Multiform Logical Time for MeMo-Codesign

Robert de Simone
Inria Sophia-Méditerranée & Université Code d'Azur

INTRODUCTION

Today

14th ACM-IEEE International Conference on Formal Methods and Models for (Embedded) System Design

Until 2013 was Formal Methods and Models for (Hardware-Software) Codesign

a) Hardware Design has traditional Models for Synthesis (up to a point)

b) System Engineering has its methodologies and diagrams

But Software Modeling and Formal Methods do not really match mainstream Software Engineering practices

So *what is* relevant about Software Modeling for Embedded/CPS **system design**?
Or what *could be* ?

OUTLINE

1. System-level, Model-driven, Platform-Based design
2. Around Hardware CAD and Synthesis
3. Around Software Engineering (*MoCs vs MoPs*)
4. A Clock Constraint Specification Language
5. Conclusion and discussion

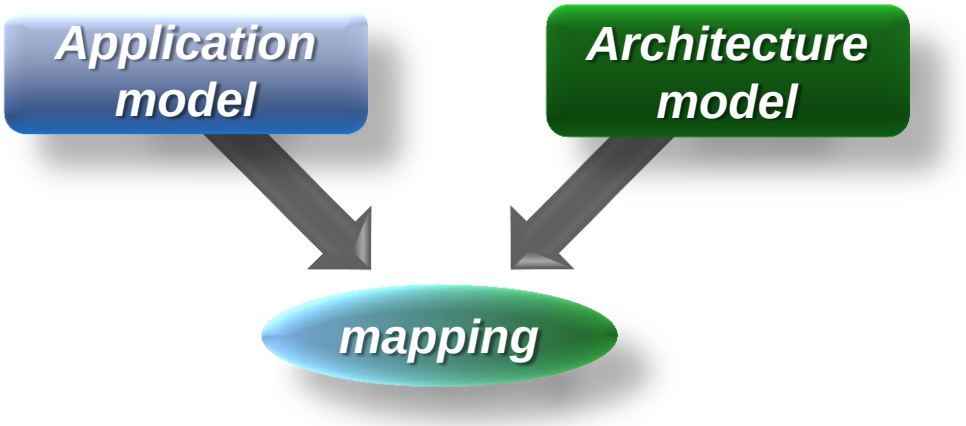
1

System-level, Model-driven, Platform-Based design

Multiviews, Cyber-Physical and Systems of Systems

Y-Chart design approach

Also branded as: Application / Architecture Adaptation (AAA)
where adaptation means optimized mapping,
and mapping covers allocation in space and scheduling in time

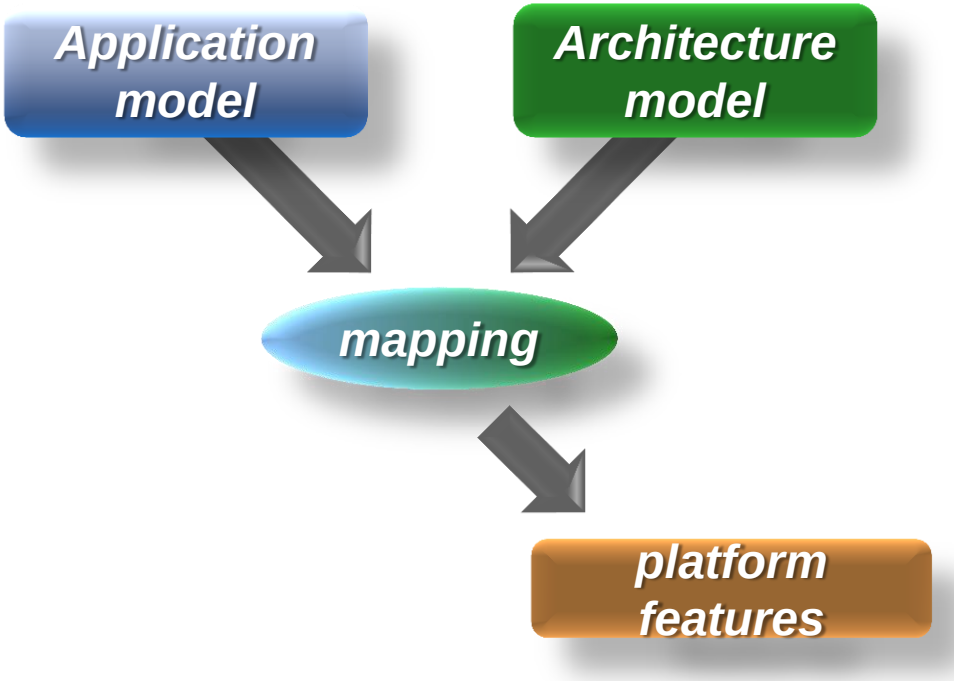


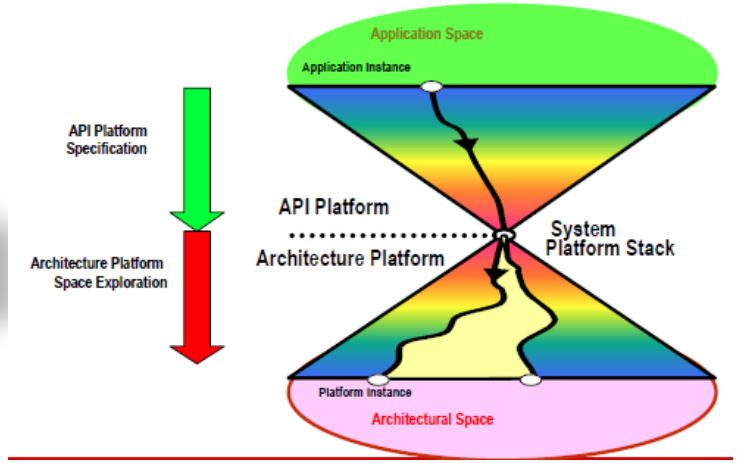
Actually used for design space exploration in distinct engineering purposes, at various modeling levels.

Targeting Hardware: Platform-Based Design

Virtual Platforms and Virtual prototypes assembled by Hardware Architects.

Application model is here *only* to provide typical use cases (booting Linux, one-day battery life) only to exercice the platform and accumulate non-functional information (typically by simulation in SystemC)

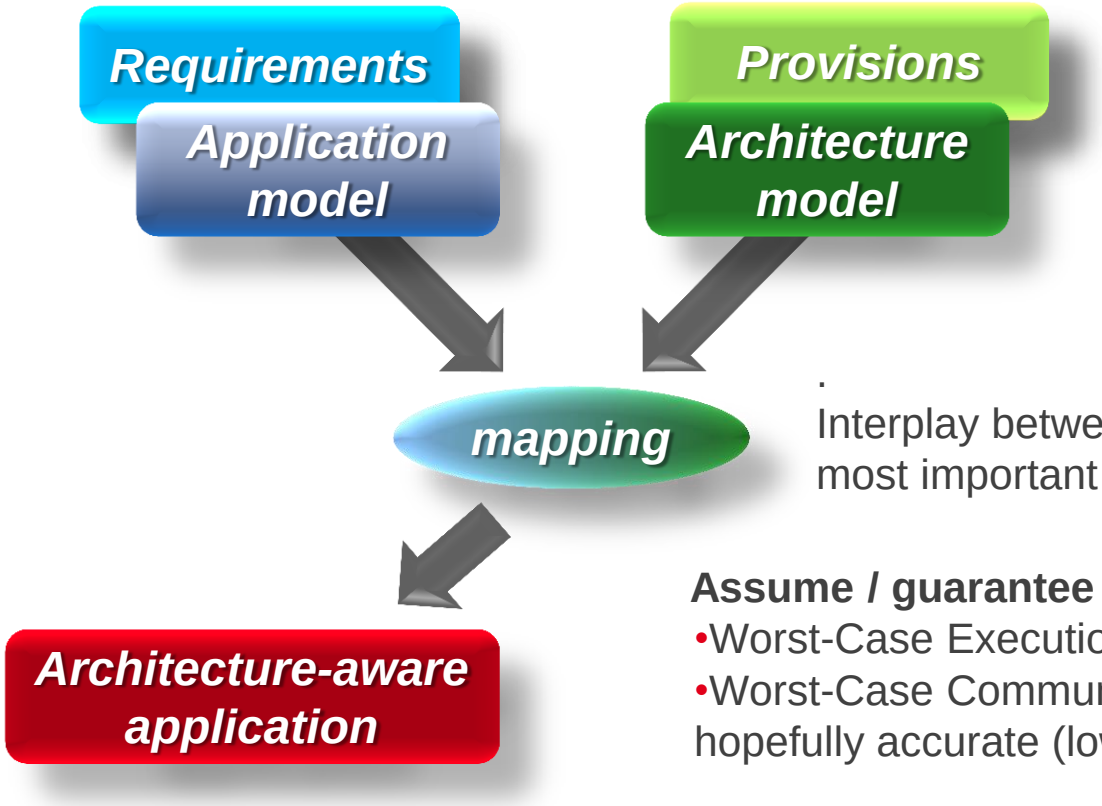




the version of Alberto Sangiovanni-Vincentelli
(UC Berkeley + MARCO Giga-Scale Silicon Research Center

Application / Architecture Adaptation (AAA)

Here mapping corresponds to abstract compilation onto parallel heterogeneous architectures.



.
Interplay between computations and communications most important (overlap and conflicts)

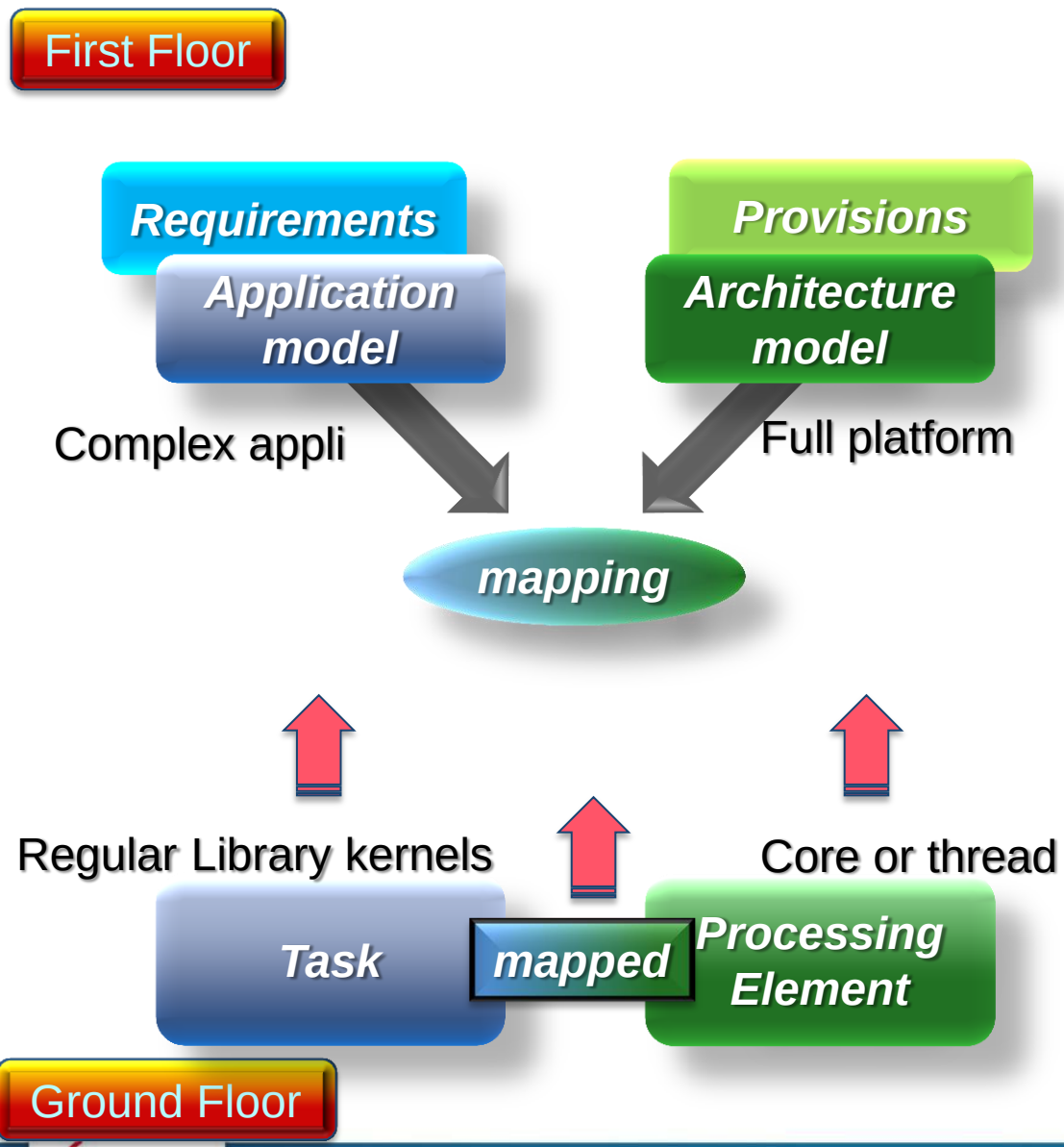
Assume / guarantee approach: one needs
- Worst-Case Execution Time **WCET**
- Worst-Case Communication Time **WCCT**

hopefully accurate (low variability, Precision Timed)

Typically used in **Real-Time Scheduling and Reactive programming**, to get logical timing guarantees

Q: can clever mapping make a non time-predictable platform act more predictably ?

A two-storey approach



The AAA approach cannot span the complete design

Predicable WCET and WCCT computed at a component level **(which?)**

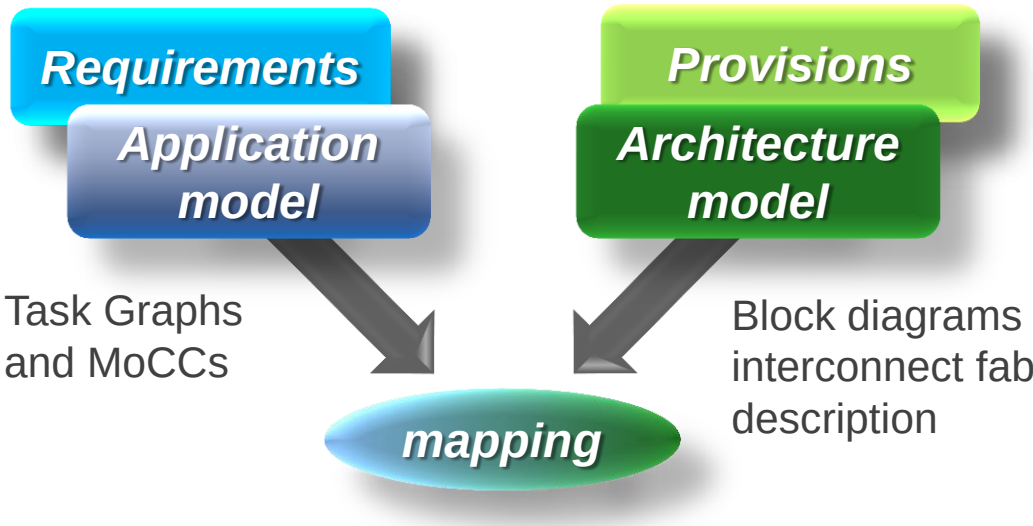
Working at Ground-level more messy (less abstract)
→ needs to be applied on regular subcases only.

Level separation may fluctuate to adjust the approach

Adjust modeling to analysis (and vice-versa)

What we want:

Logical clock constraints + simple formulas between non-functional values



+ accurate synthetized estimation (WCET/WCCT of consistent task kernels as building blocks

Automatic solver or abstract (regular) simulation for optimization

Before we can get this (in a realistic fashion?) we need more feedback
How are similar things done elsewhere ?
Can we borrow, compare, contradict ???

