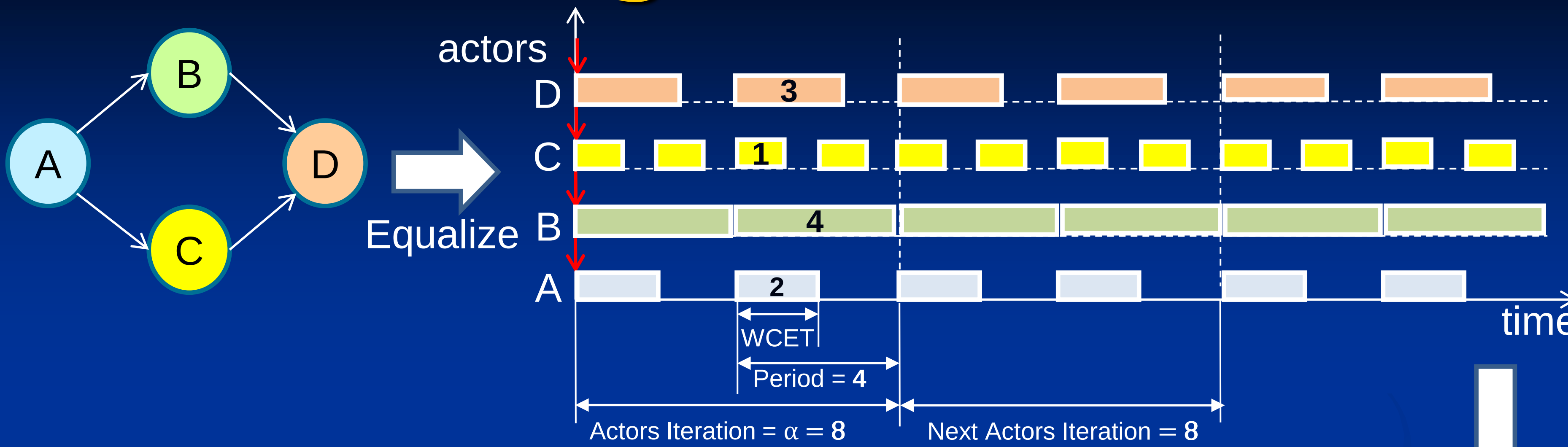
and

$\lambda_A \geq 2, \lambda_B \geq 4, \lambda_C \geq 1, \lambda_D \geq 3,$

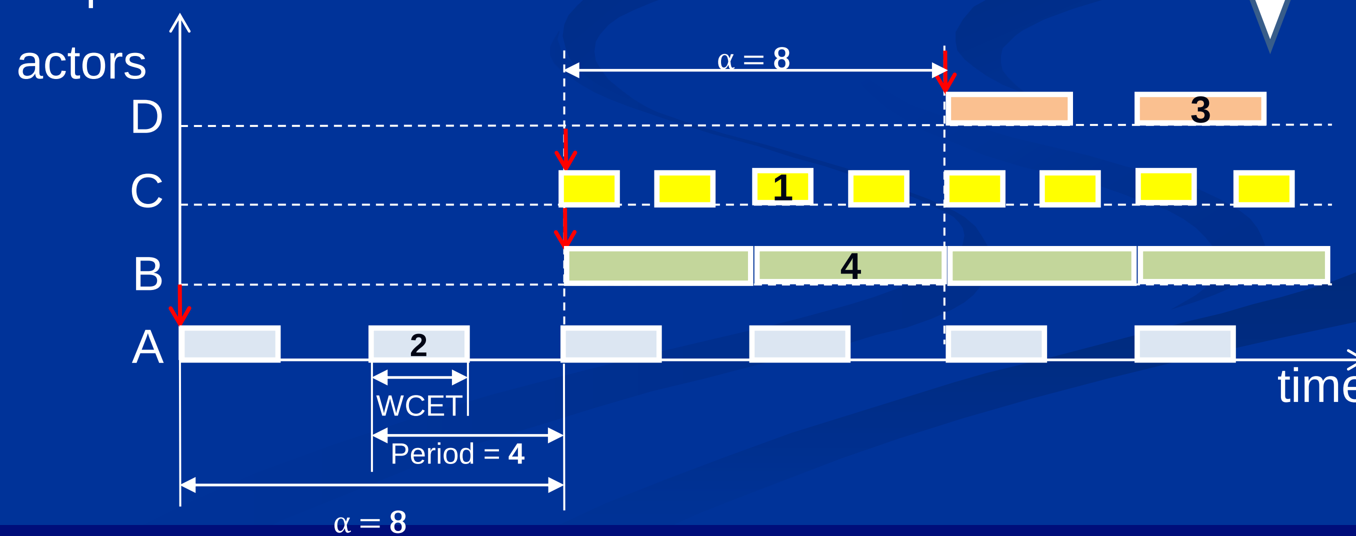
$\Rightarrow \lambda = [\ \lambda_A, \lambda_B, \lambda_C, \lambda_D\ ] = [4, 4, 2, 4]$



