PHILIPS

Interface centric approach to design and programming of embedded multiprocessors

Erwin de Kock

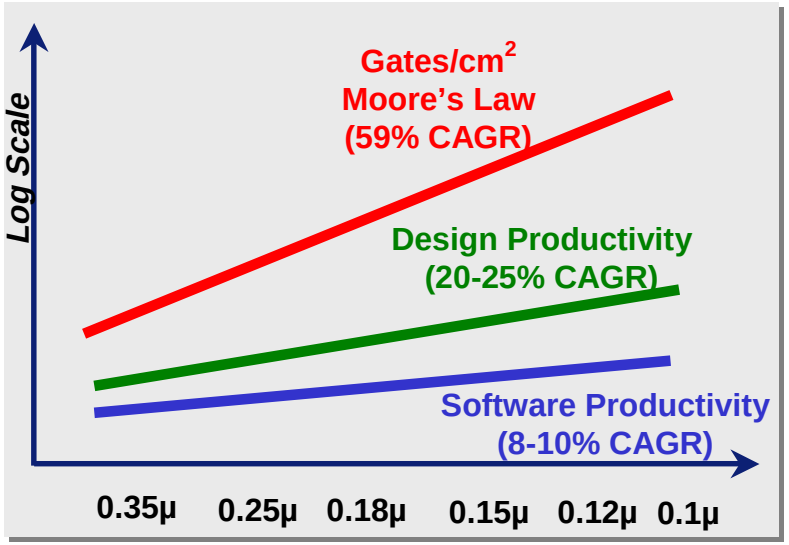
Pieter van der Wolf, Wido Kruijtzer, Tomas Henriksson,
Gerben Essink, Dennis Alders, Ondrej Popp

Outline

- Introduction
- Task Transaction Level interface: TTL
  - Abstract interface for streaming in MPSoCs
- Programming TTL multiprocessors
  - Constraint-driven code transformations
- Design cases
  - Sea-of-DSP
  - Smart Camera
  - Cake / Wasabi
- Conclusion

MPSoC Design

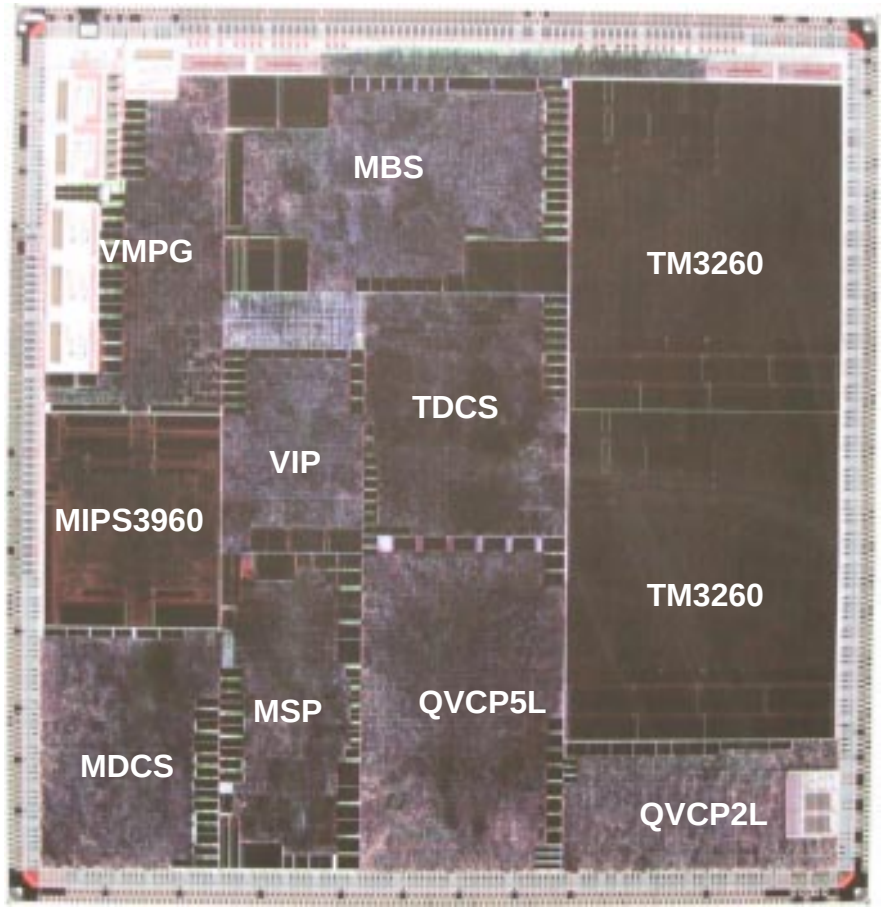
- Need for MPSoCs:
  - Implement advanced functionalities
  - Low cost
  - Power efficient
  - Flexible
- Increasing complexity of MPSoCs:
  - Increasing design efforts
  - SW effort overtaking HW effort
  - Increasing time-to-market
- Productivity increase through:
  - Raise level of abstraction
  - Structured design
  - IP reuse
  - EDA support



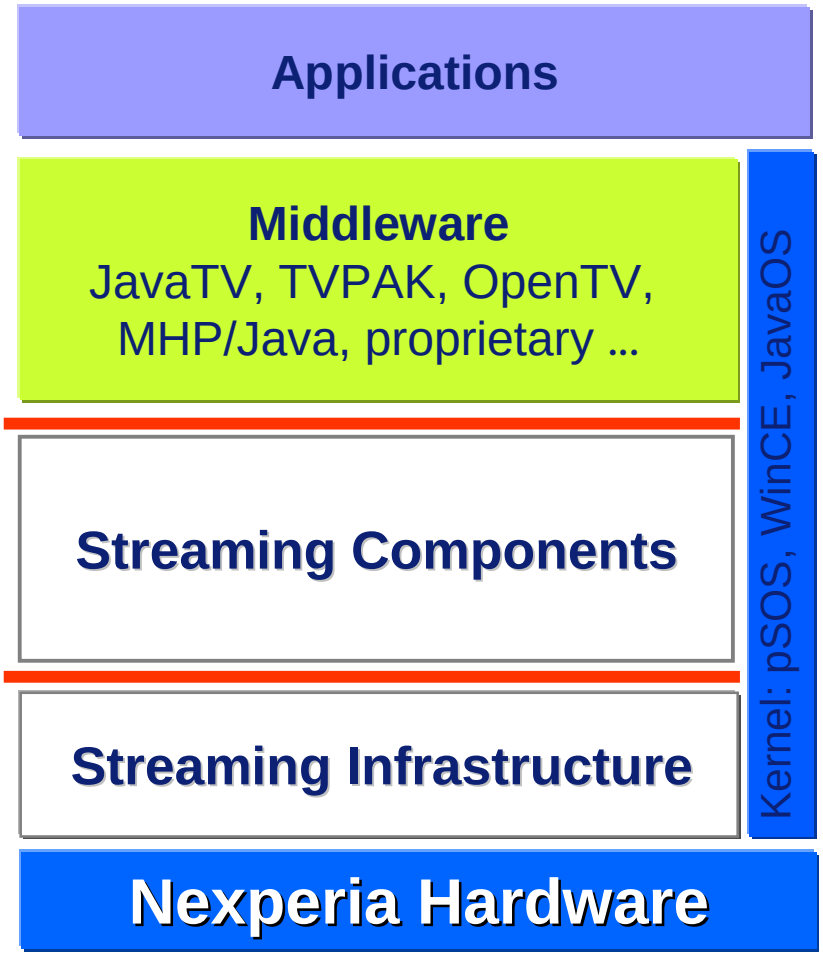
Example MPSoC Hardware

- Philips's advanced set-top box and digital TV SoC (Viper2)