Let's embark on a Tour.

research communities
A ~~hardware/software~~ stack

System Engineering

Formal Methods and Concurrency Theory (MoCCs)

(Domain Specific) Language design

Parallel compilation

Runtime execution and Optimization

Simulation and Worst-Case Execution Time

Hardware Abstraction Layer

Example model-driven platform-based AAA environments

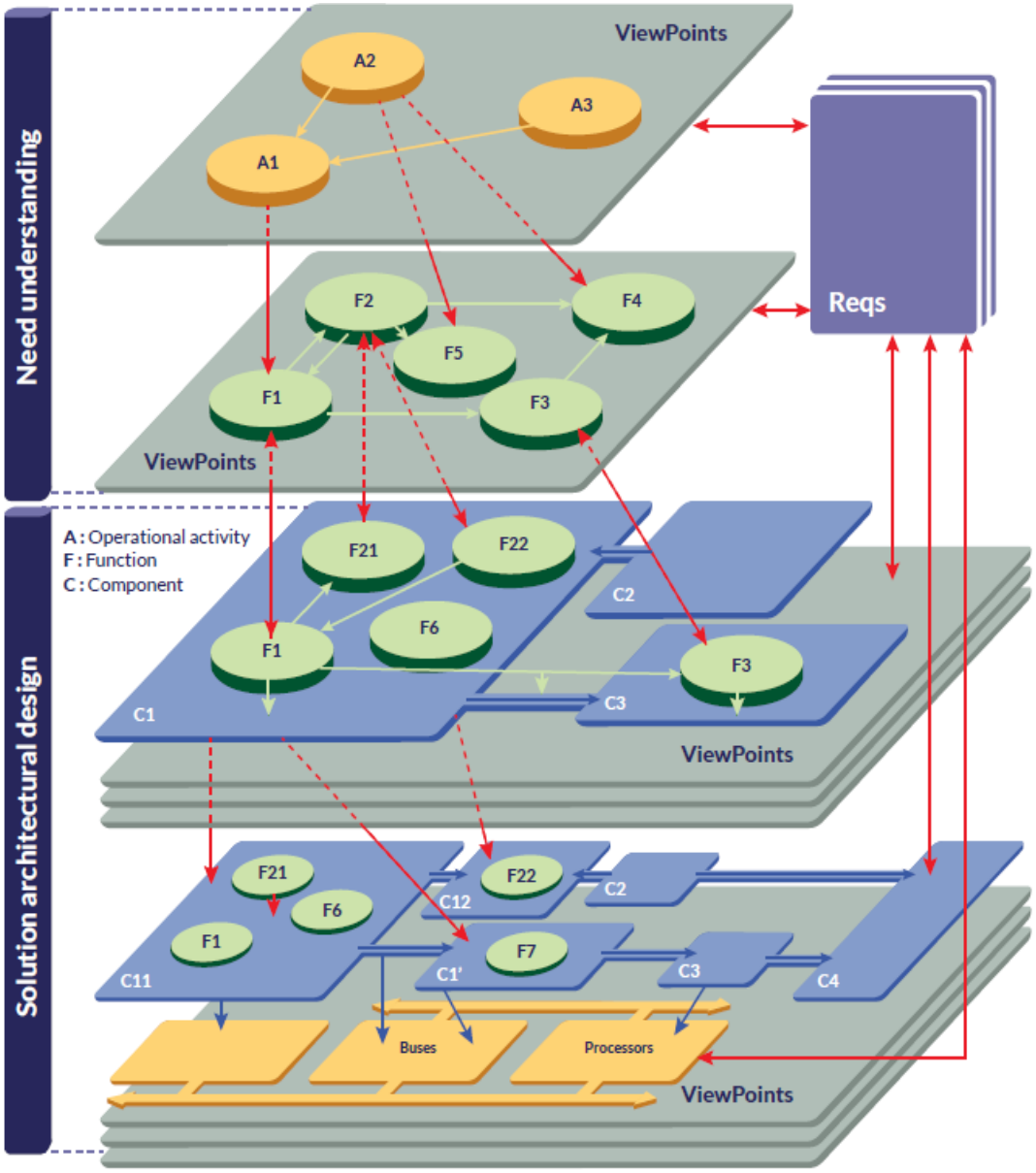
Early system-level design stages (specification)
- SysML/MARTE (AADL)
- ARCADIA / CAPELLA
- Amalthea

Hardware Virtual platforms (MoCs and SoCs)
- Synopysys MCO Platform Architect
- *MetroPolis*
- *StreamIt/Raw (Tilera)*
- *SDF3/Aelite*
- *ForSyDE/Nostrum*
- *SigmaC/Kalray*

Real-Time Scheduling
- AADL (SysML/MARTE)
- SynDEx

Much more often, the combination remains implicit…

System Engineering Design Flow: Arcadia / Capella example

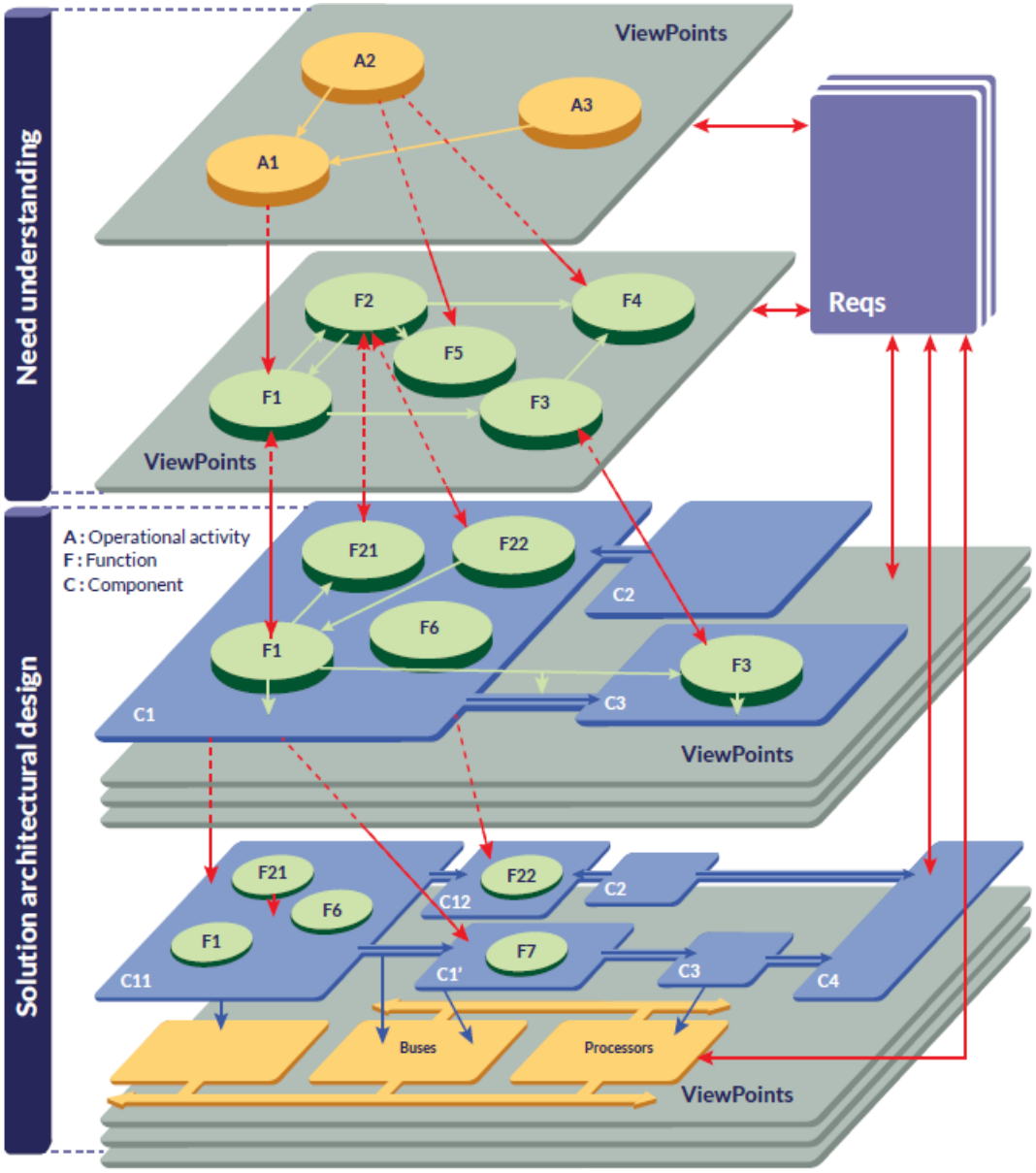


requirement elicitation

functional specification refinement

architectural specification refinement

System Engineering Design Flow: Arcadia / Capella example



Allocation made by user, on software and hardware architectur

Quality of allocation evaluated by computing **simple cost functions**:

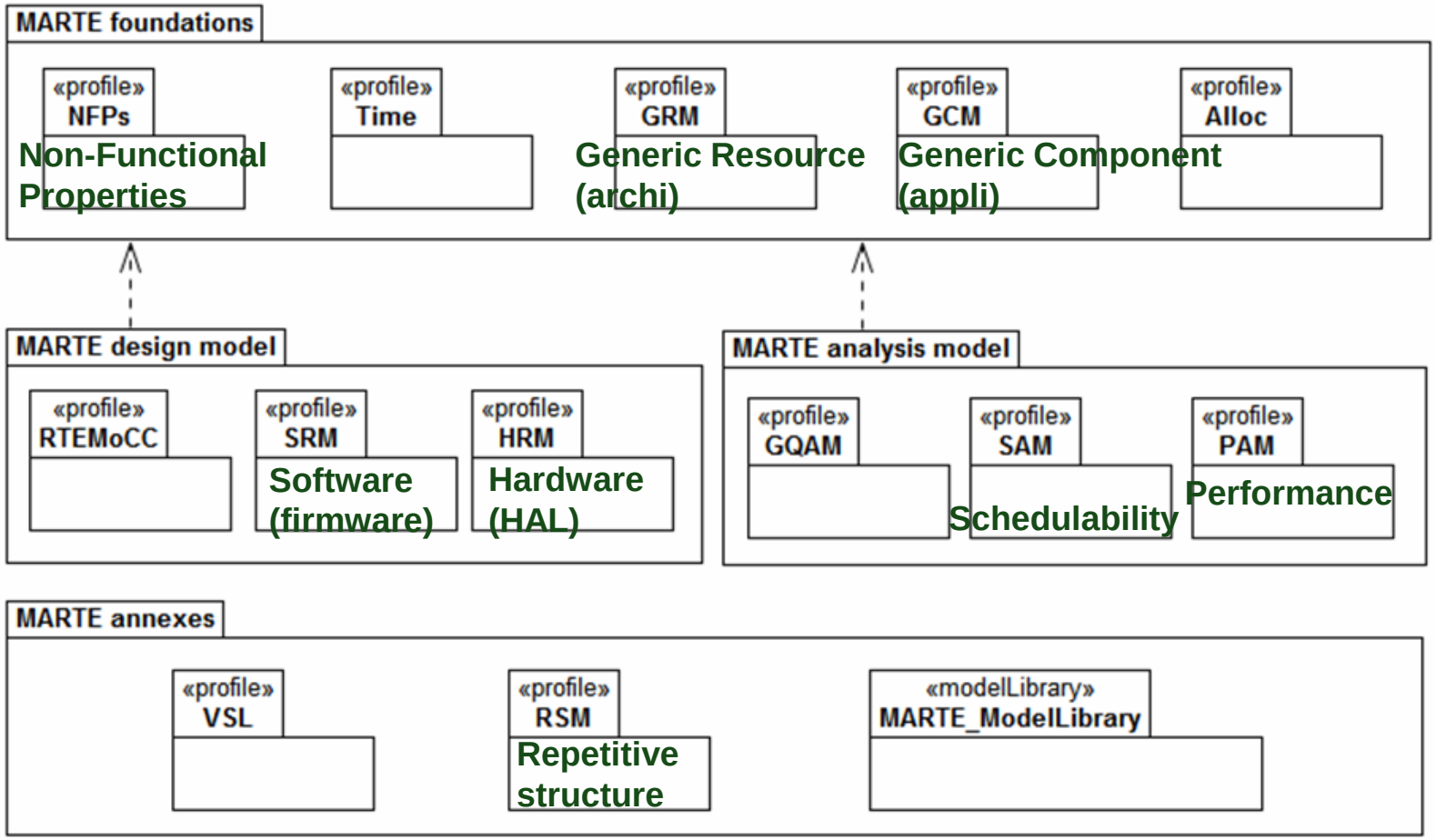
Excel-like spreadsheets mostly

formulas ←→ constraints

Example:
- (money) cost
- mass
- reliability (fault tolerance)
- security (mixed-criticality)

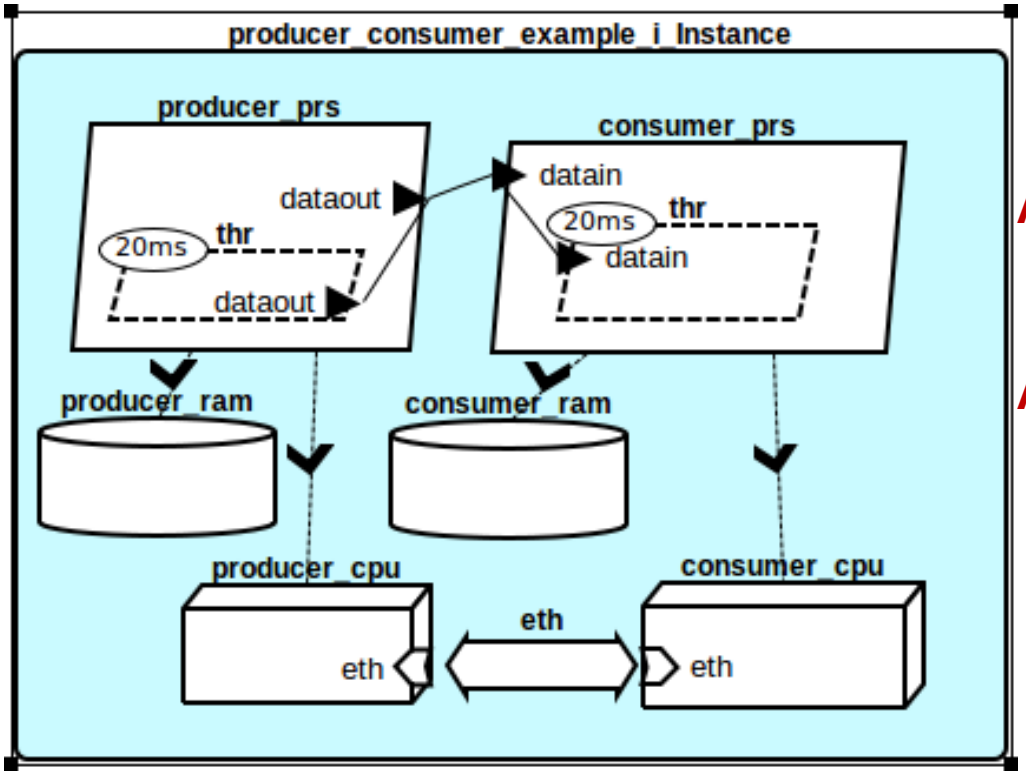
But what if dynamics involved ?
Mode&State changes

SysML/MARTE



SysML parametrics: formulas expressing physical laws (or CPS ones)