Finding Start Times

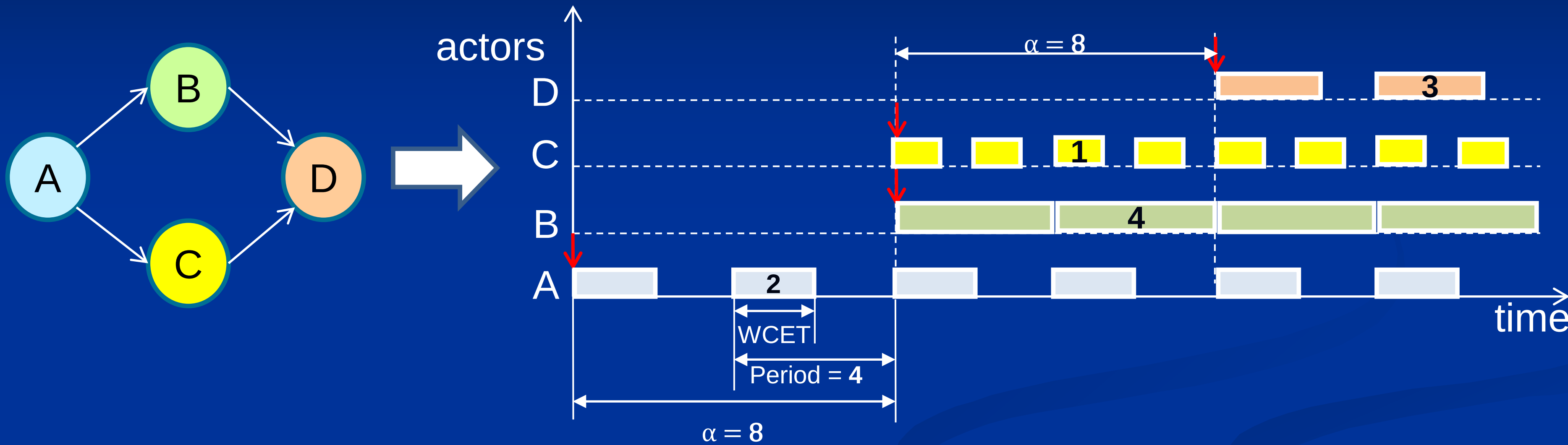


- To ensure correct strictly periodic execution:
  - **Shift stepwise the Start Time of each actor by $\alpha = q_i * \lambda_i = 8$ time-units**
  - Such that the data dependencies are satisfied



Optimizations

A strictly periodic schedule exists! ☺



- However, do we have to shift the Start Times by α?
  - Starting the actors earlier reduces latency and buffer sizes
- Earliest start times and minimum buffer sizes can be found

We have devised proven approaches to determine the minimum values for start times and buffer sizes ☺

Platform Sizing Problem

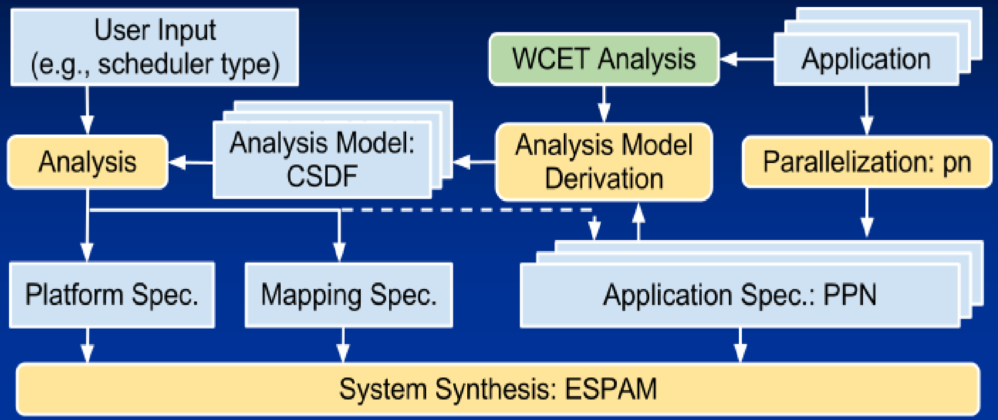
- How many processors ***M*** needed to schedule the actors?
- Computing ***M*** depends on the used HRT schedule