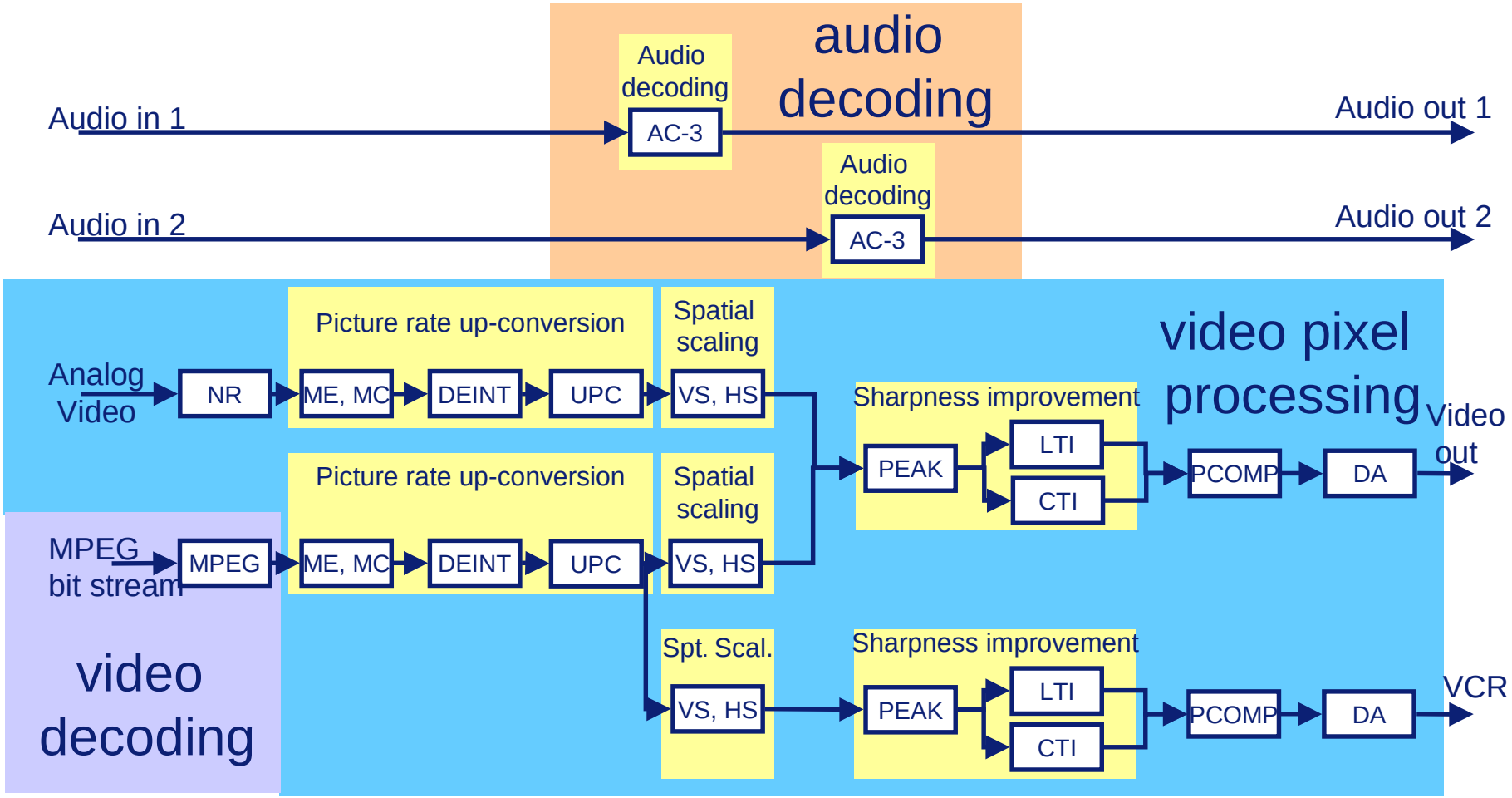
- 0.13 µm
- 50 M transistors
- 100 clock domains
- > 60 IP blocks



Example MPSoC Software Stack



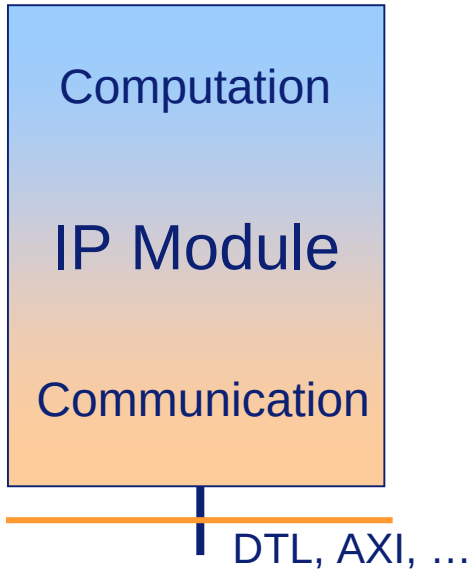
Example TV application



Many task graphs like this have to be supported

MPSoC Integration

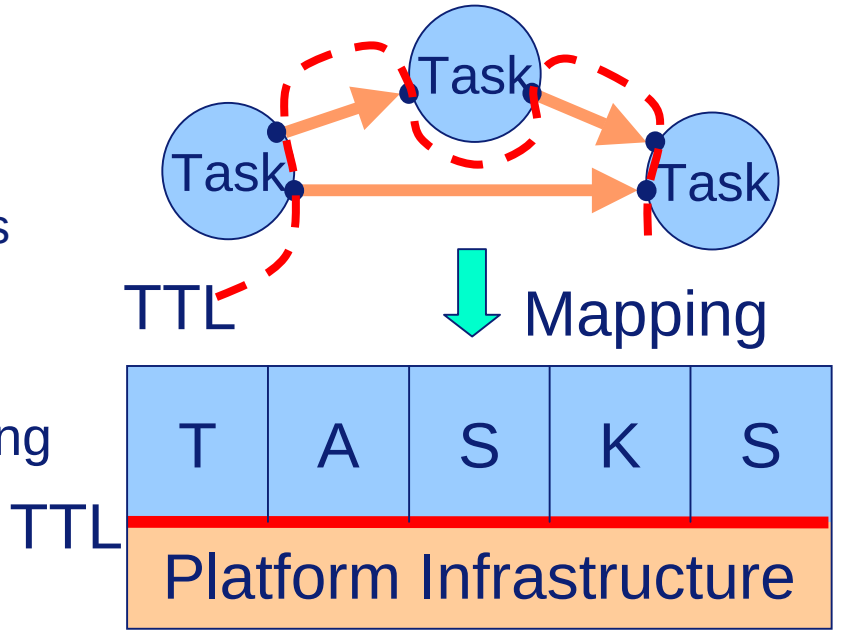
- Current practice
  - Ad hoc approaches
  - Low-level interfaces
- Examples
  - Synchronization via low-level primitives
    - Interrupts, MMIO, semaphores
  - Data access services partly in IP
    - Buffering, DMA control, address generation



- Consequence
  - Part of IP is specific for underlying communication infrastructure
    - IP just wants the next pixel or block or …
    - But also knows about burst transfers, interrupts, semaphores, ….

Interface Centric Design: TTL

- Aim: Improve MPSoC integration
- Means: Raise level of abstraction
- TTL Task Transaction Level interface:
  - Parallel application models
    - Executable specifications
  - Platform interface
    - Integration of HW and SW tasks
- Mapping technology
  - Structured design & programming

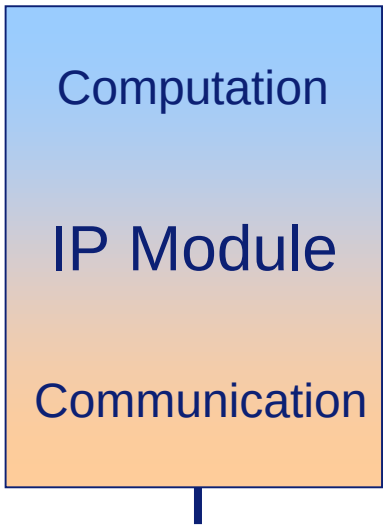


TTL Requirements

- Well-defined semantics for application modeling
  - Focus: stream processing applications
  - Make concurrency and communication explicit
- High-level interface
  - Make high-level services available
    - Inter-task communication
    - Multi-tasking
  - Easy to use for IP development
  - Facilitate reuse and integration of IP
  - Provide implementation freedom
- Allow efficient and cheap implementations
  - E.g. supporting fine grain synchronization for on-chip memory
- Support integration of hardware and software tasks

TTL Requirements

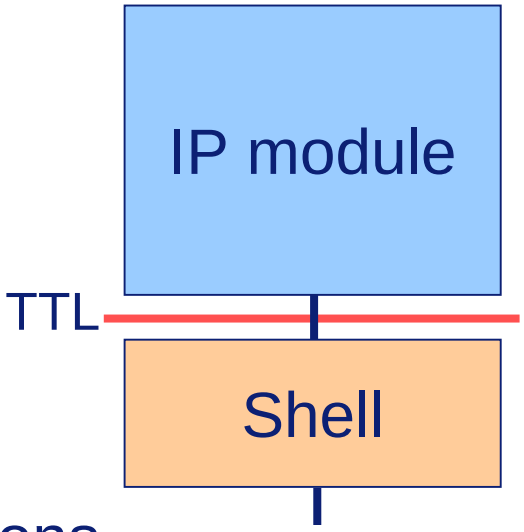
- Well-defined semantics for application modeling
  - Focus: stream processing applications
  - Make concurrency and communication explicit
- High-level interface
  - Make high-level services available
    - Inter-task communication
    - Multi-tasking
  - Easy to use for IP development
  - Facilitate reuse and integration of IP
  - Provide implementation freedom



- Allow efficient and cheap implementations
  - E.g. supporting fine grain synchronization for on-chip memory
- Support integration of hardware and software tasks