AADL



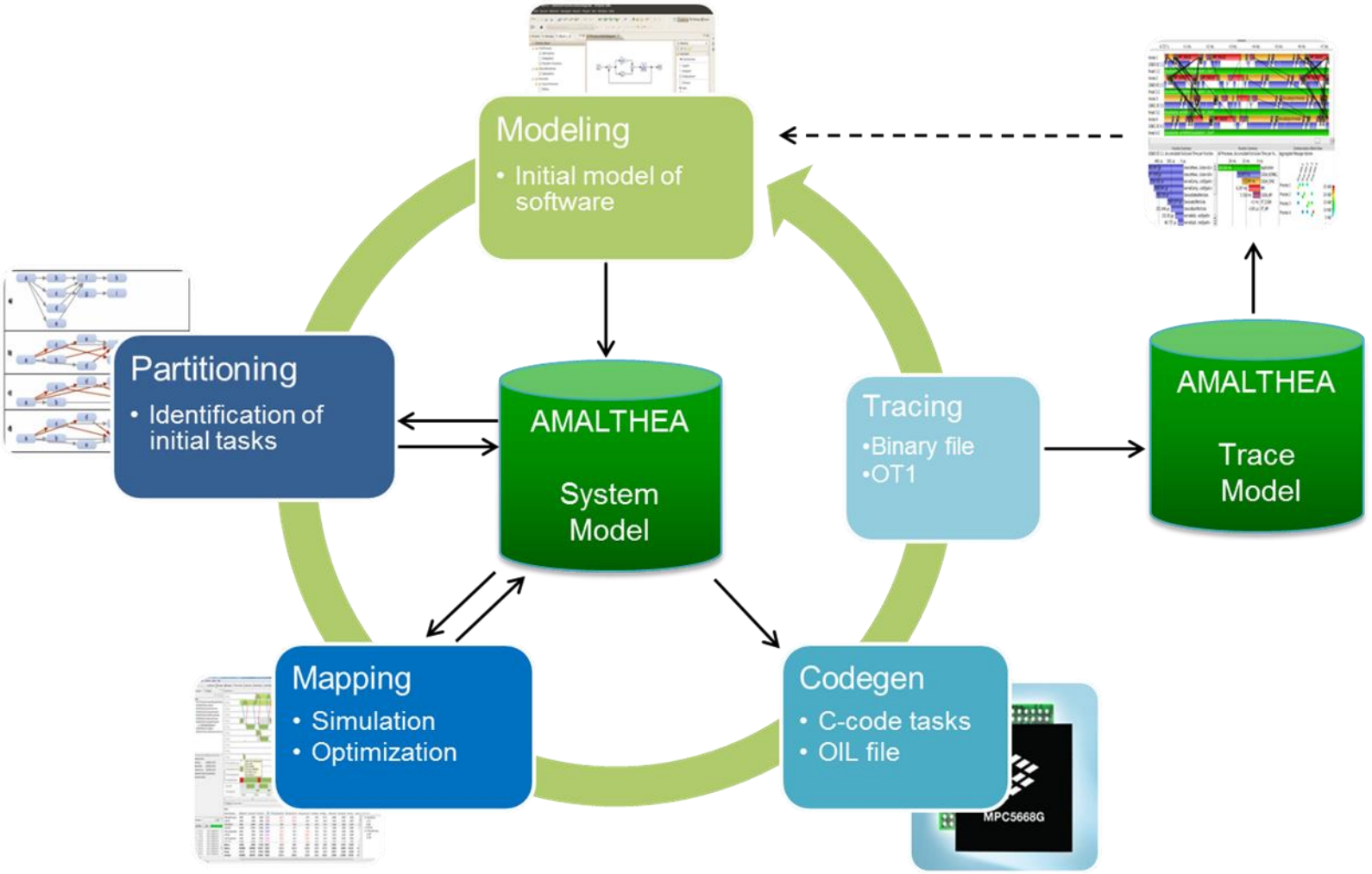
Application

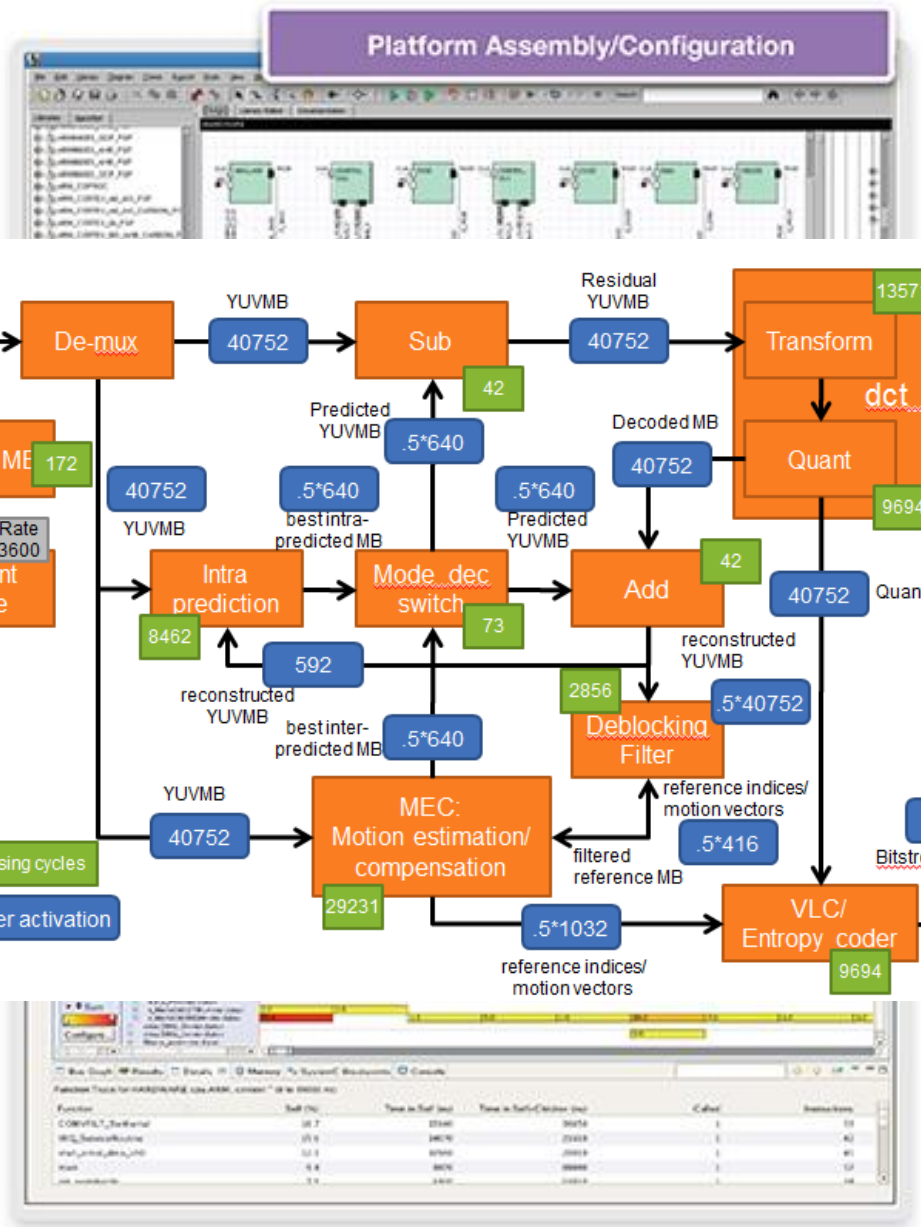
Allocation

Architecture

User provides allocations
Analysis tools compute end-to-end latency and other performance measurements

Example: Amalthea (Bosch et al)





Example:
Synopsys Platform Architect

Many other attempts:

Metropolis
StreamIt/Raw/Tilera
SDF3/Aelite
ForsyDE/Nostrum
SigmaC/Kalray
…

Lessons

Applications and Architectures should be independently described, then (only) fitted together :

- an application may be mapped to multiple execution platforms
- a SW/HW platform is versatile and supports many application

At high-level, simple cost function formulas may be in order, at lower-level a more dynamic simulation relation needed, and if applications have static control the difference is less

Full control over the (closed) system is assumed. In our case:

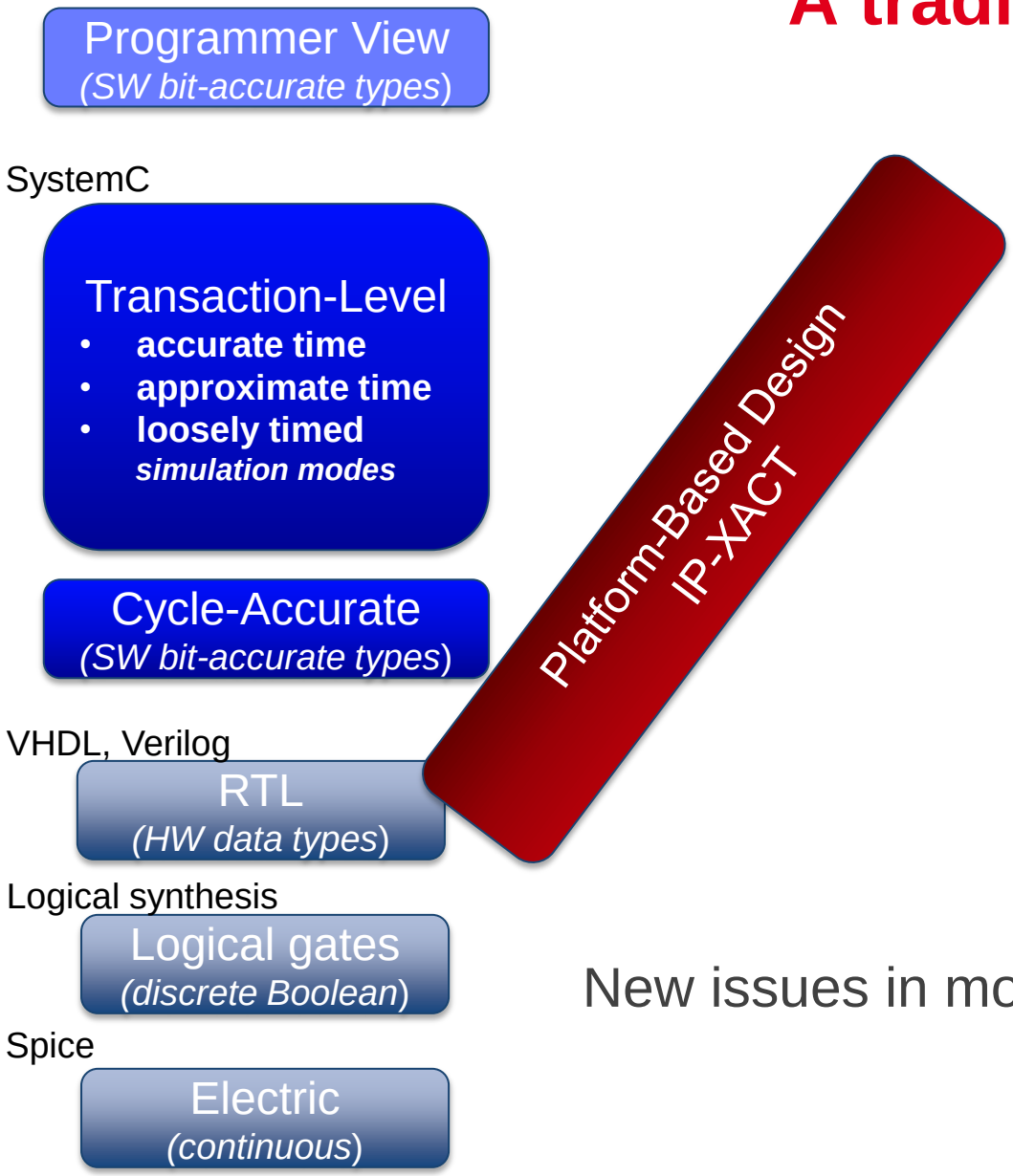
High-Performance or Real-Time Embedded Programming (HPeC)

2

Around Hardware Computer-Aided Design

Sous-titre facultatif

A tradition of models

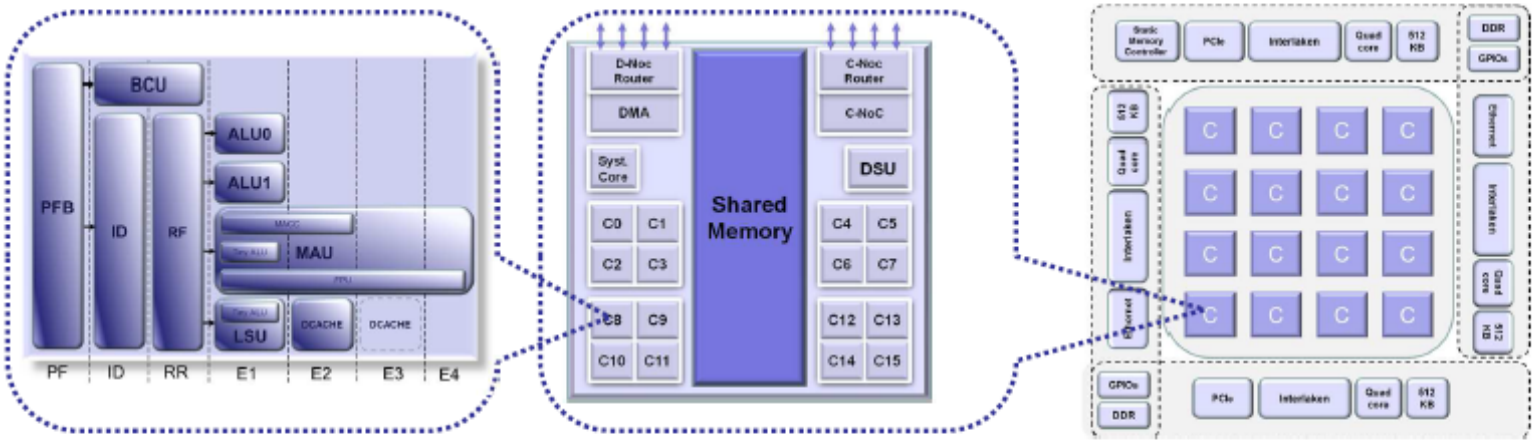


New issues in modeling and simulation

An example many-core time-deterministic processor

KALRAY MPPA®-256 Integrated Manycore Processor

Core | Cluster | Processor



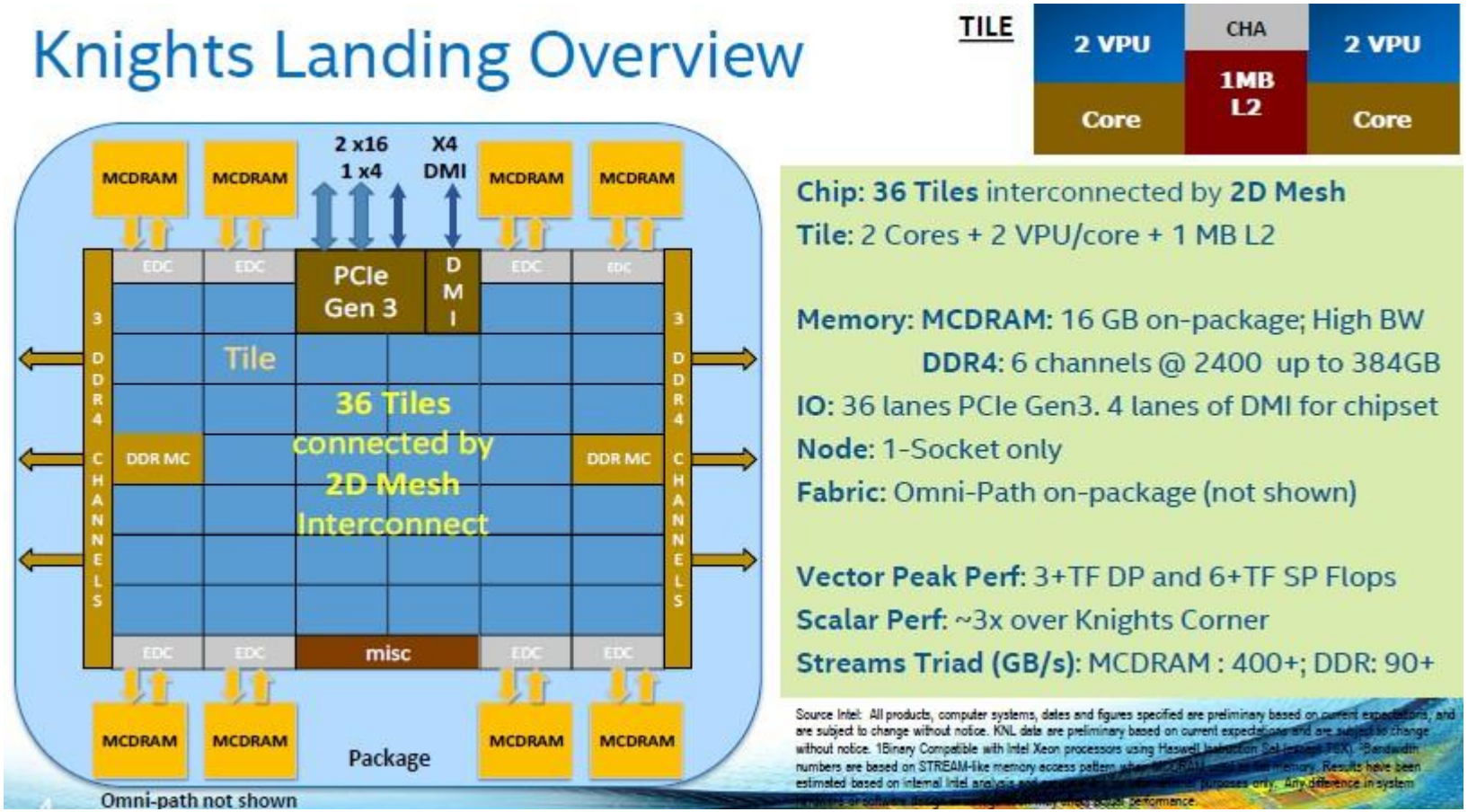
- 5-issue VLIW architecture
- Predictability & energy efficiency
- FPU: 32bits / 64 bits IEEE 754
- MMU for rich OS support

- 16 + 1 cores
- NoC Tx and Rx interfaces
- 2 MB of shared memory

- 2 NoC to connect clusters and I/O
- 47MB of on-chip Memory
- 4x Quad core SMP on each of the four I/O subsystems

- Execution time predictability
  - "Timing Compositional Core", no timing anomalies (Wilhelm)
  - Cluster configuration where each memory bank accessed by 1 core
  - Network calculus applied to the Data NoC to bound transfer latency

A large audience example



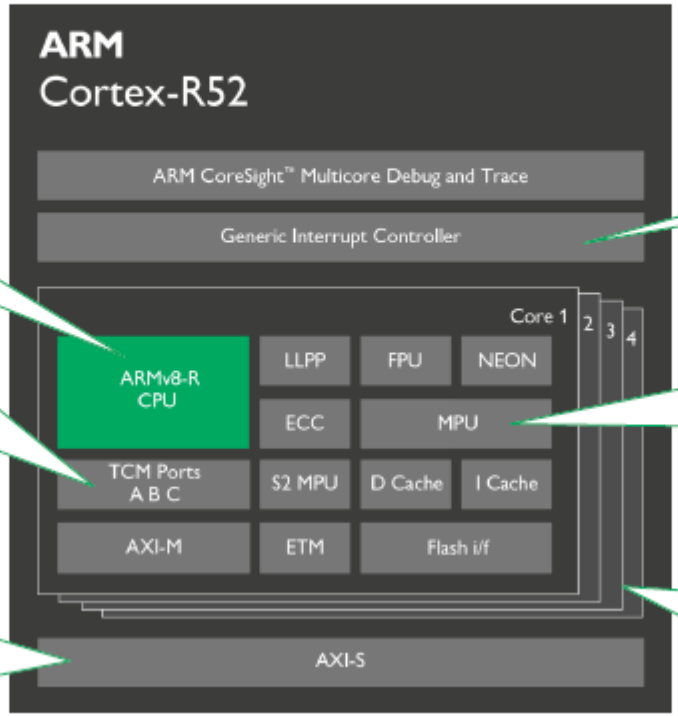
An example Processor for car applications

Built for real-time determinism

Deterministic microarchitecture
- In order execution and retrial
- Superscalar pipeline with extensive dual issuing

Deterministic memory
- Scalable tightly coupled memory for fast access
- Flexible data or instruction allocation

Extensibility
- Rich set of interface ports
- Dedicated low latency ports
- Wide Flash interface port



Fastest interrupt entry
- 2x faster than Cortex-R5
- Interrupt controller integrated within cluster

Rapid responsiveness
- 14x faster context switch than Cortex-R5
- Hard real-time determinism

Scalability
- Option from 1 to 4 cores
- Up to 4 cores in lock step

big. LITTLE alternative mappings

Clustered Switching:

Linux Scheduler picks any **One Cluster** at a time,
But, no combinations

High Cluster is picked if at-least one High Core is needed, else **Low Cluster**

Either

A57 **High Cluster** (High in Perf, Power) of 4 Cores

Cortex -A57 | Cortex -A57 | Cortex -A57 | Cortex -A57

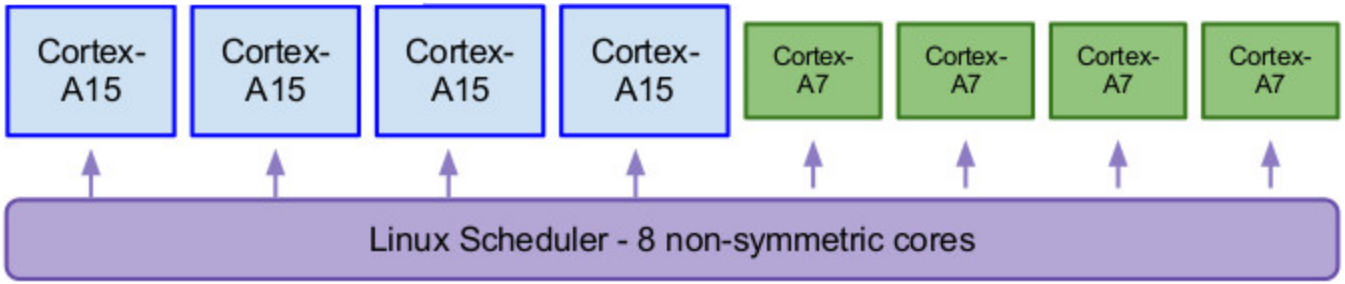
Or

A53 **Low Cluster** (Low in Perf, Power) of 4 Cores

Cortex -A53 | Cortex -A53 | Cortex -A53 | Cortex -A53

$$t.duration = \sum_{r \in R} \sum_{o \in O} \underbrace{((t.opp == o) \times (t.map == r)}_{\text{Boolean decision variables}} \times cost[t][r][o])$$

Global Task Scheduling:



What form of info does one need to organize scheduling ?