Complex DSE is NOT needed to find *M*! ☺

- Example: CSDF application with 4 actors {A, B, C, D}

	A	B	C	D
WCET_i	5	2	3	2
Period_i	8	8	4	6
$U_i = \frac{\text{WCET}_i}{\text{Period}_i}$	5/8	2/8	3/4	2/6
$U_{sum} = \sum_{\tau_i \in \tau} U_i$	$47/24 = 1.9583$			
M(Optimal)	$\lceil U_{sum} \rceil = \lceil 47/24 \rceil = \mathbf{2}$			
M(P-EDF+FF)	$\min\{x \in \boldsymbol{N}: B \text{ is x-partition of } \tau \text{ and } U_{sum} \leq 1 \forall y \in B\} = \mathbf{3}$			

Results: Flow Execution Times



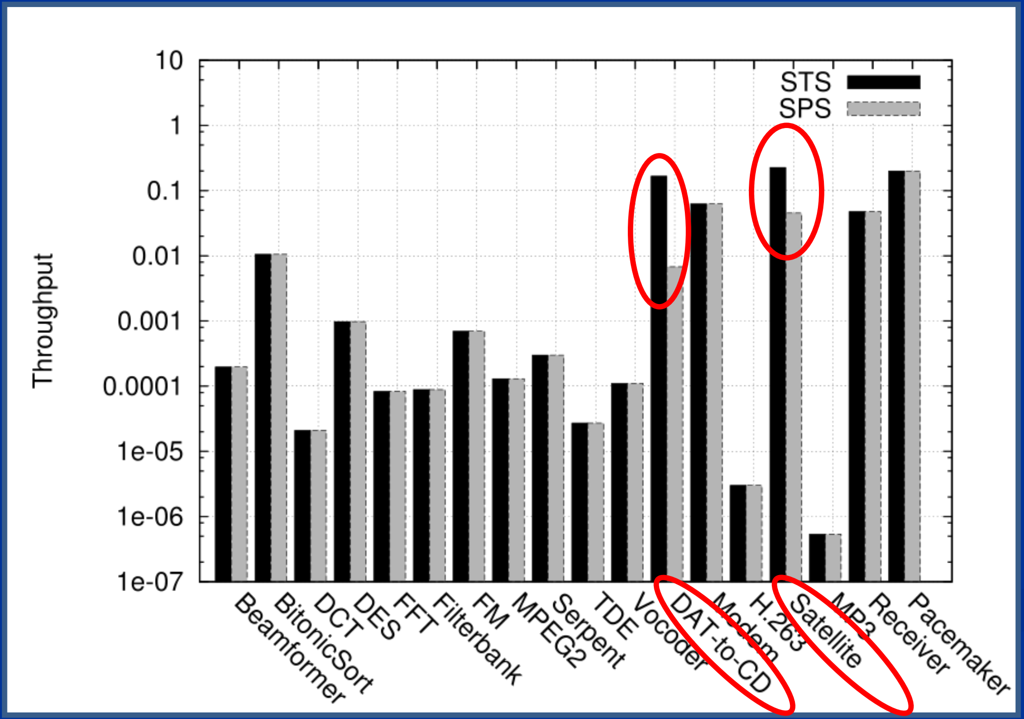
- Multiple Applications:
  - Edge-detection filter (Sobel)
  - Motion JPEG decoder
  - Motion JPEG encoder
- Run simultaneously

	Daedalus^{RT}	
Number of applications	3	
Phase	**Time**	**Automation**
Parallelization	0.48 sec.	Yes
WCET Analysis	1 day	No
Deriving the CSDFs	5 sec.	Yes
Deriving the Platform/Mapping	0.03 sec.	Yes
System Synthesis	2.16 sec.	Yes
Total	~1 day	-
Total excluding WCET	~8 sec.	-

Daedalus^{RT} : significantly reduces design time & effort! ☺

Results: Quality of Schedule

- How Good is our Strictly Periodic Scheduling (SPS)?
  - Use 19 real-life applications
  - Compute throughput using our SPS
  - Compare with maximum achievable throughput using STS



Our SPS achieves maximum throughput for 17 out of 19 apps! ☺