TTL Requirements

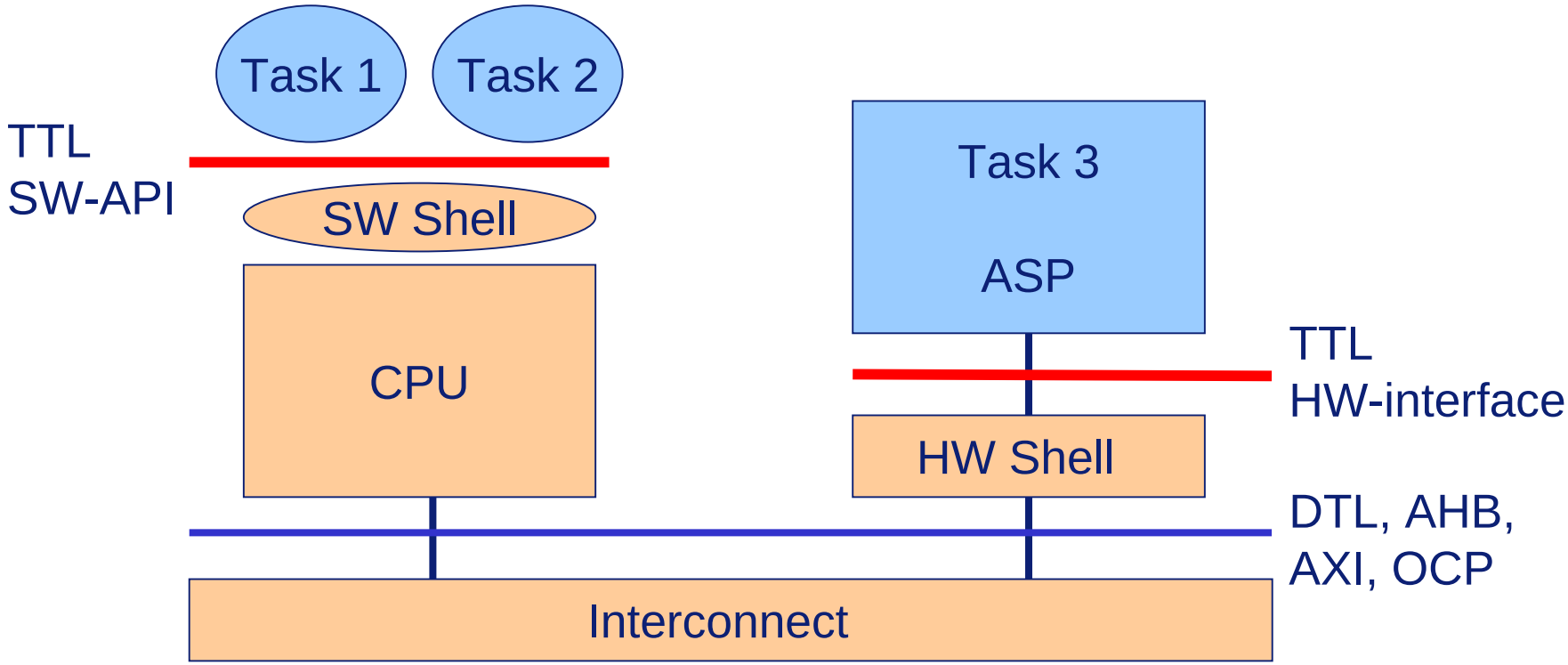
- Well-defined semantics for application modeling
  - Focus: stream processing applications
  - Make concurrency and communication explicit
- High-level interface
  - Make high-level services available
    - Inter-task communication
    - Multi-tasking
  - Easy to use for IP development
  - Facilitate reuse and integration of IP
  - Provide implementation freedom



- Allow efficient and cheap implementations
  - E.g. supporting fine grain synchronization for on-chip memory
- Support integration of hardware and software tasks

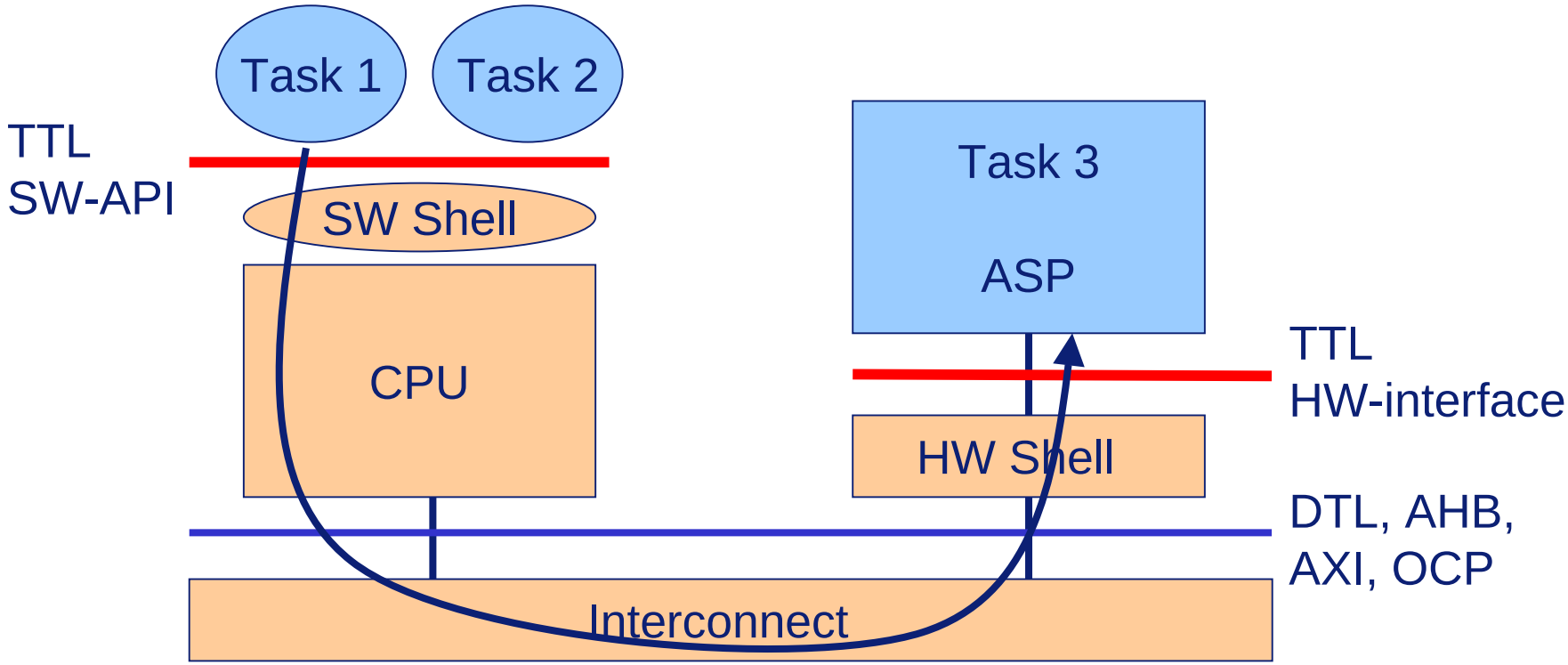
TTL in Example Architecture

- Platform interface for integration of HW and SW tasks
  - Enable communication in heterogeneous MPSoCs