(cf. upcoming talk by Emilien Kofman)

Lessons

Big issues in practice are

- **Overlapping communications with computations**
- **Dealing with different (physical) clock speeds**

  Processor cores run faster than bus/NoC, than external memory
- **Dealing with performance and low-power (and temperature)**

  Different clock domains and power domains dark silicon, DVFS…

Ambition is to make the architectural platform model time-predictable

(as much as possible)

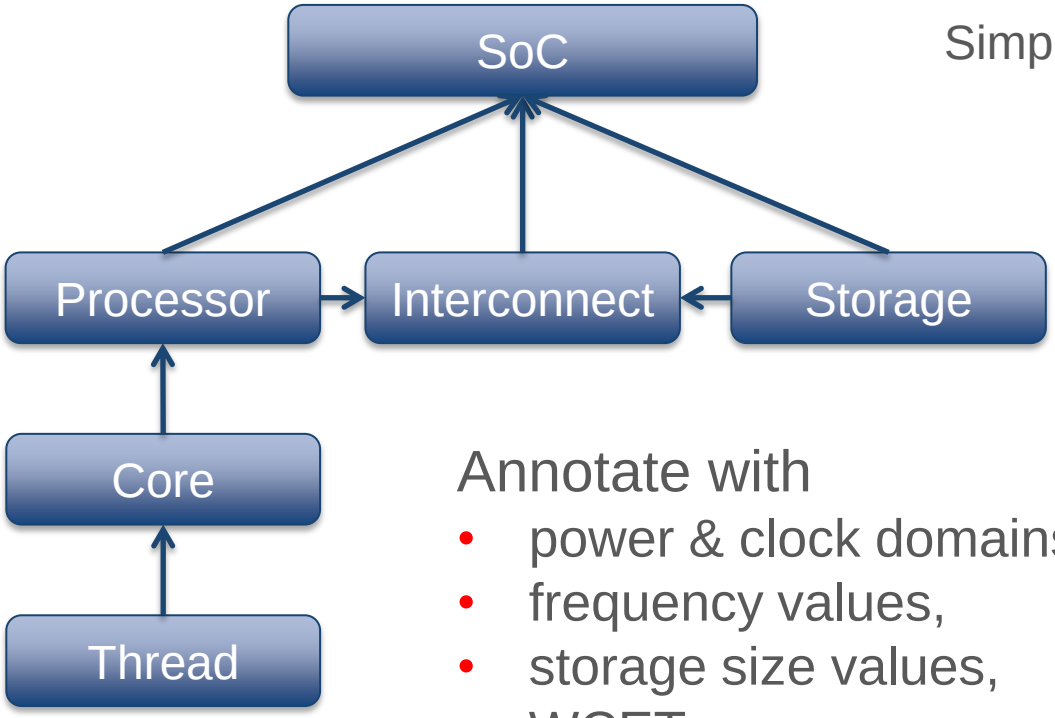
- Either by supposing the real architecture is
- Or by using it in a way that computations and communications fit inside their alloted processing and interconnect resources
  - Avoiding cache misses
  - Avoiding media contention

Requires again static, data-independent control in applications: static planning

Simple relations between dimensions, here again ?

(Hardware Architects use Excel for Timing-Closure)…

Sketch: just enough SoC structure (to support annotation and build constraints)



Simple metamodel ☺ (vs MARTE ☹)

Annotate with

- power & clock domains,
- frequency values,
- storage size values,
- WCET,
- WCCT,
- capacity to overlap computation with communication (DMA and NIC)

3

Around Software Engineering

Programming Models *for parallel processing*

Parallel Programming Models

OpenMP (shared memory)

1. Annotations to instruct the compiler on potential parallelism(s):
2. Parallel for-loops, regions, synchro barriers (ingredients of static oontrol programs)
3. With successive versions, always more annotation types (simd, target device)
4. Also more and more ways to inquire the platform about its dimensions and set up affinities

→ ways to instruct “some” mapping “by hand”

OpenCL (data parallelism

1. Programs split as sets of kernels/tasks, to be applied in a data-parallel fashion
2. Same remark as for OpenMP (4) above

MPI (distributed memory task parallelism and streaming)

1. Networks of (parallel) Processes with message-passing
2. Description of interconnect topology for platform (regular graphs)
3. Placement done by compiler (obscure, may use affinity)

In all cases ,generality (no global static control assumption) is desired.
Some mapping (affinity) and scheduling always done, but without any search for optimality.

No orthogonality (architecture description hinted inside application)

OpenMP Multiprocessing

- Originally, annotation pragmas to indicate which for-loops could be executed in parallel (if distinct iterations data-independent)
- Supposedl equivalent to sequential form, and shared memory (unlike MPI)
- Issue: no real check that pragmas are correct (remember polyhedral model a few foils ago).
- New in v4: Tasks (so that scheduling is really dynamic); goes again the idea of planning before-hand (compilation)
- More and more pragmas (simd, target…) are letting the actual (supposed) architecture crawl into the program description (no orthogonality) Appli should first be archi agnostic, then only made archi aware by compilation

Runtime Library Routines for C/C++

Execution environment routines affect and monitor threads, processors, and the parallel environment. The library routines are external functions with "C" linkage.

Execution Environment Routines

omp_set_num_threads [3.2.1] [3.2.1]
Affects the number of threads used for subsequent parallel regions not specifying a num_threads clause, by setting the value of the first element of the *nthreads-var* ICV of the current task to *num_threads*.

```
void omp_set_num_threads(int num_threads);
```

omp_get_num_threads [3.2.2] [3.2.2]
Returns the number of threads in the current team. The binding region for an omp_get_num_threads region is the innermost enclosing parallel region.

```
int omp_get_num_threads(void);
```

omp_get_max_threads [3.2.3] [3.2.3]
Returns an upper bound on the number of threads that could be used to form a new team if a parallel construct without a num_threads clause were encountered after execution returns from this routine.

```
int omp_get_max_threads(void);
```

omp_get_thread_num [3.2.4] [3.2.4]
Returns the thread number of the calling thread within the current team.

```
int omp_get_thread_num(void);
```

omp_get_num_procs [3.2.5] [3.2.5]
Returns the number of processors that are available to the device at the time the routine is called.

```
int omp_get_num_procs(void);
```

omp_in_parallel [3.2.6] [3.2.6]
Returns *true* if the *active-levels-var* ICV is greater than zero; otherwise it returns *false*.

```
int omp_in_parallel(void);
```

omp_set_dynamic [3.2.7] [3.2.7]
Enables or disables dynamic adjustment of the number of threads available for the execution of subsequent parallel regions by setting the value of the *dyn-var* ICV.

```
void omp_set_dynamic(int dynamic_threads);
```

omp_get_dynamic [3.2.8] [3.2.8]
This routine returns the value of the *dyn-var* ICV, which is *true* if dynamic adjustment of the number of threads is enabled for the current task.

```
int omp_get_dynamic(void);
```

omp_get_cancellation [3.2.9] [3.2.9]
Returns the value of the *cancel-var* ICV, which is *true* if cancellation is activated; otherwise it returns *false*.

```
int omp_get_cancellation(void);
```

omp_set_nested [3.2.10] [3.2.10]
Enables or disables nested parallelism, by setting the *nest-var* ICV.

```
void omp_set_nested(int nested);
```

omp_get_nested [3.2.11] [3.2.10]
Returns the value of the *nest-var* ICV, which indicates if nested parallelism is enabled or disabled.

```
int omp_get_nested(void);
```

omp_set_schedule [3.2.12] [3.2.12]
Affects the schedule that is applied when runtime is used as schedule kind, by setting the value of the *run-sched-var* ICV.

```
void omp_set_schedule(omp_sched_t kind, int chunk_size);
```

kind: One of the following, or an implementation-defined schedule:

omp_sched_static	= 1
omp_sched_dynamic	= 2
omp_sched_guided	= 3
omp_sched_auto	= 4

omp_get_schedule [3.2.13] [3.2.13]
Returns the value of *run-sched-var* ICV, which is the schedule applied when runtime schedule is used.

```
void omp_get_schedule(
  omp_sched_t *kind, int *chunk_size);
```

See *kind* for **omp_set_schedule**.

omp_get_thread_limit [3.2.14] [3.2.14]
Returns the value of the *thread-limit-var* ICV, which is the maximum number of OpenMP threads available.

```
int omp_get_thread_limit(void);
```

omp_set_max_active_levels [3.2.15] [3.2.15]
Limits the number of nested active parallel regions, by setting *max-active-levels-var* ICV.

```
void omp_set_max_active_levels(int max_levels);
```

omp_get_max_active_levels [3.2.16] [3.2.16]
Returns the value of *max-active-levels-var* ICV, which determines the maximum number of nested active parallel regions.

```
int omp_get_max_active_levels(void);
```

omp_get_level [3.2.17] [3.2.17]
For the enclosing device region, returns the *levels-vars* ICV, which is the number of nested parallel regions that enclose the task containing the call.

```
int omp_get_level(void);
```

omp_get_ancestor_thread_num [3.2.18] [3.2.18]
Returns, for a given nested level of the current thread, the thread number of the ancestor of the current thread.

```
int omp_get_ancestor_thread_num(int level);
```

omp_get_team_size [3.2.19] [3.2.19]
Returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs.

```
int omp_get_team_size(int level);
```

omp_get_active_level [3.2.20] [3.2.20]
Returns the value of the *active-level-vars* ICV, which determines the number of active, nested parallel regions enclosing the task that contains the call.

```
int omp_get_active_level(void);
```

omp_in_final [3.2.21] [3.2.21]
Returns *true* if the routine is executed in a final task region; otherwise, it returns *false*.

```
int omp_in_final(void);
```

omp_get_proc_bind [3.2.22] [3.2.22]
Returns the thread affinity policy to be used for the subsequent nested parallel regions that do not specify a proc_bind clause.

```
omp_proc_bind_t omp_get_proc_bind(void);
```

Returns one of:

omp_proc_bind_false	= 0
omp_proc_bind_true	= 1
omp_proc_bind_master	= 2
omp_proc_bind_close	= 3
omp_proc_bind_spread	= 4

omp_get_ num_places [3.2.23]
Returns the number of places available to the execution environment in the place list.

```
int omp_get_num_places(void);
```

omp_get_place_num_procs [3.2.24]
Returns the number of processors available to the execution environment in the specified place.

```
int omp_get_place_num_procs(int place_num);
```

omp_get_place_proc_ids [3.2.25]
Returns the numerical identifiers of the processors available to the execution environment in the specified place.

```
void omp_get_place_proc_ids(
  int place_num, int *ids);
```

omp_get_place_num [3.2.26]
Returns the place number of the place to which the encountering thread is bound.

```
int omp_get_place_num(void);
```

omp_get_partition_num_places [3.2.27]
Returns the number of places in the place partition of the innermost implicit task.

```
int omp_get_partition_num_places(void);
```

omp_get_partition_place_nums [3.2.28]
Returns the list of place numbers corresponding to the places in the *place-partition-var* ICV of the innermost implicit task.

```
void omp_get_partition_place_nums(int *place_nums);
```

Continued

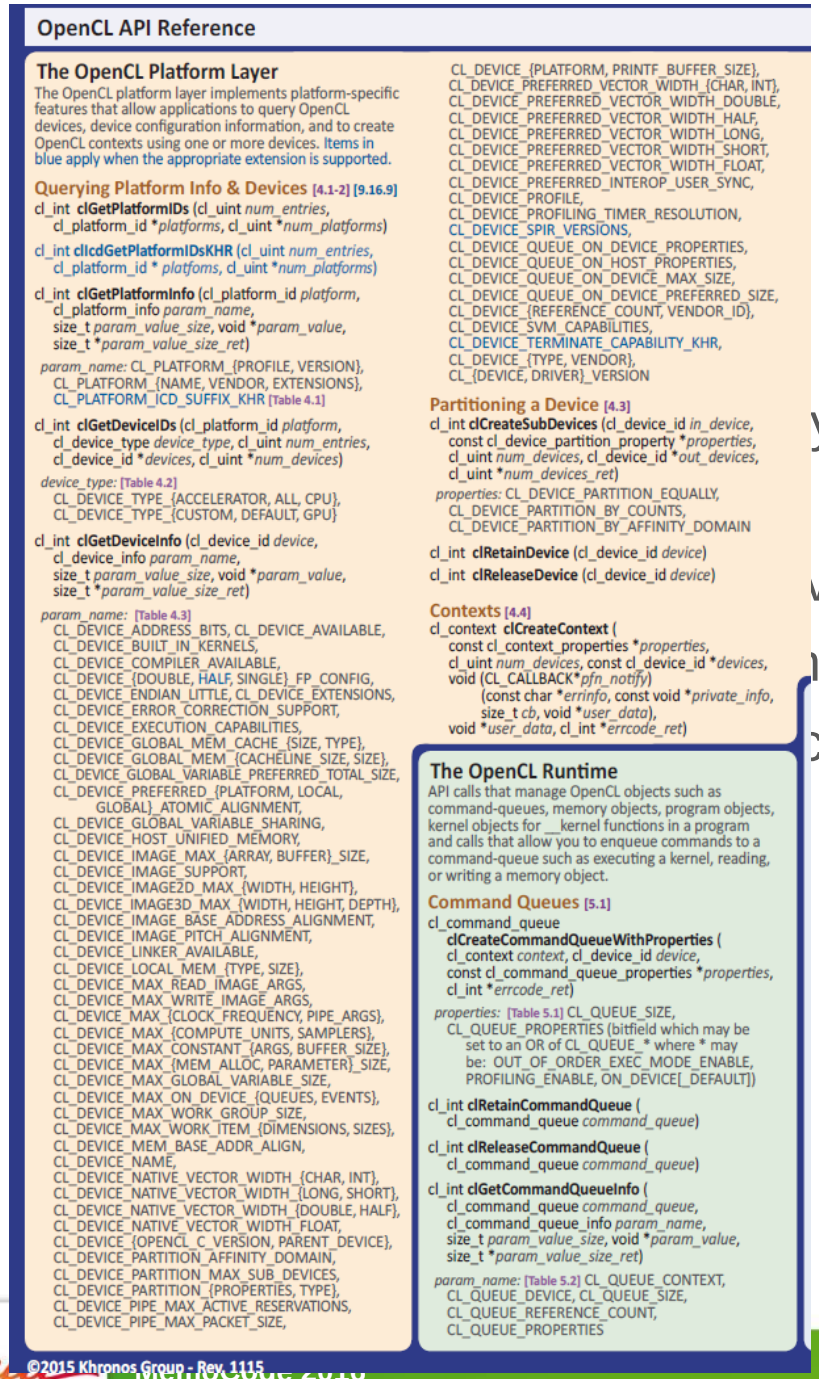
OpenCL

y and explicit data transfer for GPU

verfeed the processing elements, and

ave arrived (finger-crossed)

computation and communication



OpenCL API Reference

The OpenCL Platform Layer

The OpenCL platform layer implements platform-specific features that allow applications to query OpenCL devices, device configuration information, and to create OpenCL contexts using one or more devices. Items in blue apply when the appropriate extension is supported.

Querying Platform Info & Devices [4.1-2] [9.16.9]

cl_int **clGetPlatformIDs** (cl_uint *num_entries*, cl_platform_id **platforms*, cl_uint **num_platforms*)

cl_int **clIcdGetPlatformIDsKHR** (cl_uint *num_entries*, cl_platform_id * *platforms*, cl_uint **num_platforms*)

cl_int **clGetPlatformInfo** (cl_platform_id *platform*, cl_platform_info *param_name*, size_t *param_value_size*, void **param_value*, size_t **param_value_size_ret*)

param_name: CL_PLATFORM_{PROFILE, VERSION}, CL_PLATFORM_{NAME, VENDOR, EXTENSIONS}, CL_PLATFORM_ICD_SUFFIX_KHR [Table 4.1]

cl_int **clGetDeviceIDs** (cl_platform_id *platform*, cl_device_type *device_type*, cl_uint *num_entries*, cl_device_id **devices*, cl_uint **num_devices*)

device_type: [Table 4.2]
CL_DEVICE_TYPE_{ACCELERATOR, ALL, CPU}, CL_DEVICE_TYPE_{CUSTOM, DEFAULT, GPU}

cl_int **clGetDeviceInfo** (cl_device_id *device*, cl_device_info *param_name*, size_t *param_value_size*, void **param_value*, size_t **param_value_size_ret*)

param_name: [Table 4.3]
CL_DEVICE_ADDRESS_BITS, CL_DEVICE_AVAILABLE, CL_DEVICE_BUILT_IN_KERNELS, CL_DEVICE_COMPILER_AVAILABLE, CL_DEVICE_{DOUBLE, HALF, SINGLE}_FP_CONFIG, CL_DEVICE_ENDIAN_LITTLE, CL_DEVICE_EXTENSIONS, CL_DEVICE_ERROR_CORRECTION_SUPPORT, CL_DEVICE_EXECUTION_CAPABILITIES, CL_DEVICE_GLOBAL_MEM_CACHE_{SIZE, TYPE}, CL_DEVICE_GLOBAL_MEM_{CACHELINE_SIZE, SIZE}, CL_DEVICE_GLOBAL_VARIABLE_PREFERRED_TOTAL_SIZE, CL_DEVICE_PREFERRED_{PLATFORM, LOCAL, GLOBAL}_ATOMIC_ALIGNMENT, CL_DEVICE_GLOBAL_VARIABLE_SHARING, CL_DEVICE_HOST_UNIFIED_MEMORY, CL_DEVICE_IMAGE_MAX_{ARRAY, BUFFER}_SIZE, CL_DEVICE_IMAGE_SUPPORT, CL_DEVICE_IMAGE2D_MAX_{WIDTH, HEIGHT}, CL_DEVICE_IMAGE3D_MAX_{WIDTH, HEIGHT, DEPTH}, CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT, CL_DEVICE_IMAGE_PITCH_ALIGNMENT, CL_DEVICE_LINKER_AVAILABLE, CL_DEVICE_LOCAL_MEM_{TYPE, SIZE}, CL_DEVICE_MAX_READ_IMAGE_ARGS, CL_DEVICE_MAX_WRITE_IMAGE_ARGS, CL_DEVICE_MAX_{CLOCK_FREQUENCY, PIPE_ARGS}, CL_DEVICE_MAX_{COMPUTE_UNITS, SAMPLERS}, CL_DEVICE_MAX_CONSTANT_{ARGS, BUFFER_SIZE}, CL_DEVICE_MAX_{MEM_ALLOC, PARAMETER}_SIZE, CL_DEVICE_MAX_GLOBAL_VARIABLE_SIZE, CL_DEVICE_MAX_ON_DEVICE_{QUEUES, EVENTS}, CL_DEVICE_MAX_WORK_GROUP_SIZE, CL_DEVICE_MAX_WORK_ITEM_{DIMENSIONS, SIZES}, CL_DEVICE_MEM_BASE_ADDR_ALIGN, CL_DEVICE_NAME, CL_DEVICE_NATIVE_VECTOR_WIDTH_{CHAR, INT}, CL_DEVICE_NATIVE_VECTOR_WIDTH_{LONG, SHORT}, CL_DEVICE_NATIVE_VECTOR_WIDTH_{DOUBLE, HALF}, CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT, CL_DEVICE_{OPENCL_C_VERSION, PARENT_DEVICE}, CL_DEVICE_PARTITION_AFFINITY_DOMAIN, CL_DEVICE_PARTITION_MAX_SUB_DEVICES, CL_DEVICE_PARTITION_{PROPERTIES, TYPE}, CL_DEVICE_PIPE_MAX_ACTIVE_RESERVATIONS, CL_DEVICE_PIPE_MAX_PACKET_SIZE, CL_DEVICE_{PLATFORM, PRINTF_BUFFER_SIZE}, CL_DEVICE_PREFERRED_VECTOR_WIDTH_{CHAR, INT}, CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE, CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF, CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG, CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT, CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT, CL_DEVICE_PREFERRED_INTEROP_USER_SYNC, CL_DEVICE_PROFILE, CL_DEVICE_PROFILING_TIMER_RESOLUTION, CL_DEVICE_SPIR_VERSIONS, CL_DEVICE_QUEUE_ON_DEVICE_PROPERTIES, CL_DEVICE_QUEUE_ON_HOST_PROPERTIES, CL_DEVICE_QUEUE_ON_DEVICE_MAX_SIZE, CL_DEVICE_QUEUE_ON_DEVICE_PREFERRED_SIZE, CL_DEVICE_{REFERENCE_COUNT, VENDOR_ID}, CL_DEVICE_SVM_CAPABILITIES, CL_DEVICE_TERMINATE_CAPABILITY_KHR, CL_DEVICE_{TYPE, VENDOR}, CL_{DEVICE, DRIVER}_VERSION

Partitioning a Device [4.3]

cl_int **clCreateSubDevices** (cl_device_id *in_device*, const cl_device_partition_property **properties*, cl_uint *num_devices*, cl_device_id **out_devices*, cl_uint **num_devices_ret*)

properties: CL_DEVICE_PARTITION_EQUALLY, CL_DEVICE_PARTITION_BY_COUNTS, CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN

cl_int **clRetainDevice** (cl_device_id *device*)

cl_int **clReleaseDevice** (cl_device_id *device*)

Contexts [4.4]

cl_context **clCreateContext** (const cl_context_properties **properties*, cl_uint *num_devices*, const cl_device_id **devices*, void (CL_CALLBACK**pfn_notify*)(const char **errinfo*, const void **private_info*, size_t *cb*, void **user_data*), void **user_data*, cl_int **errcode_ret*)

The OpenCL Runtime

API calls that manage OpenCL objects such as command-queues, memory objects, program objects, kernel objects for __kernel functions in a program and calls that allow you to enqueue commands to a command-queue such as executing a kernel, reading, or writing a memory object.

Command Queues [5.1]

cl_command_queue **clCreateCommandQueueWithProperties** (cl_context *context*, cl_device_id *device*, const cl_command_queue_properties **properties*, cl_int **errcode_ret*)

properties: [Table 5.1] CL_QUEUE_SIZE, CL_QUEUE_PROPERTIES (bitfield which may be set to an OR of CL_QUEUE_* where * may be: OUT_OF_ORDER_EXEC_MODE_ENABLE, PROFILING_ENABLE, ON_DEVICE[_DEFAULT])

cl_int **clRetainCommandQueue** (cl_command_queue *command_queue*)

cl_int **clReleaseCommandQueue** (cl_command_queue *command_queue*)

cl_int **clGetCommandQueueInfo** (cl_command_queue *command_queue*, cl_command_queue_info *param_name*, size_t *param_value_size*, void **param_value*, size_t **param_value_size_ret*)

param_name: [Table 5.2] CL_QUEUE_CONTEXT, CL_QUEUE_DEVICE, CL_QUEUE_SIZE, CL_QUEUE_REFERENCE_COUNT, CL_QUEUE_PROPERTIES

©2015 Khronos Group - Rev. 1115

Message-Passing Interface (MPI)

- Highly used for High-Performance Computing (grids, clusters)
- Mostly message-passing and synchronization primitives, to be applied with any general-purpose sequential language (in fact Fortran or C++ generally)
- Point-to-point or collective (broadcast, all-to-all) communications
- Means to describe virtual network topology as graph
- Big issue is to assign processes to processors (statically?), and to map the virtual communication onto the real interconnect infrastructures
- Both prominent frameworks (OpenMPI and MPICH) have fancy dedicated library, commercial offers are made by specific hardware vendors
- Mapping done at runtime, but what about static control processes and static mapping at compile time ?

MPI quick card (excerpt)

- **Communicators with Topology**
- Create with cartesian topology. (§ 6.5.1)
- int **MPI_Cart_create** (MPI_Comm comm_old,
- int ndims, int *dims, int *periods, int
- reorder, MPI_Comm *comm_cart)
- Suggest balanced dimension ranges. (§ 6.5.2)
- int **MPI_Dims_create** (int nnodes, int
- ndims, int *dims)
- Determine rank from cartesian coordinates. (§ 6.5.4)
- int **MPI_Cart_rank** (MPI_Comm comm, int
- *coords, int *rank)
- Determine cartesian coordinates from rank. (§ 6.5.4)
- int **MPI_Cart_coords** (MPI_Comm comm, int
- rank, int maxdims, int *coords)
- Determine ranks for cartesian shift. (§ 6.5.5)
- int **MPI_Cart_shift** (MPI_Comm comm, int
- direction, int disp, int *rank_source,
- int *rank_dest)
- Split into lower dimensional sub-grids. (§ 6.5.6)
- int **MPI_Cart_sub** (MPI_Comm comm, int
- *remain_dims, MPI_Comm *newcomm)
- *Related Functions:* MPI_Graph_create, MPI_Topo_test,
- MPI_Graphdims_get, MPI_Graph_get,
- MPI_Cartdim_get, MPI_Cart_get,
- MPI_Graph_neighbors_count, MPI_Graph_neighbors,
- MPI_Cart_map, MPI_Graph_map

Compilation and Runtime Execution

- HPC Runtime: StarPU, XKaapi, RMC
- Parallel compilation & Polyhedral model : ClooG, Graphite, R-Stream
  - (often produces OpenMP-ish code)

These always operate by trying to exhibit maximal paralleism on the application side (and the kind of parallelism wanted first), then try to adjust it dynamically to a “phntom” architecture only reflected by a few features showing avaibility of simple resources

Can we do better with a real architecture model?

(Of course static control restrictions for compile-time decisions)

Data/Task parallel levels: Nested For-Loop programs with affine bounds

Both code *and* model:

- Strict separation between indexes and other variables
  (only indexes impact control)
- Other variables are array locations
  (referenced by affine expressions of indexes)

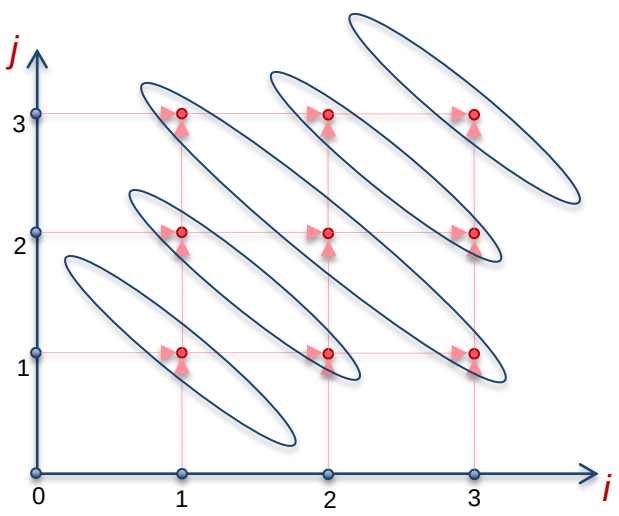
May produce explicit control/data flowgraphs to extracy potential parallelism
(from dependences between operations)

- Extended: precise but untractable
- Reduced: quality of solution depends on information preserved
  - Dependence levels (Allen-Kennedy)
  - Direction vectors (Lamport, Wolf & Lam)
  - Polyhedral models (Feautrier)

In all 3 cases, solutions found are expressed by regular (affine) scheduling relations between operations : logical clocks !

Silly simple example

```
for j = 1 to N
   for i = 1 to N
        a(i+1,j+1)= a(i, j+1) + a(i+1,j)
   end
end
```



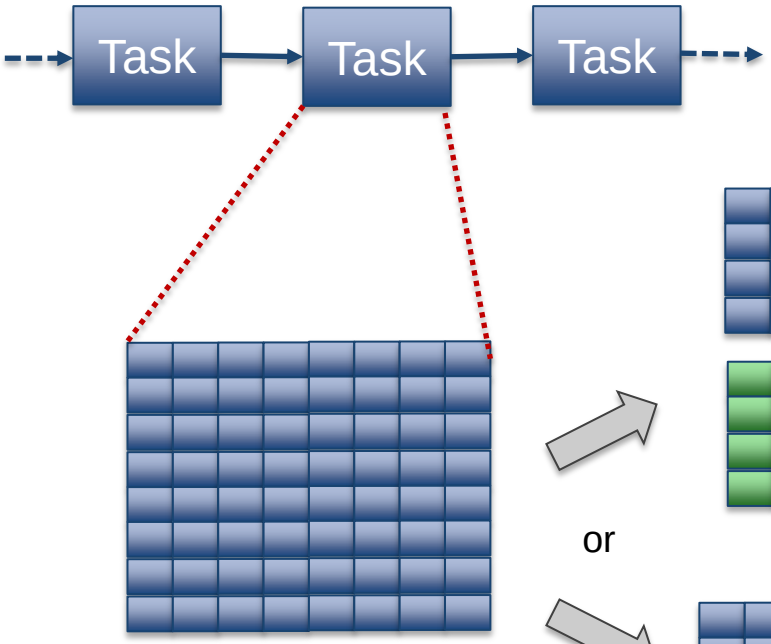
dependences

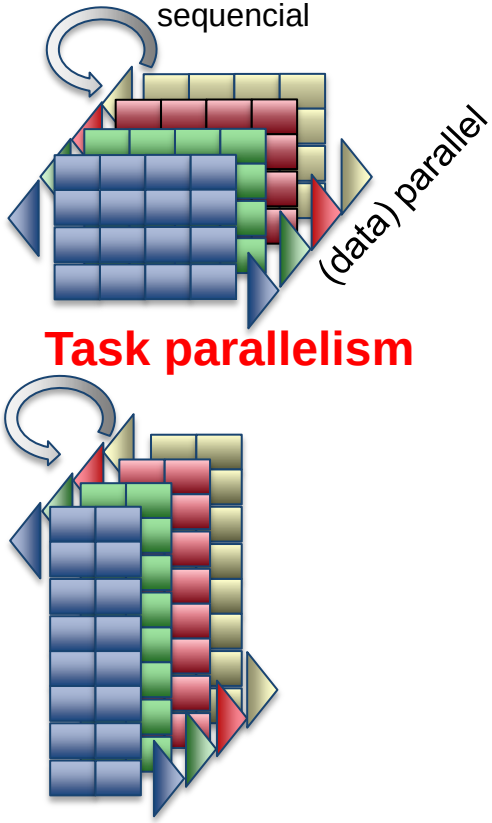

```
for j' = 2 to 2N
   parfor i' = max(1,j'-N) to min(N,j'-1)
        a(i,j'-i')= a(i'-1, j'-i') + a(i',j'-i'-1)
   end
end
```

step(a(i,j))= i+j-1

Tiling

Task streaming, software pipelining





Task parallelism

Data parallelism

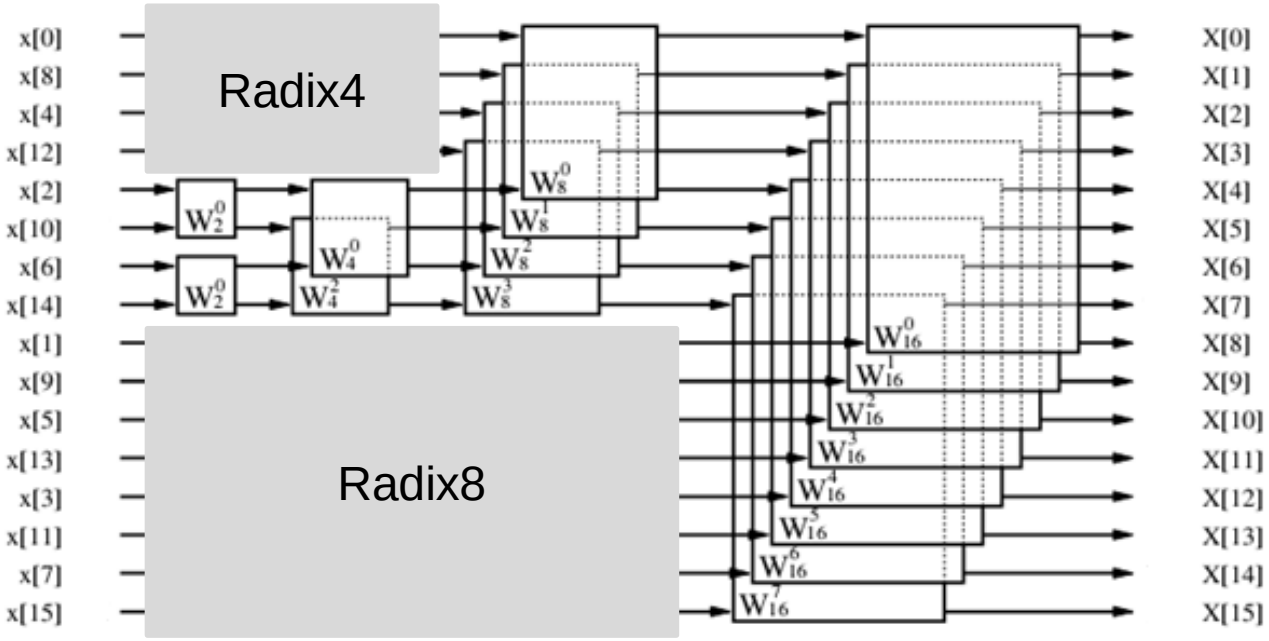
- Change the granularity
- …to adjust it to match the size of a single Processing Element
- Locality and shared memory space makes performance highly predictable

Typical (and useful) library kernels

- **Linear algebra**: LinPack ScalaPack (Blast),…
- **Convolutions** Fast Fourier Transform: FFTW, **Spiral**
- **Neural Networks** (only execution, not training part): a mix of both
- **Signal Processing** (filters) : **Halide**
- Stencil algorithms for numerical analysis ./ scientific computations