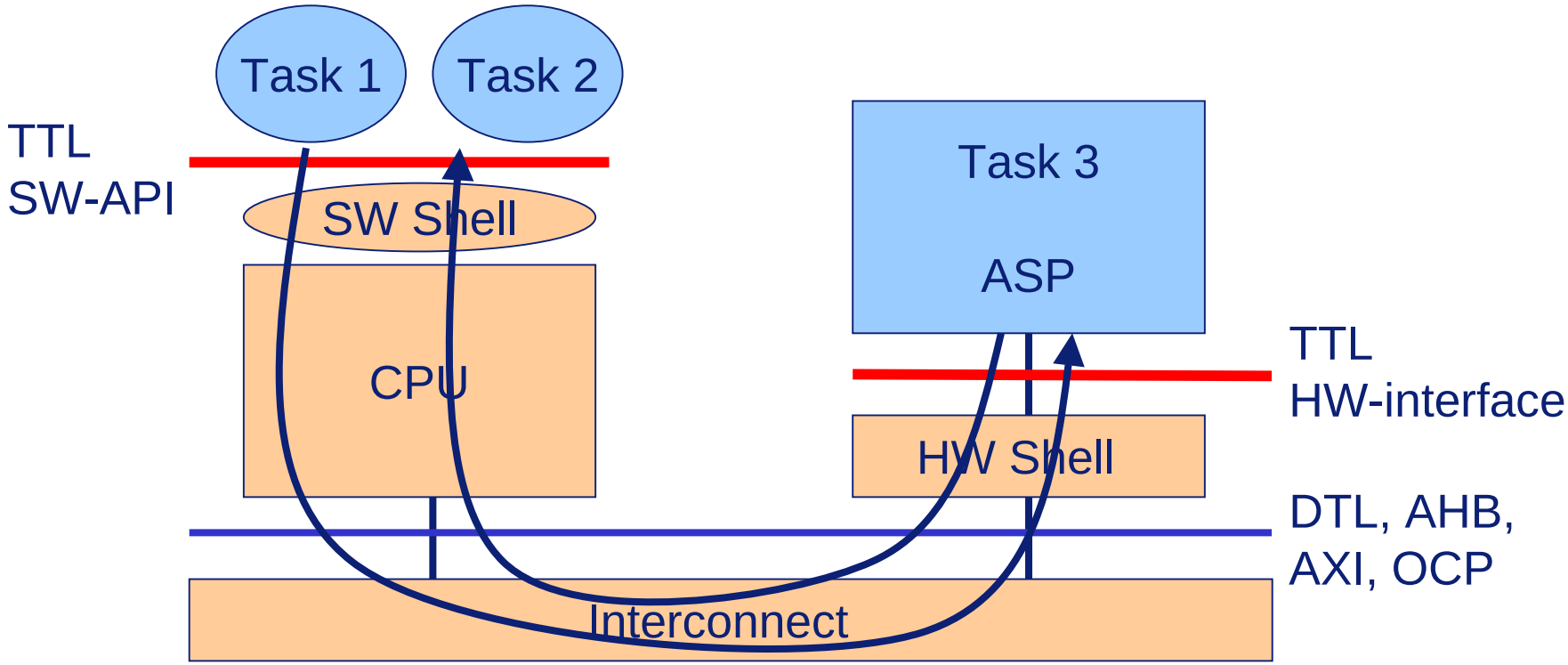
TTL in Example Architecture

- Platform interface for integration of HW and SW tasks
  - Enable communication in heterogeneous MPSoCs



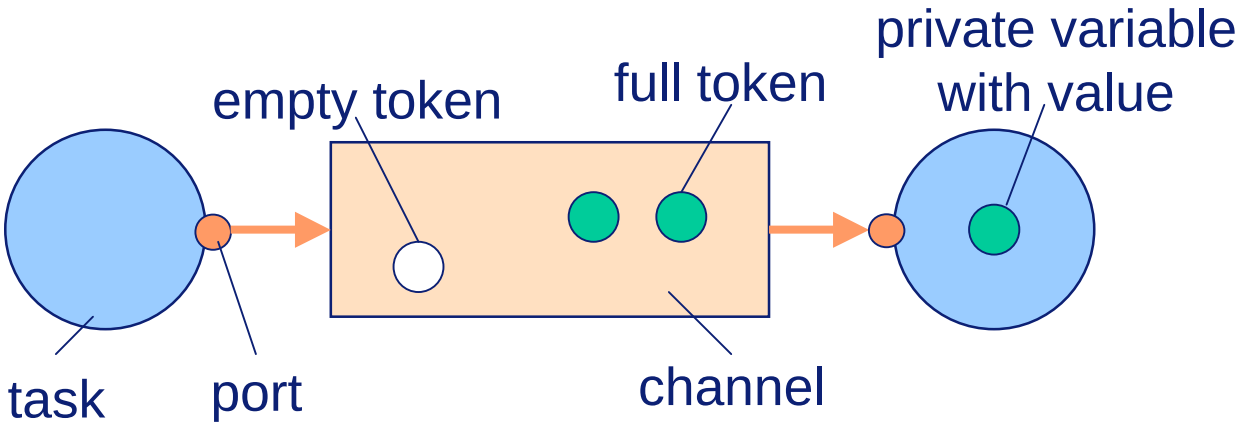
TTL in Example Architecture

- Platform interface for integration of HW and SW tasks
  - Enable communication in heterogeneous MPSoCs



TTL Inter-Task Communication

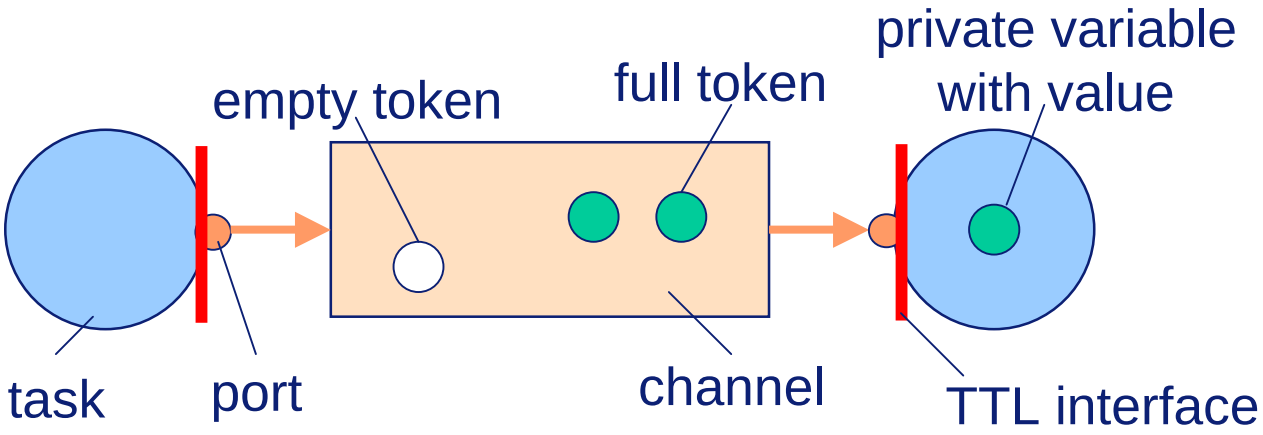
Logical model and terminology



- Communicating tasks are organized as task graph
- Tasks communicate by invoking TTL interface functions on their ports
- Uni-directional channels with reliable ordered communication
- Arbitrary data types, but single type per channel
- Support for multi-cast

TTL Inter-Task Communication

Logical model and terminology



- Communicating tasks are organized as task graph
- Tasks communicate by invoking TTL interface functions on their ports
- Uni-directional channels with reliable ordered communication
- Arbitrary data types, but single type per channel
- Support for multi-cast

Example: Message Passing Interface

Producer side

- write(port, data, …)
  - Write data into channel connected to port

Consumer side

- data = read(port, …)
  - Read data from channel connected to port

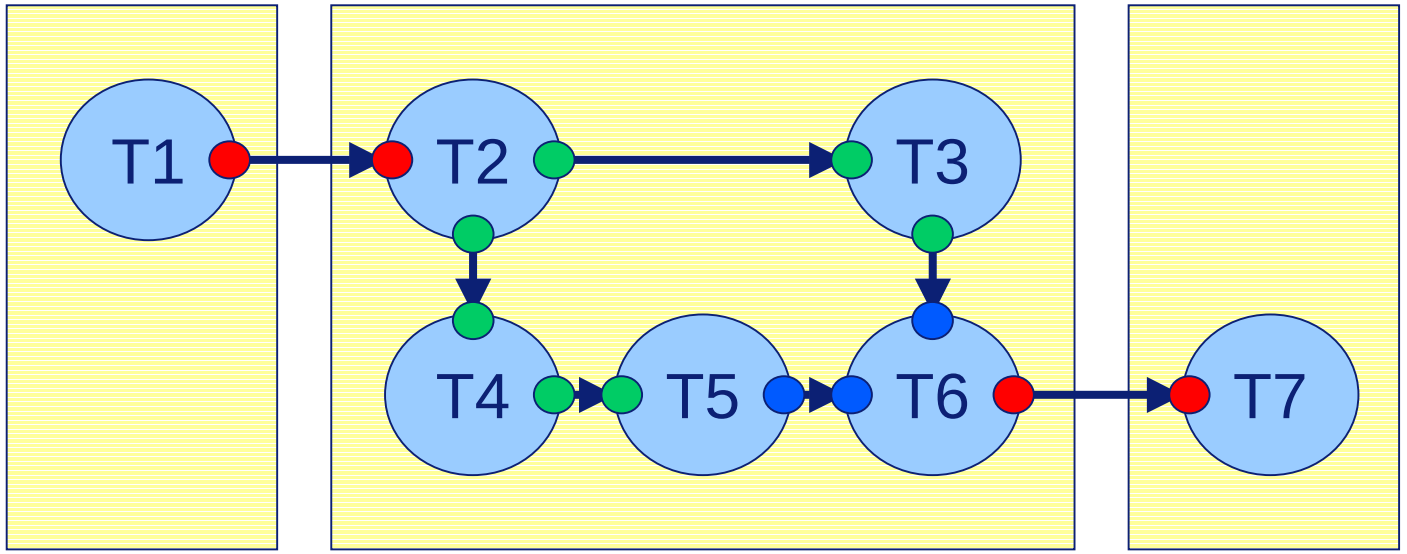
- Abstract interface for tasks
- Right interface ?
  - Appropriate for modeling application ?
  - Appropriate for implementation on architecture ?