This type of effort lead to **Domain-Specific Languages**
Restricted expressivity: linear or DAG fillter streams
Transformations to exhibit some form of parallelism (data or vectorize)
Scheduling often not fully automatic, or obtained by progressive selection (Dynamic Programming, Auto-tuning, or “Auto-Scheduler”…)
…although algorithm stream definitions are often recursive in terms of the data (multi-array) size, so that with a simple core architecture target model it is easy to compute which size fits in (and then loop sequentially to get results). (Application ground floor resembles architecture groundfloor)

Fast Fourier Transform



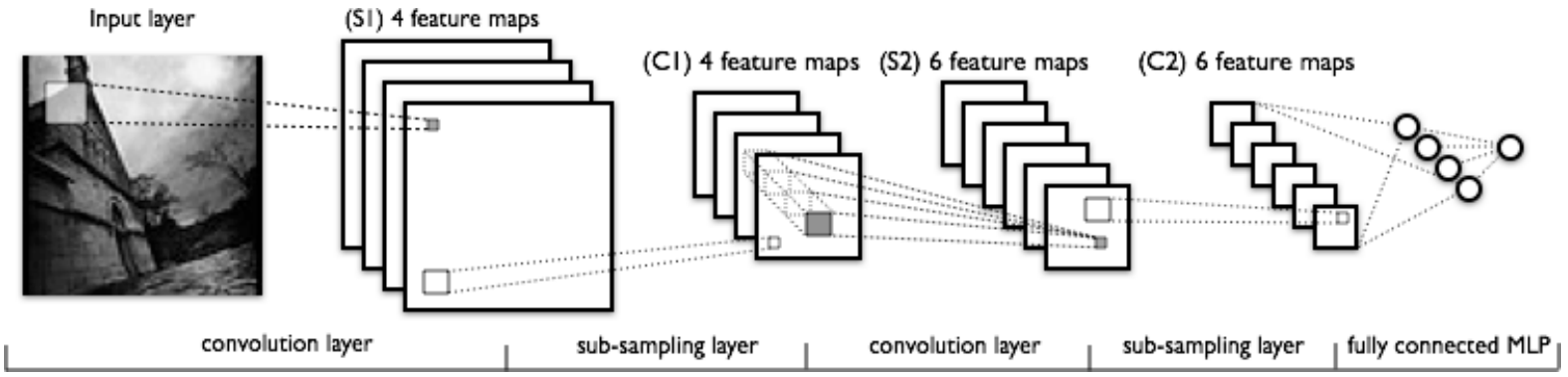
Can easily be decomposed (recursively up to intermediate permutations) along sizes 2^N

Given a target processing core (or thread):

- **its local register count 2^R**
- **its local memory 2^M**
- **its number of vectorized (simd) or parallel (GPU) ALUs 2^C**

one can compute by proportions a pattern adjusting the available parallelism (and sequential loops around)

Convolutional Neural Networks



source: http://deeplearning.net/tutorial/lenet.html

Task streamin for layers
+ potential data parallelism according to locality and tile overlap

Dream…

When

- Application restricted to simple filter DAG/pipeline kernel
- Data value organization well understoood (and statically known)

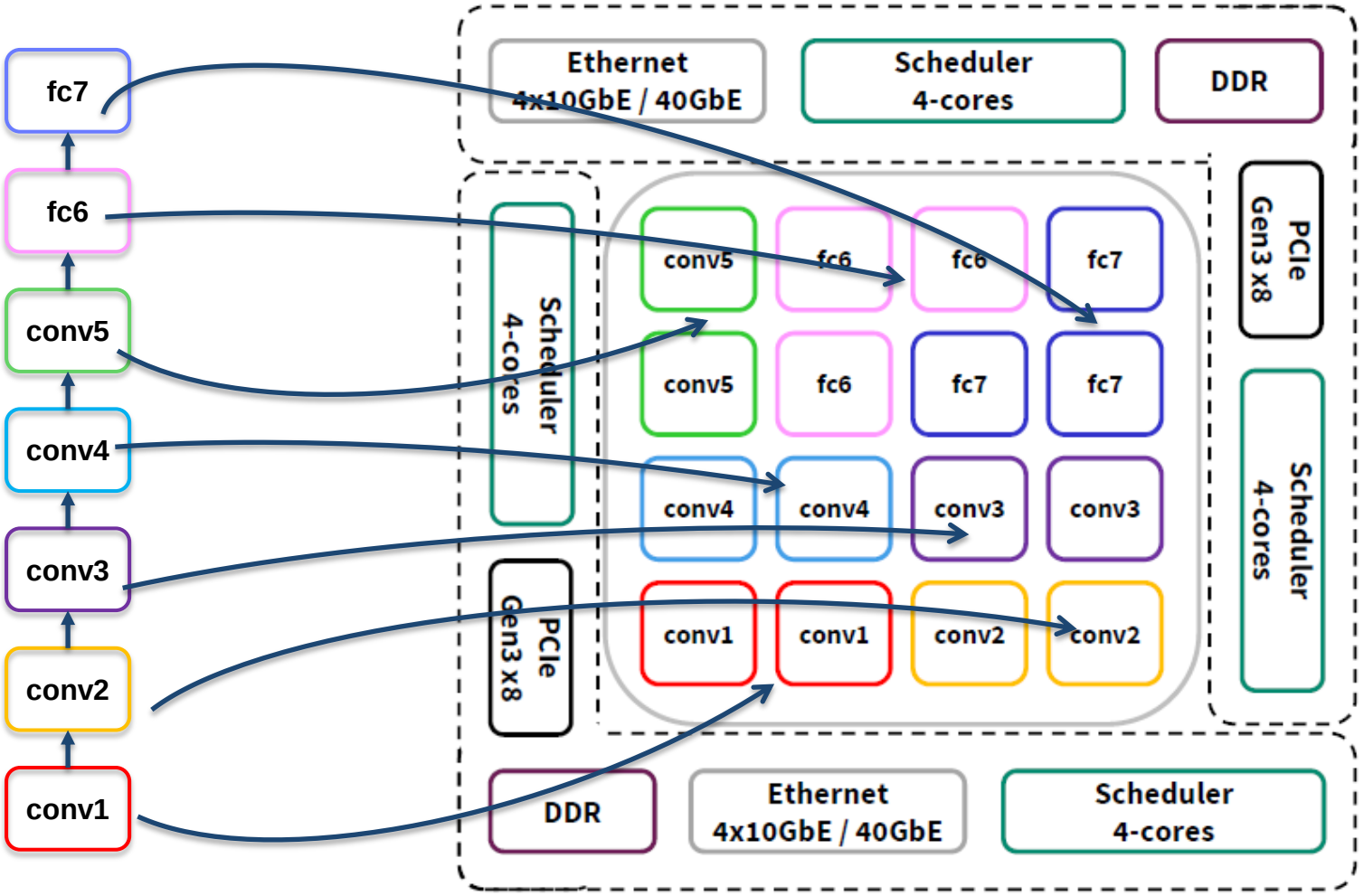
- Simple processor targeted (single core, processing element, or thread)
- Shared memory, low-cost communication (data transfer)

- Mutual correspondance of sizing between appli and archi, and simple cost function formulas,

Then the ground floor mapping of library kernels to core should deliver time-predictable results (WCET)

Then one could consider optimizing full (static control) applications on full platform, designing time-predicatble communication pattterns.

ConvNets on Kalray MPPA



source: B. Ganne (Kalray) , Machine Learning on Multicores, NeuroStic Days 2015 (in French)

FFT on Intel Haswell

(Mohamed Bergach PhD thesis)

SIMD CPU cores (4 cores)

- each core enough L1/register memory to execute 4 stages at once :

  2 256-bit SIMD ALUs, 16 256-bit registers, 128 floating-point numbers

- FFT 1064 (radix 10, 10 stages) performed as iteration 4+4+2
- in C++ with simd intrinsics

GPU Processing Elements (40)

- each PE enough L1/register memory to execute 3 stages at one

  2 128-bit ALUs, 2K bit registers, 64 floating-point numbers

- FFT 1024 executed as iteration 3+3+3+2
- in OpenCL

To force allocation one needs affinity (dismiss hyperthreading etc), currently not obvious

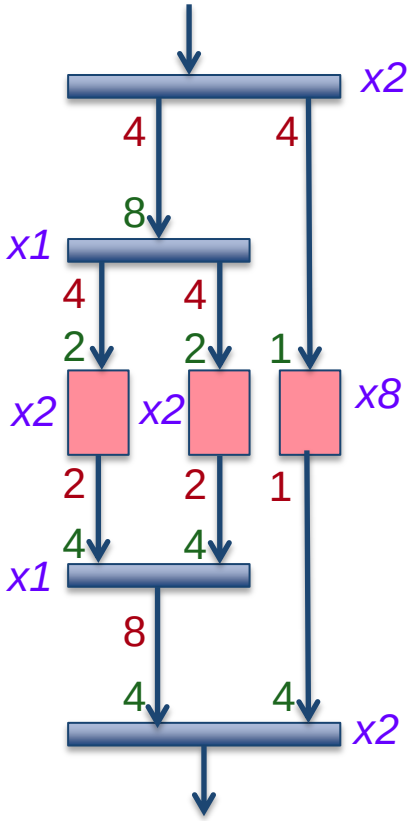
In the end one gets a libray of mapped functions with reasonably accurate cost, to be used by upper floor analyses

4

Concurrent Models of Computation and Communication

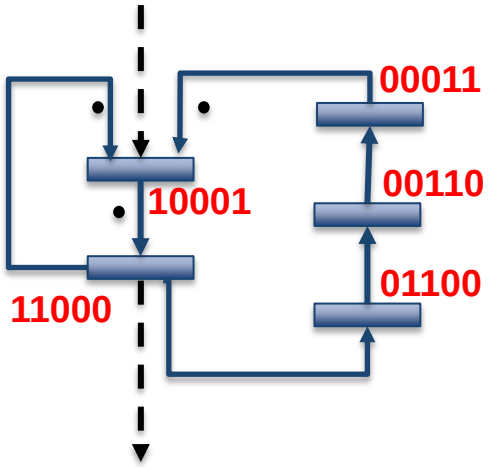
Process Networks, Scheduling and Routing

SDF (synchronous data-flow process networks)



- Weighted Marked Graphs (conflict-free subset of Petri Nets)
- Distinct issues for acyclic graphs and strongly connected components
- First-level self-timed semantics
- Second stage: optimal scheduling exist, assuming no parallelism limitations
- Static schedules lead to regular form (the binary word of activations of any node is of specific form
- all nodes adopt the same rate

Regular schedules in SDF



The noticeable thing here:

- (static) schedules can be represented as regular infinite binary words

  u.(v) , **u** initial, **v** repeated

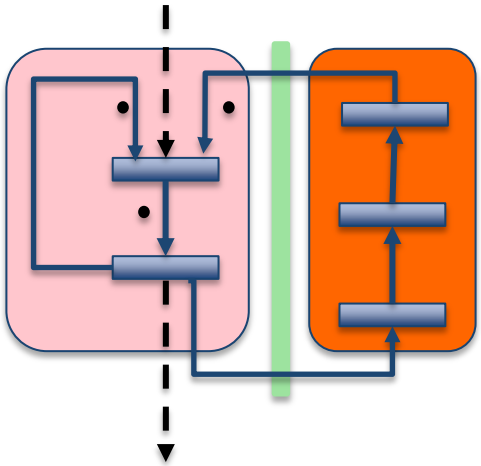
where

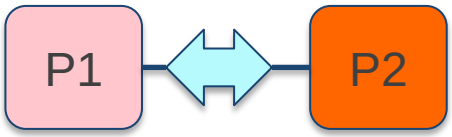
- 1 at location n means active at step n
- 0 at location n means idle at step n

While very primitive, it corresponds to the way ASAP schedule assignment goes
Later it can be turned into Gantt Charts, when sequences get long

No architectural resource constraints considered here: ideal parallelism

Regular schedules in SDF





Now what if we map (here superpose) to an execution platform ?

- Cost of communication is no longer uniform (nuMA)
- Computations or communications sharing a single resource may need to be serialized (multitasking)

But the principles of asap scheduling stays the same !
Even more constraints may be applied and task fission/fusion also…

To what extent can we represent in this abstract setting the phenomena encountered earlier ?

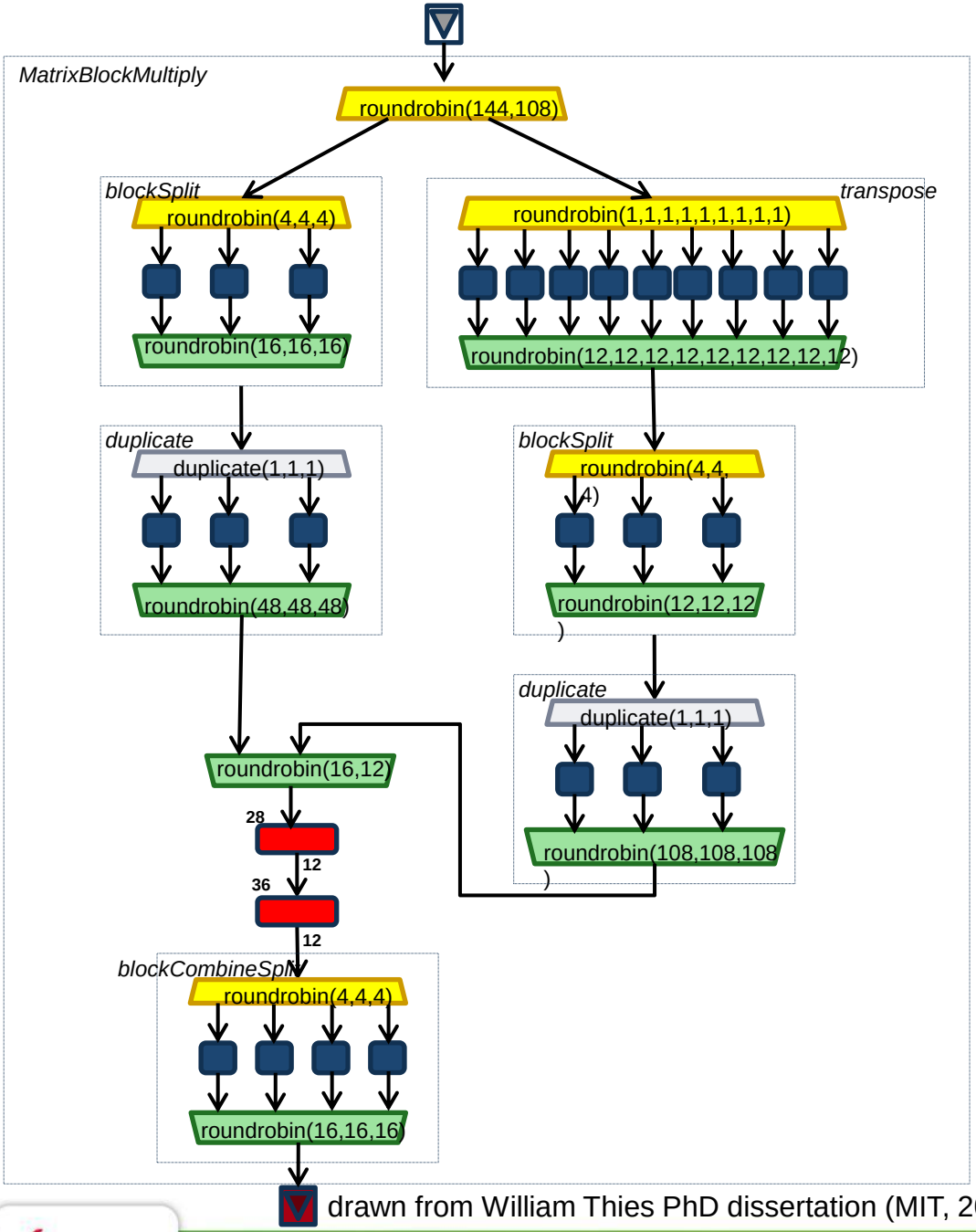
Adding predictable deterministic control to SDF

- add a switch node), but with internal switching condition (as in Kahn Process Networks → functional determinism, latency-insensitive design

- done in Cyclo-Static Data-Flow and StreamIt graphs (but without initialization patterns
- Our own view: add regular **switching/routing** patterns in the *same flavor* as schedule words activating conditions

→ KRG process network model

- Study (equational, algebraic) graph transformations that preserve functionality (self-timed)
- these transformations made to change the buffer/data dependencies to adjust to a given platform communication topology graph (eg, NoC)

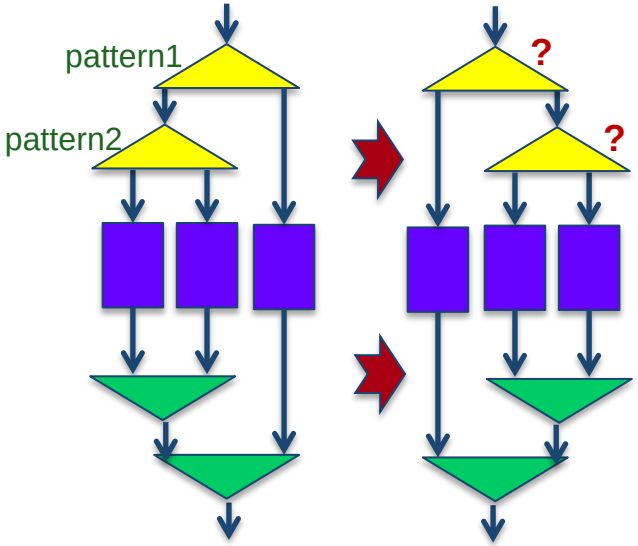


drawn from William Thies PhD dissertation (MIT, 2009)