TTL Interface Types

- Different needs for communication arising from:
  - Different applications
    - In-order – out-of-order
  - Different implementation styles
    - Hardware – software
    - Shared memory – message passing
- Support set of interface types
  - Each interface type offers narrow interface
    - Easy to use
    - Simple to implement
  - Each interface type supports particular communication style
  - Offer multiple interface types in one framework
  - **Based on single model for interoperability and design techn.**

TTL Interface Types

- TTL offers a number of different interface types
- Allow selection of interface type per port of task
- Enable interoperability by allowing mix & match



TTL Interface Types

Acronym	Full name
CB	Combined Blocking
RB	Relative Blocking
RN	Relative Non-blocking
DBI	Direct Blocking In-order
DNI	Direct Non-blocking In-order
DBO	Direct Blocking Out-of-order
DNO	Direct Non-blocking Out-of-order

Interface Type CB

Producer side

- write(port, vector, size)
  - Write vector of size values into channel

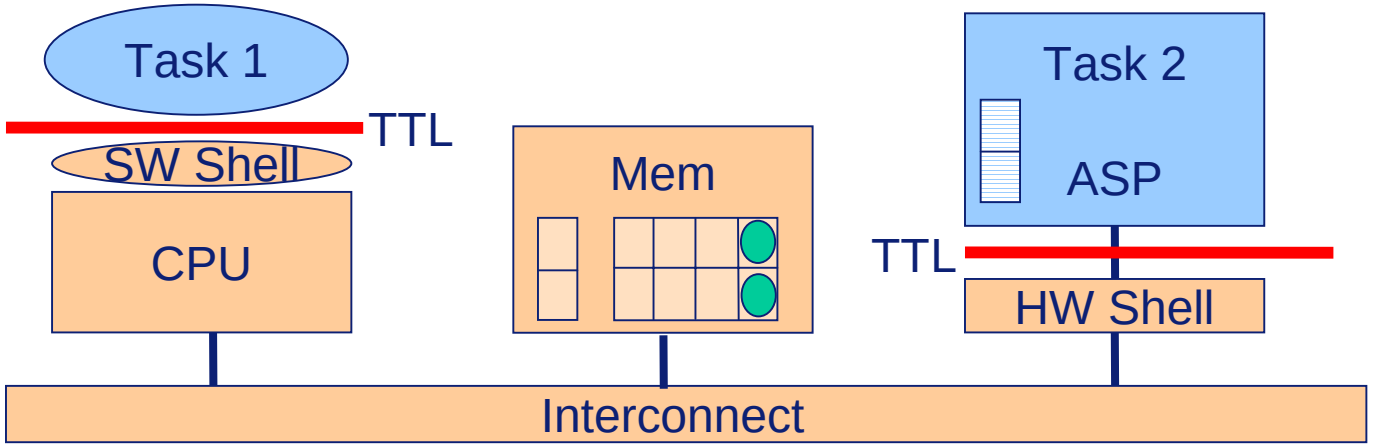
Consumer side

- read(port, vector, size)
  - Read vector of size values from channel

- Most abstract TTL interface type
- Blocking semantics
- Combined synchronization and data transfer
- Vector operations
- Based on earlier work on YAPI for KPN style modeling

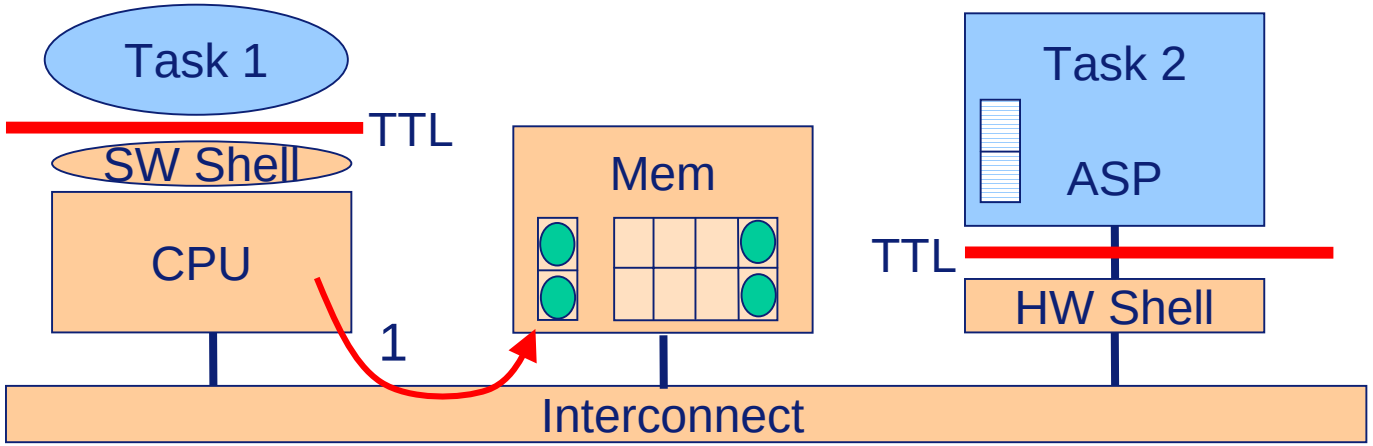
Pros / Cons Interface Type CB

+ Easy to use
+ Reusable tasks
– Copying overhead if private variables not in local buffers
  – Smart compiler may help in some cases
– If local buffers:
  – Large tokens / vectors → large local buffers
  – Small tokens / vectors → large synchronization overhead



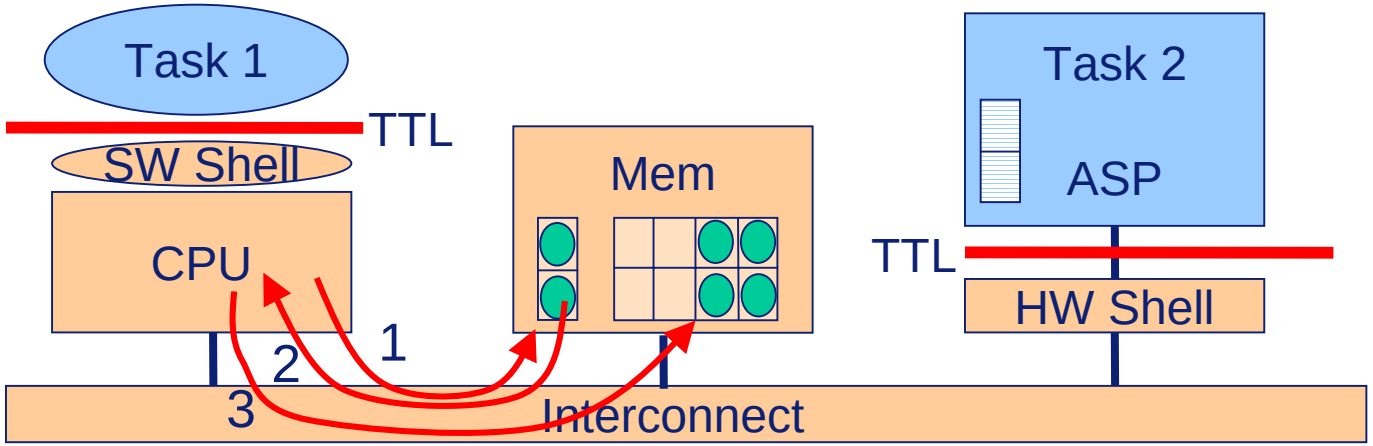
Pros / Cons Interface Type CB

+ Easy to use
+ Reusable tasks
– Copying overhead if private variables not in local buffers
  – Smart compiler may help in some cases
– If local buffers:
  – Large tokens / vectors → large local buffers
  – Small tokens / vectors → large synchronization overhead



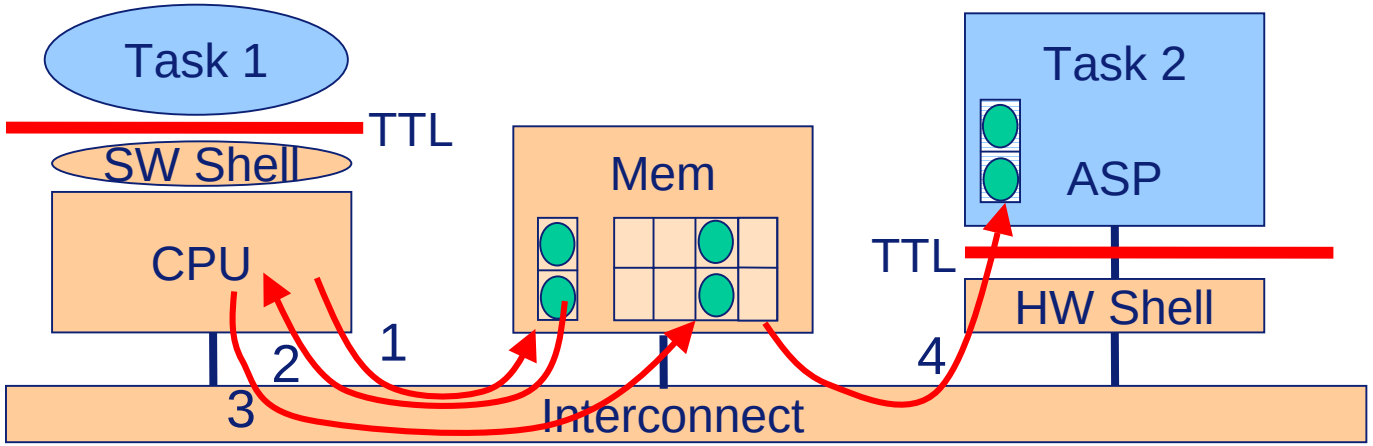
Pros / Cons Interface Type CB

+ Easy to use
+ Reusable tasks
– Copying overhead if private variables not in local buffers
  – Smart compiler may help in some cases
– If local buffers:
  – Large tokens / vectors → large local buffers
  – Small tokens / vectors → large synchronization overhead