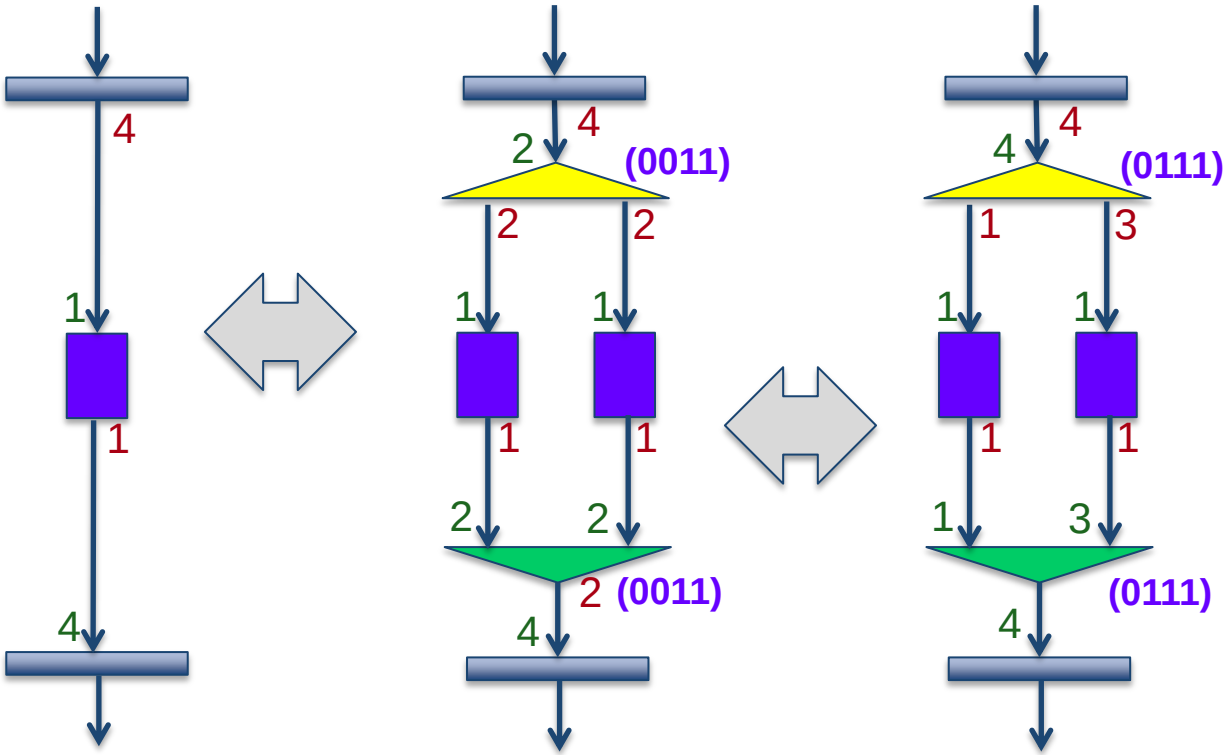
StreamIt example

roundrobin branch dispatcher and **duplicate** parallel output confused as “splitters” (unfortunate?)

Q.



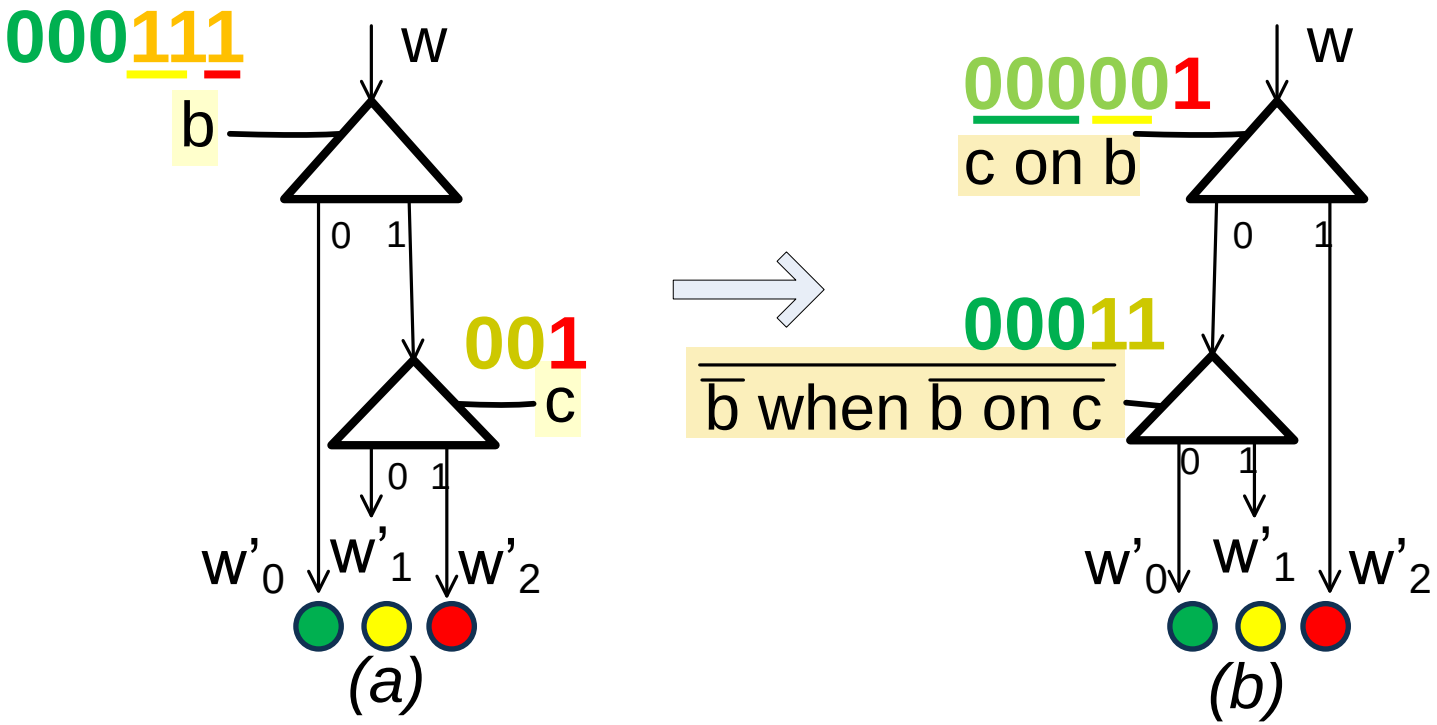
Silly little KRG example



split/demux/map node

merge/mux/reduce node

Transitivity of Selects



on/when operators

(0.u) **on** v = 0.(u **on** v)

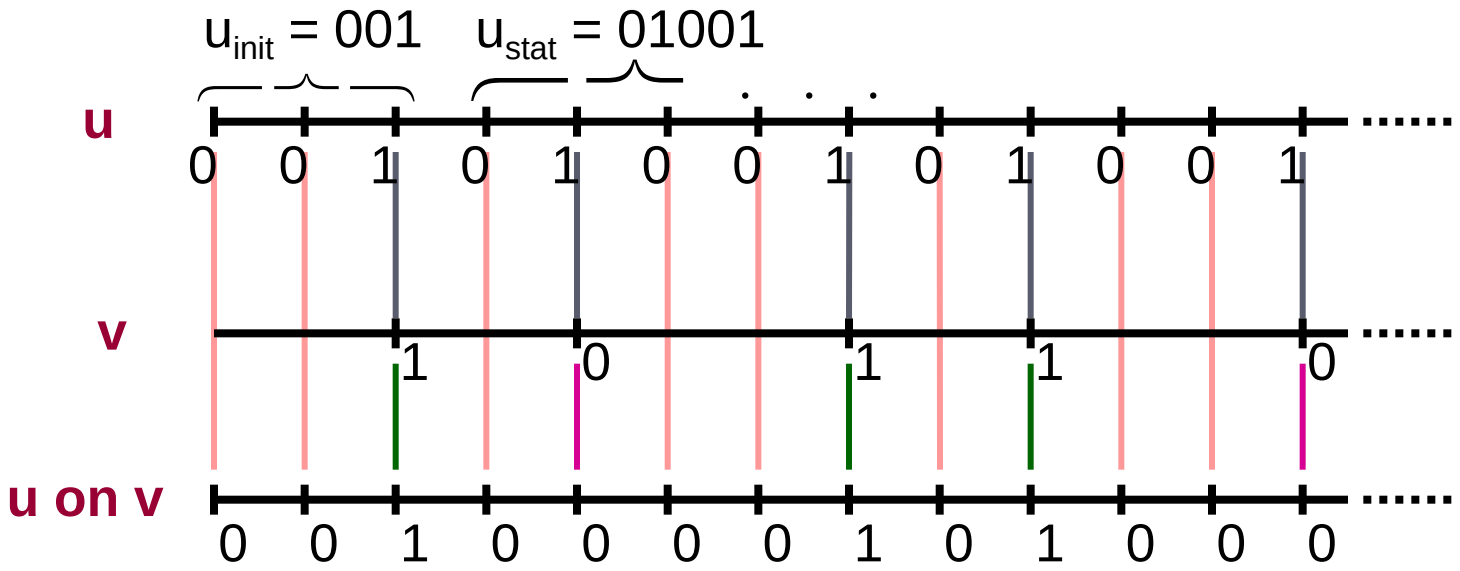
(1.u) **on** x.v = x.(u **on** v)

(x.u) **when** (0.v) = u **when** v

(x.u) **when** (1.v) = x. (u **when** v)

on effects

(0.u) **on** v = 0.(u **on** v)
(1.u) **on** x.v = x.(u **on** v)

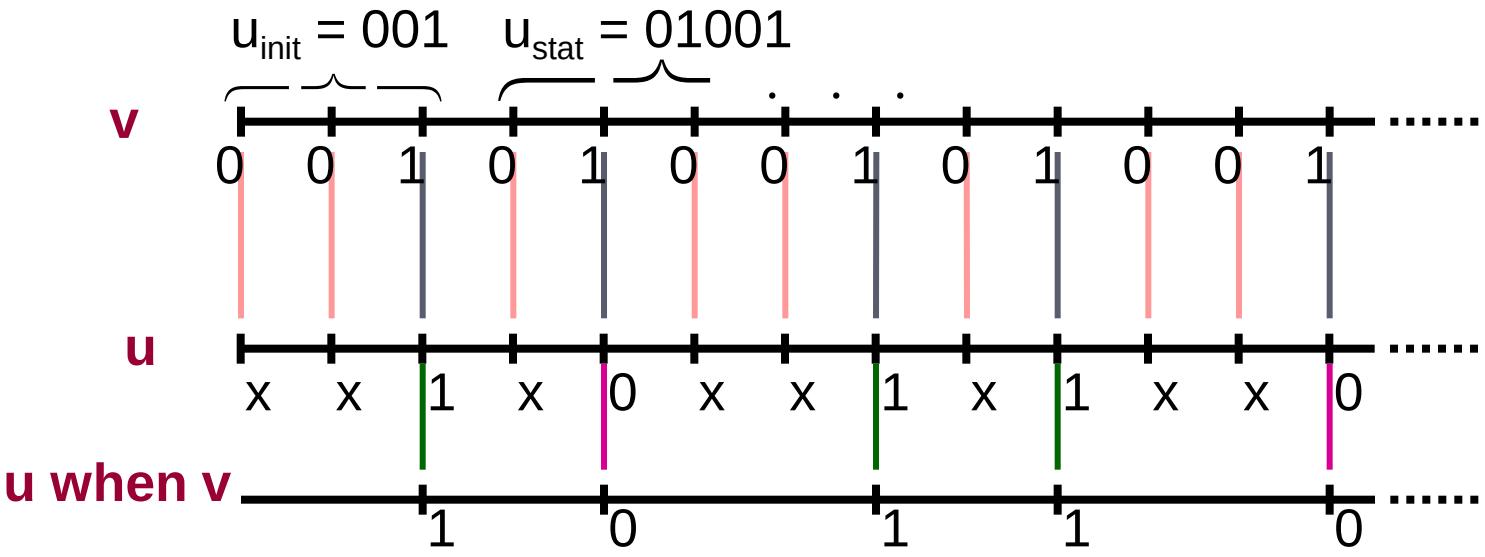


when effects

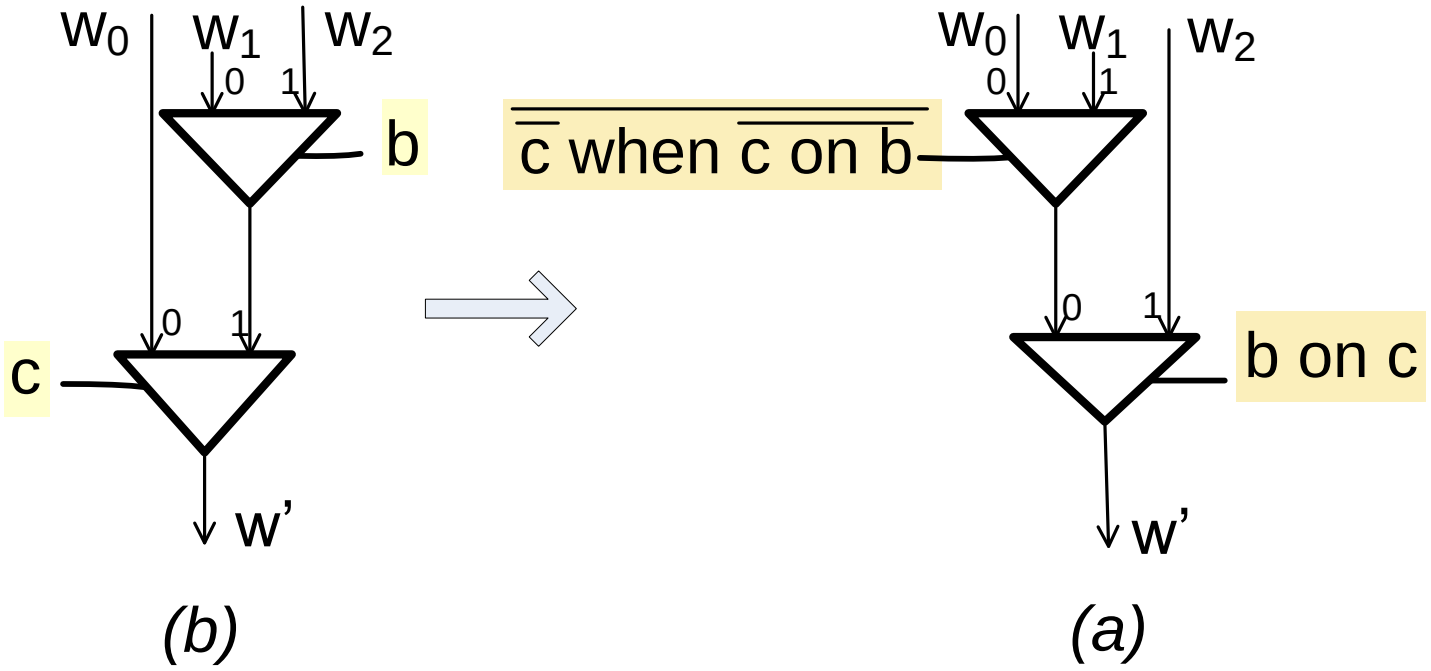
most meaningful whenever
u subclock of v

(x.u) **when** (0.v) = u **when** v

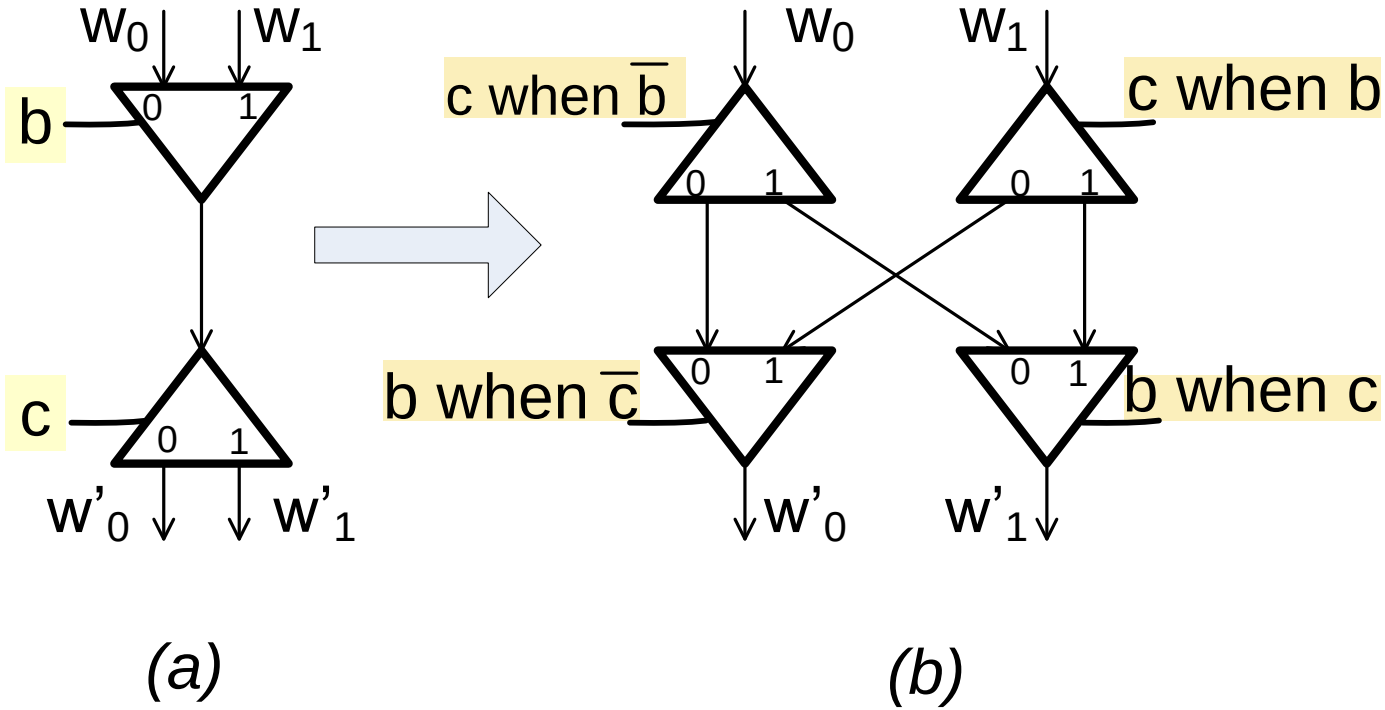
(x.u) **when** (1.v) = x. (u **when** v)



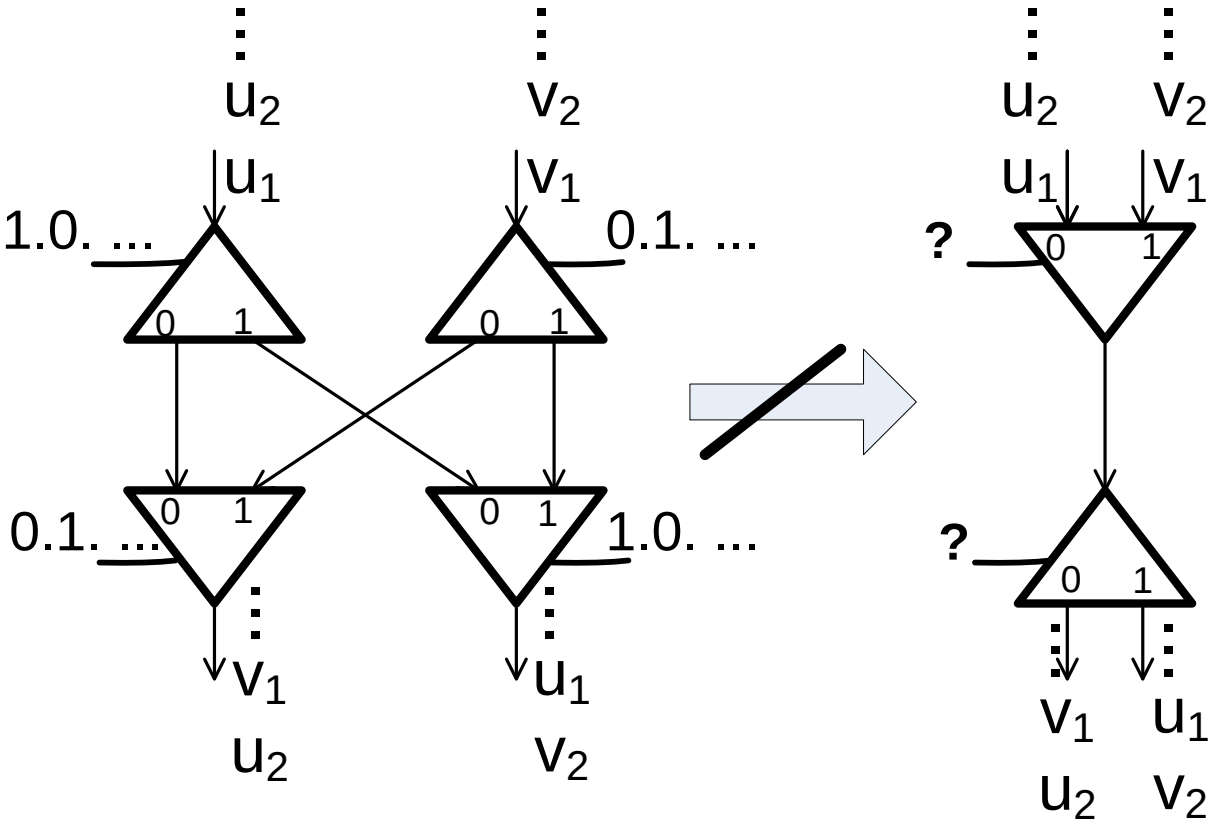
Transitivity of Merges



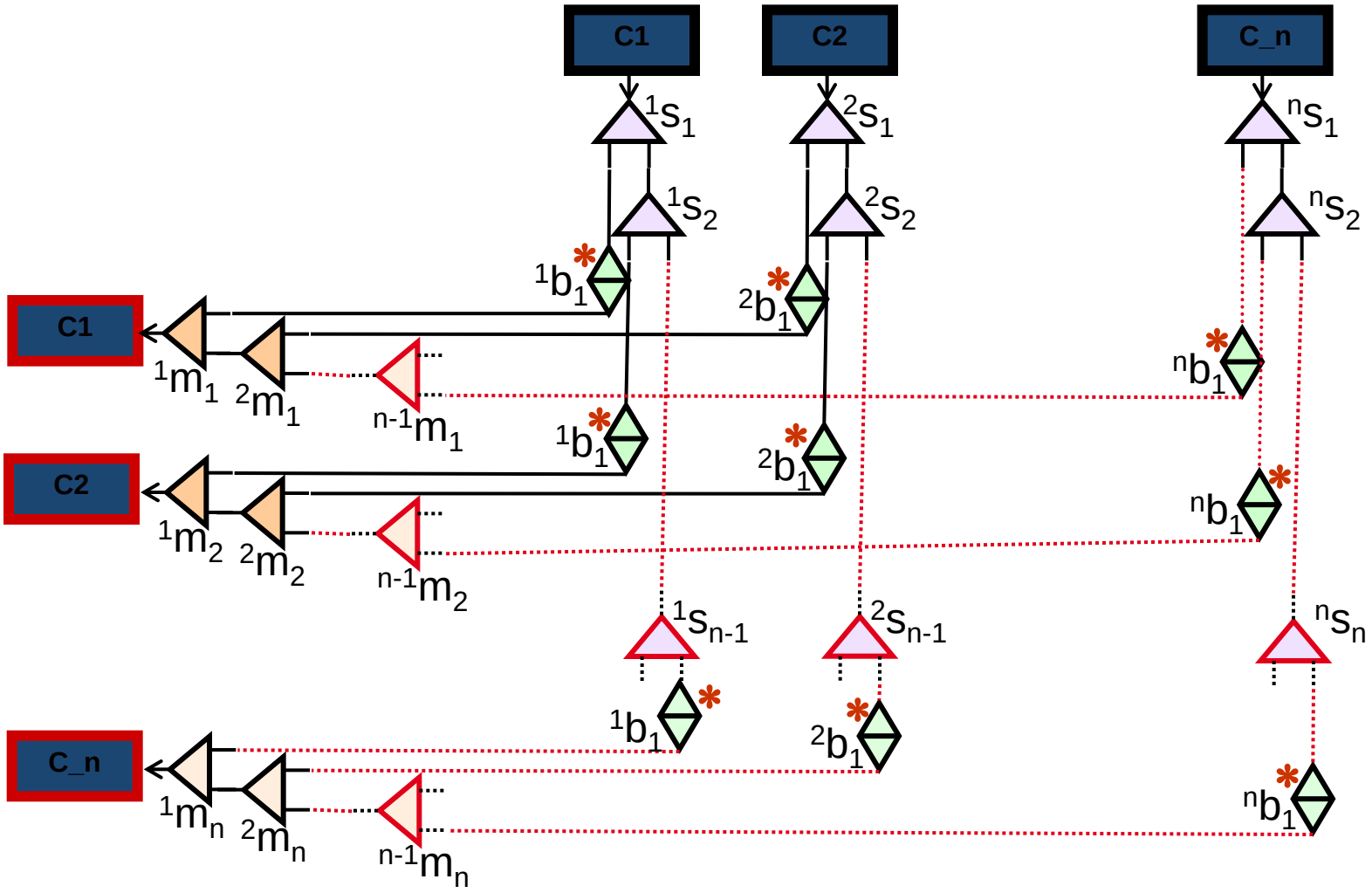
Selects up across Merges



Sharing vs disorder



Normal forms (point-to-point links)



Lessons

Regular scheduling arises naturally as solution space for optimal results

- **Classical scheduling of Process Networks**
- **Scheduling of nested For-loops programs wth affine bounds**
- **…**

Regular switching patterns can match the expressive level of description

- **represent the transformations in data transfer/communication pattern**
- **play with the boundaries between data- and task- parallelism**

Task streaming level: Scheduling Theory

Original problem is to schedule independant tasks with hard deadline requirements on a mono- or multi-processor

- static scheduling of periodic tasks: Rate-Monotonic Analysis, Deadline-Monotonic
- dynamic schedling of periodic/sporadic preemptible tasks: Earliest-deadline-first, least-laxity first,…

Simplified assmptions lead to positive methods/results

Then accounting for communications, context-switches in preemption, and further issues make life harder (and literature bigger)

First step is to specify in clear mathematical formulas what are the real-time constraints

5

A Clock Constraint Specification Language

Declarative specifications for the top floor

Multiform Logical Time

Main idea is simple:

every (pure) event that occurs repeatedly can be used as a *logical clock*

logical clock = sequence/flow of ticks/signals = activation condition

examples:

- (physical) clock cycles in a modern embedded processor
- ignition in a car (4 times each turn of engine, no matter the speed)
- sensor event detection (such as infra-red cells, gesture detection, lidars,..)
- *by the way, regular clockwatch physical time can also be considered logical…*

We claim Multiform Logical Time is natural and invaluable at design/specification time

- Expanding multiform logical time to uniform physical time depends on implementation conditions

Clock Constraint Specification Language) CCSL

- Meant to express constraints and properties in (multiform) logical time
- Targets the platform-based design/AAA framework

- Inspired from Synchronous (actually polychronous) languages
- Inspired from Classical and Real-Time Scheduling notions.

- Not usually stand-alone (extracts the ordering relations between events whose meaning is part of a larger specification)
- Formal syntax to reason about logical time relations (including simultaneity)
- Concerns for expressivity and decidability

Two natural partial orders

a subclock b ($a \subset b$)

- inspired from synchronous (polychronous) languages
- inpired from shapes of regular static scheduling / parallel allocation
- inspired from hardware and system design

a faster than b ($a \leq b$)

- inspired from real-time scheduling
- inspired from Timed Automata
- …

used either in an imperative or declarative fashion
each tractable individually, but only the combination truly expressive
… and problematic

Brief recap on Synchronous languages 1. Signal /Lustre

x = fct(y,z)

x = init -> pre(y)

x = y default z

x = y when z (z:Boolean flow)

c = when z, x = y @ c

^x=^y

^x#^y

c_y = c_z c:= c_y

c_x := c_y

c_x = c_y union c_z

c = c_z filtered_by ?z (?z Boolean sequence)

c_x = c_y inter c

c_x = c_y

c_x # c_y

Logical clocks not syntactic elements in Lustre, no totally first-class in Signal, explicit activation condition in Scade

Brief recap on Synchronous languages 2. Esterel

pause (next instant)
P;Q
P || Q
loop P end
present S then P else Q
if ?S then P else Q
emit S
signal S in P end

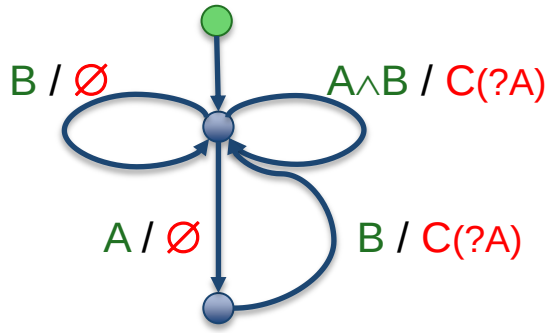
start P when S
abort P when S
stop P when S
suspend P when S
activate P when S

+ sequential computations on data variables

Signals are first-class citizen logical clocks
They represent shared variables with precise constructive consensus semantics

An example program

```
loop
  await A;
  await immediate B
  emit C(A?)
end
```

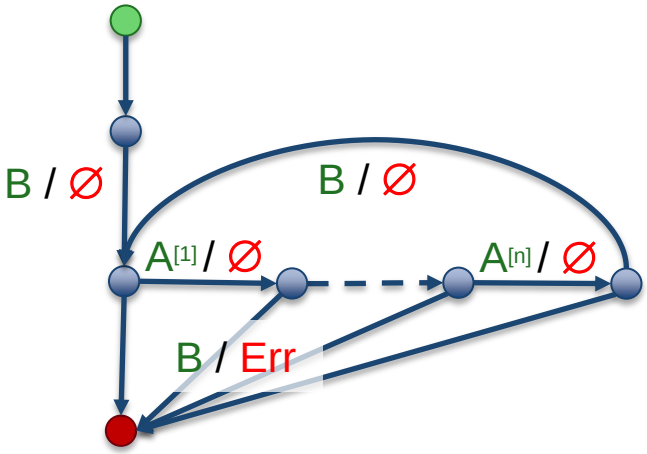


C= A sampledOn B

- Logical clocks are the event-driven control structures (with registers that are latching instants)
- new clocks from old clock (only present-else case is problem, but usually triggered by some previous clock/register)
- Simple Mealy machine interpretation for each construct
- “Sensible” clocks should tick infinitely often

A second example

```
(weak)every B
  do (weak)abort
          await B; emit Err
       when n*A
  done
end
```



B SporadicOn A

- Constraints seen as Observers
- Assume/Guarantee

CCSL in a foil

a, b, c .. clocks, w mask (infinite regular binary word)

expressions

a union b	only way to « speed up » rate (more ticks)
a inter b	(less ticks, may become finite)
a minus b	(less ticks, may become finite)
fastest(a,b)	min of timings
slowest(a,b)	max of timings
sync(a,b)	when timings match

a filteredBy w	subsampling, solution-oriented (k-periodic, regular)
a sampledOn b	sporadic sampling

relations/constraints

a = b

a ≠ b

a ≤ b

a < b

a # b

a AlternatesWith b

Results (extracts)

Each individual constructs translates into a:

transition-labelled extended Büchi Mealy transition systems.

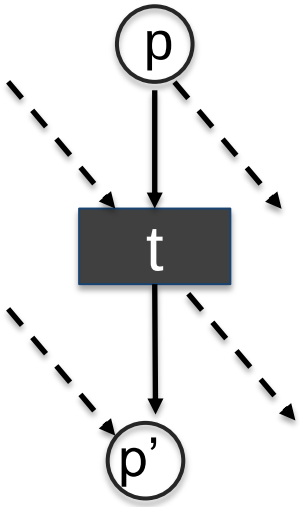
Not always Finite-State machine, also integer counters
(cf fastest(C,D),or C ≤ D)

Full system is a parallel product of such t-E Büchi Mealy machines

Too Expressive (can encode Petri Nets with inhibitor arcs, and similar arguments as classical 2 counter machines) → Turing-Complete

But many sources of sensible restrictions ensuring decidability

Encoding asynchronous Petri Nets



Clocks

- **t_p**, **p_t** for each t, p connected
- **t** for each t *(firing the transition)*
- p_in and **p_out** for put/get token to/from that place

t = **inter**(**p_t**, p in •t) ∩ **inter**(**t_p**, p in t•)

t1 # t2 … # tn

(transition are exclusive in time, not to mix tokens)

p_in = **union**(**t_p**, p in •t)

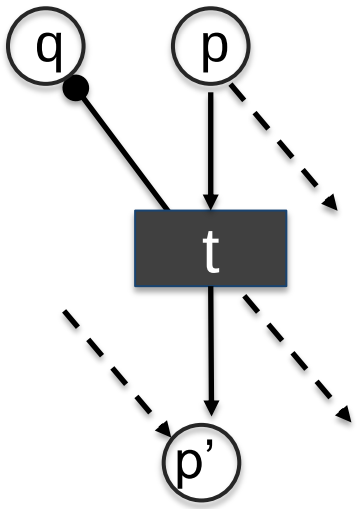
p_out = **union**(**p_t,** p in t•)

(**p_out shiftedBy init_Tok_p**) ≥ **p_in**

(tokens consumed only after being produced)

Note: in general Petri Nets, weights can be replaced by multiple places)

Encoding PN inhibitor arcs



t = inter(p_t, p in •t) ∩ inter(t_p, p in t•)

t1 # t2 … # tn

(transition are exclusive in time, not to mix tokens)

p_in = union(t_p, p in •t)

p_out = union(p_t, p in t•)

(p_out shiftedBy init_Tok_p) ≥ p_in

p_in # p_out

void_q = fastest((q_out shiftedBy init_Tok_q),q_in) inter q_out

ticks when place becomes empty

unvoid_q = void_q SampledBy q_in

ticks when place becomes occupied

t SampledOn (void_q union unvoid_q) = t SampledOn unvoid q

void alternates with unvoid, these main equation states that t occurs after the void and before the unvoid

Schedulability issue: simplest example

- $B \subseteq A$ A filtered by (01) = B filtered by (01)

Here in fact B=A, since all even occurrences must coincide (while odd occurrences seem free), B can never catch up with any delay.

But it may be noticed too late in plain simulation (without backtracking)

Many tractable subsets

If the clock dependency is a forest (no reconvergence)
If it is a DAG but without synchronizing operators (inter,..) in the middle
More to be defined
…

6

Conclusions

Thank you

www.inria.fr

6

Platform-Based Design and Y-Chart methodology

Sous-titre facultatif