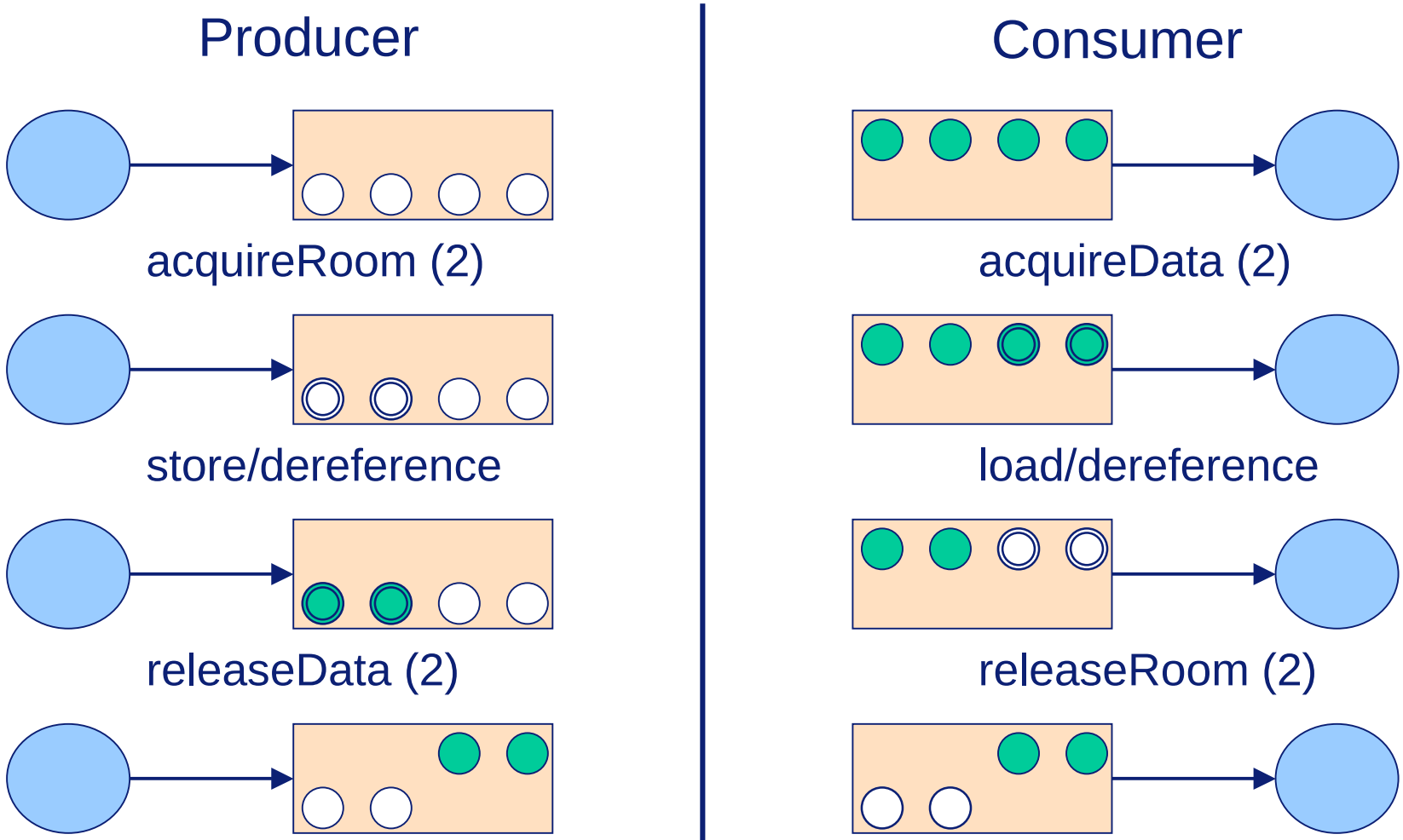


Pros / Cons Interface Type CB

+ Easy to use
+ Reusable tasks
– Copying overhead if private variables not in local buffers
  – Smart compiler may help in some cases
– If local buffers:
  – Large tokens / vectors → large local buffers
  – Small tokens / vectors → large synchronization overhead



Separate Synchronization and Data Transfer



Interface Types RB and RN

Producer side

- reAcquireRoom(port, count) (RB)
- tryReAcquireRoom(port, count) (RN)
  - Acquire count empty tokens, blocking (RB) / non-blocking (RN)
- store(port, offset, vector, size)
  - Store vector of size values into the tokens with offset..offset+size-1 to the oldest acquired token
- releaseData(port, count)
  - Release count oldest acquired tokens as full tokens

- Separate synchronization and data transfer
- Vector operations
- Re-acquire operations do not change state of the channel

Pros / Cons Interface Types RB / RN

+ Coarse grain synchronization with fine grain data transfer
  – Low synchronization overhead with small local buffers
+ Out-of-order data accesses
  – Reduce cost of private variables
+ Load only subset of tokens from channel
  – Reduce cost of data transfers

– Less abstract than CB
  – Increases programming effort
  – Makes tasks less reusable
– Inefficiencies upon data transfers
  – Function call, access to channel admin, address calculations
– Copying may still occur

Interface Types DBI and DNI

Producer side

- acquireRoom(port, &token) (DBI)
- tryAcquireRoom(port, &token) (DNI)
  - Acquire empty token, blocking (DBI) / non-blocking (DNI)
- token->field = value;
  - Assign value to (part of) token
- releaseData(port)
  - Release oldest acquired token as full token

- Separate synchronization and data transfer
- Direct access to data via token references (pointers)
- Scalar operations only
- Tokens are released in same order as they are acquired

Pros / Cons Interface Types DBI / DNI

+ Coarse grain synchronization with fine grain data transfer
+ Out-of-order data accesses for acquired token(s)
+ Load only part of token from channel
+ Direct data accesses
  - Efficient data transfers

– Less abstract than CB / RB / RN
  - Exposes memory addresses
  - Makes tasks less reusable
– No vector operations
  - Would complicate interface / expose channel implementation

Interface Types DBO and DNO

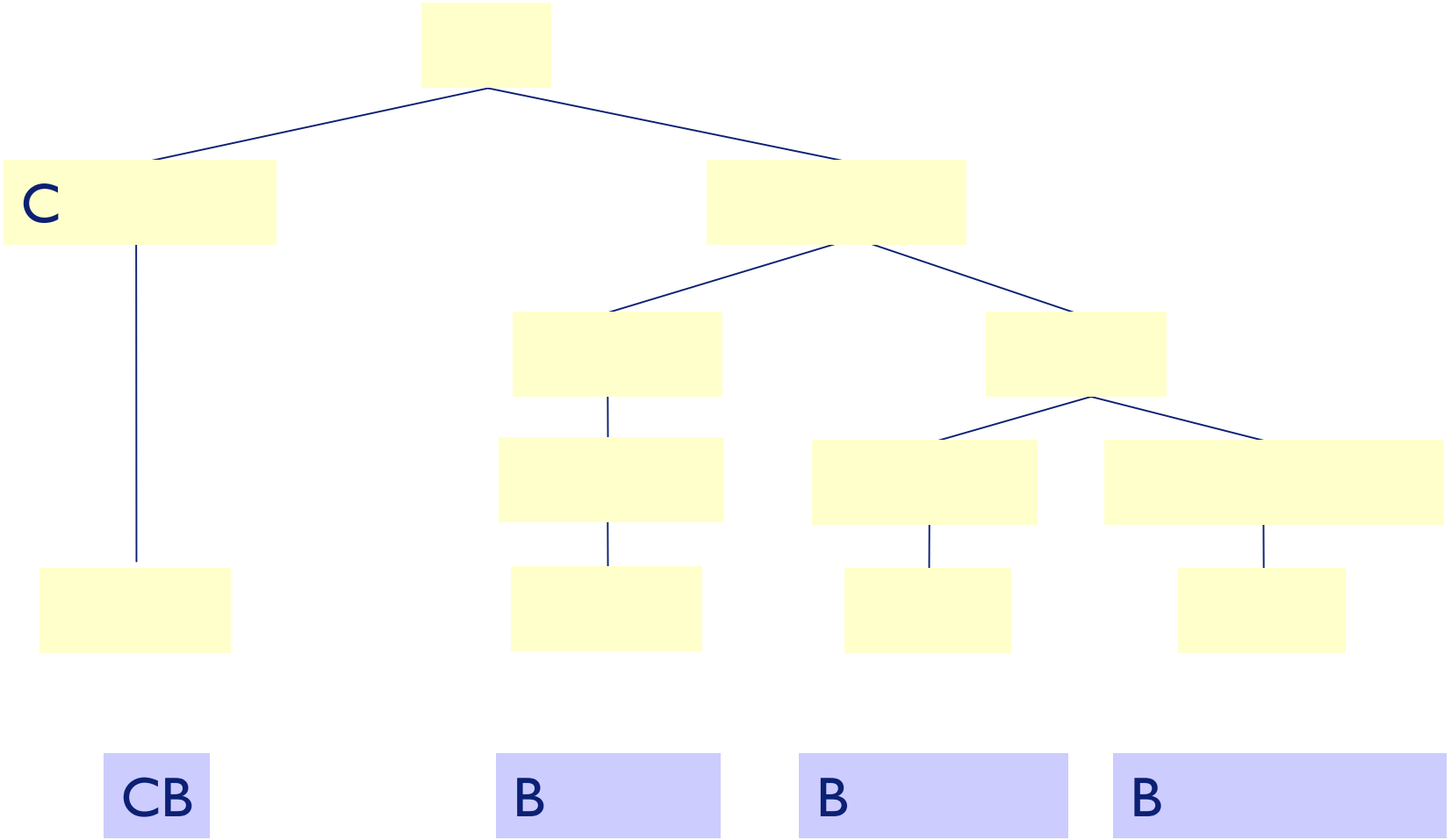
Producer side

- acquireRoom(port, &token) (DBO)
- tryAcquireRoom(port, &token) (DNO)
  - Acquire empty token, blocking (DBO) / non-blocking (DNO)
- token->field = value;
  - Assign value to (part of) token
- releaseData(port, &token)
  - Release token as the next full token

\+ Out-of-order release supports efficient use of memory

– More complex implementation of the channel

TTL Interface Types



TTL Multi-Tasking Interface

TTL offers three task types:

1. Process
   - Own thread of execution
   - No explicit interaction with scheduler
   - Implicit task switching and state saving
2. Co-routine
   - Explicit interaction with scheduler via **suspend()** function
   - Implicit state saving
3. Actor
   - Fire-exit tasks that return to scheduler
   - State saving to be performed by task

TTL APIs and Implementations

- TTL interface is available as:
  - C++ API
  - C API
  - Hardware interface
- Generic run-time environment
  - Functional modeling and verification of TTL application models in C++ / C
- Platform implementations
  - Sea-of-DSP
  - Smart Camera
  - Cake / Wasabi

Outline

- Introduction
- Task Transaction Level interface: TTL
  - Abstract interface for streaming in MPSoCs
- Programming TTL multiprocessors
  - Constraint-driven code transformations
- Design cases
  - Sea-of-DSP
  - Smart Camera
  - Cake / Wasabi
- Conclusion

Problem

How to efficiently program applications on platforms using the TTL interface?

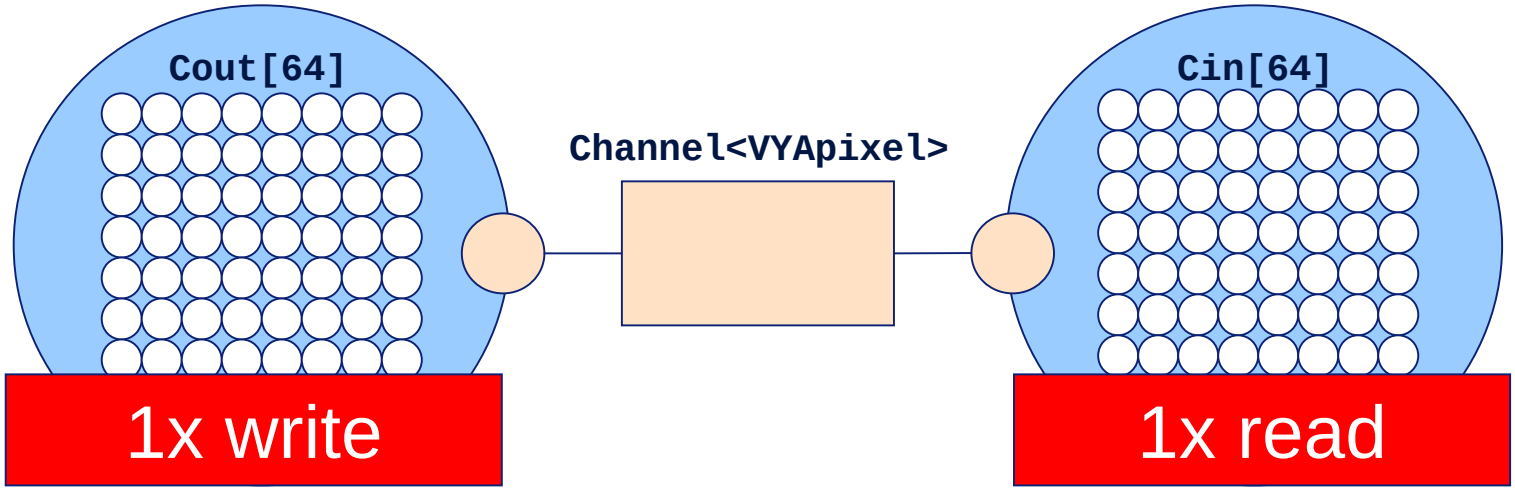
- Efficient = cost + performance + effort

- The cost and performance of TTL interface functions varies on different platforms
- The cost and performance of different TTL interface types varies on one platform

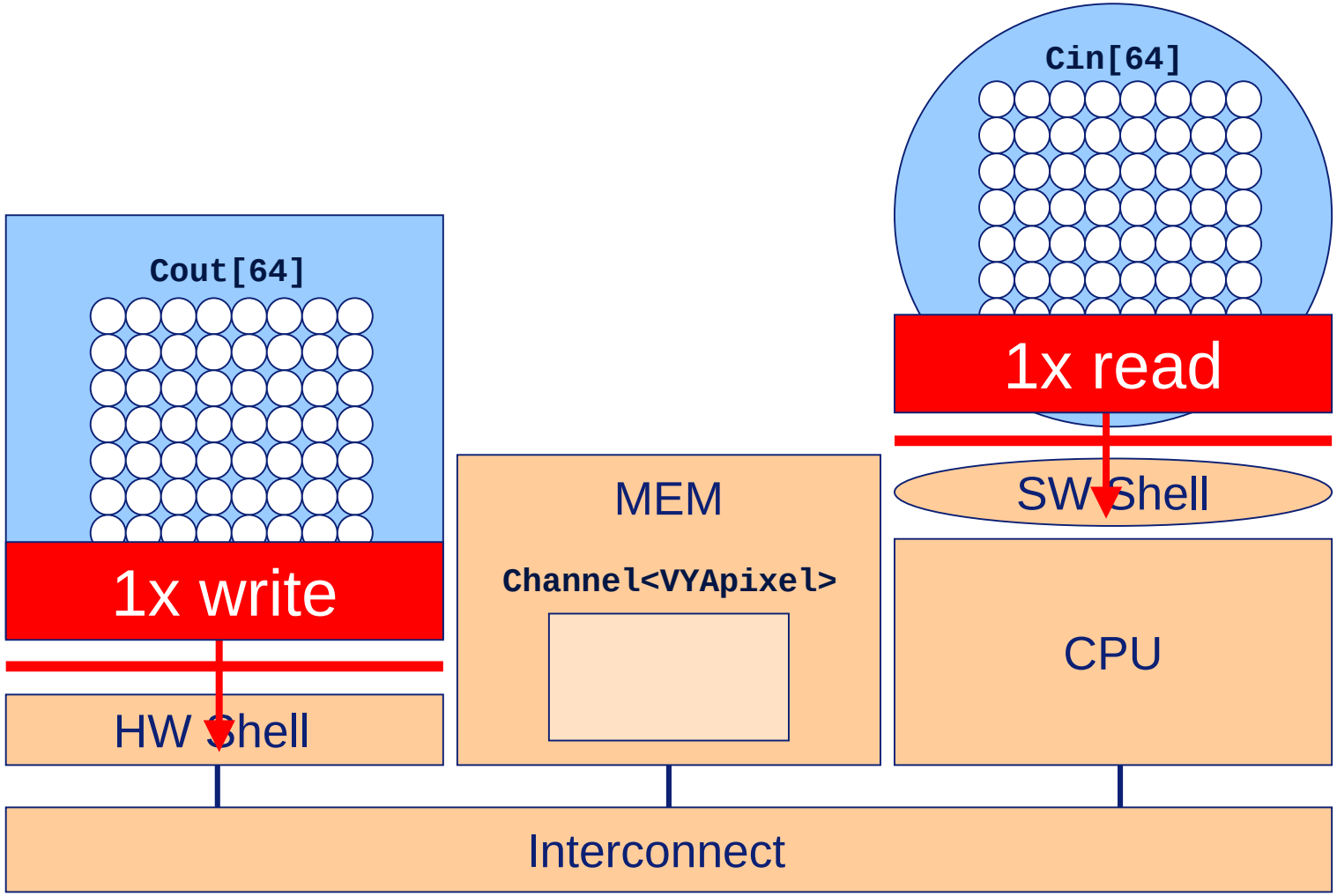
Example IQ→IZZ Using CB

```
01 void IQ::main()
02   while (true)
03     for(int j=0; j<vi; j++)
04       for(int k=0; k<hi; k++)
05         VYApixel Cout[64];
06         for(int l=0; l<64; l++)
07           VYApixel Cin;
08           read(CinP, Cin);
09           Cout[l] = QT[t][l]*Cin;
10         write(CoutP, Cout, 64);
```

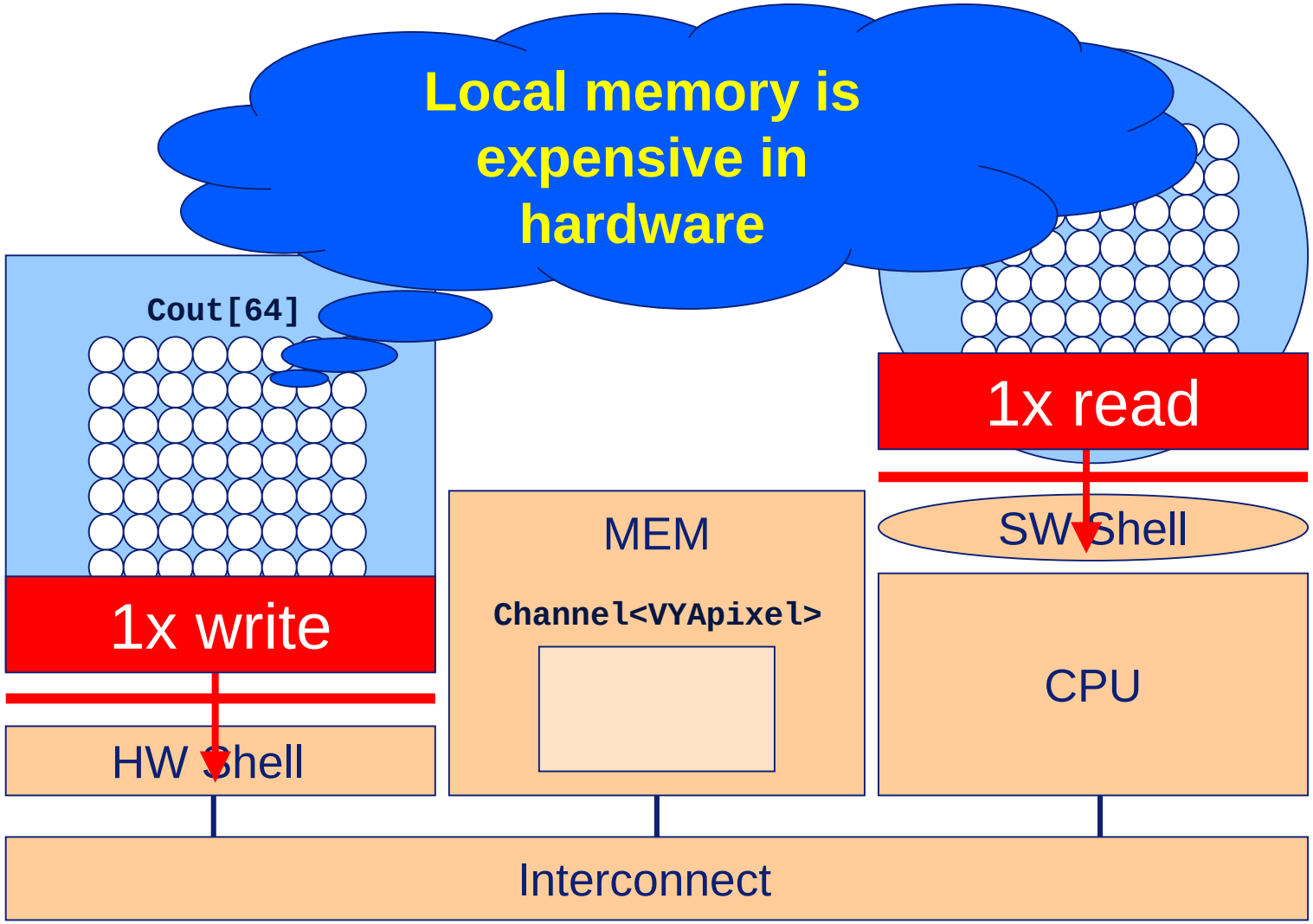
```
01 void IZZ::main()
02   while (true)
03     VYApixel Cin[64];
04     VYApixel Cout[64];
05     read(CinP, Cin, 64);
06     for(int i=0; i<64; i++)
07       Cout[zigzag[i]] = Cin[i];
08     write(CoutP, Cout, 64);
```



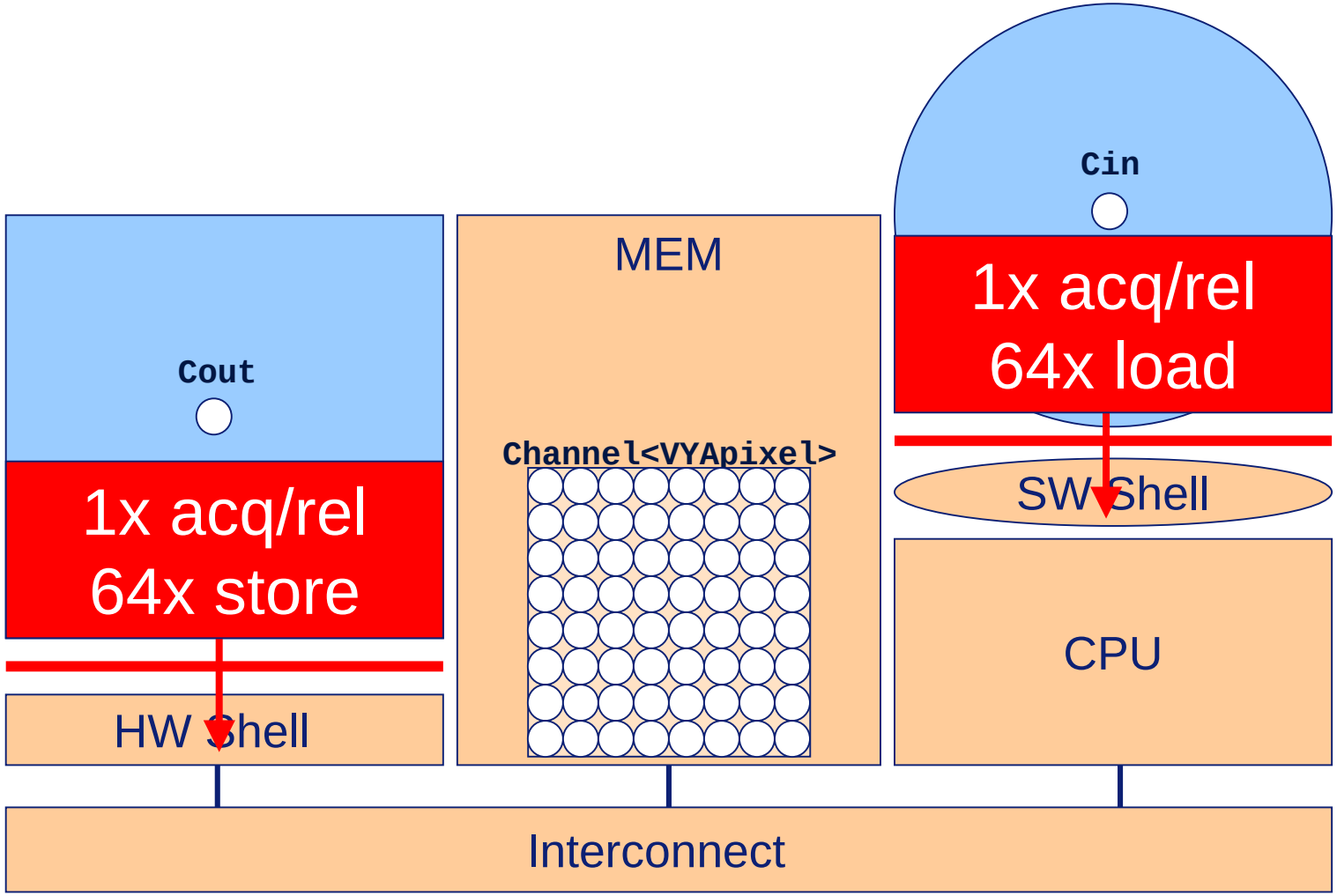
Efficiency of IQ→IZZ Using CB (HW)



Efficiency of IQ→IZZ Using CB (HW)



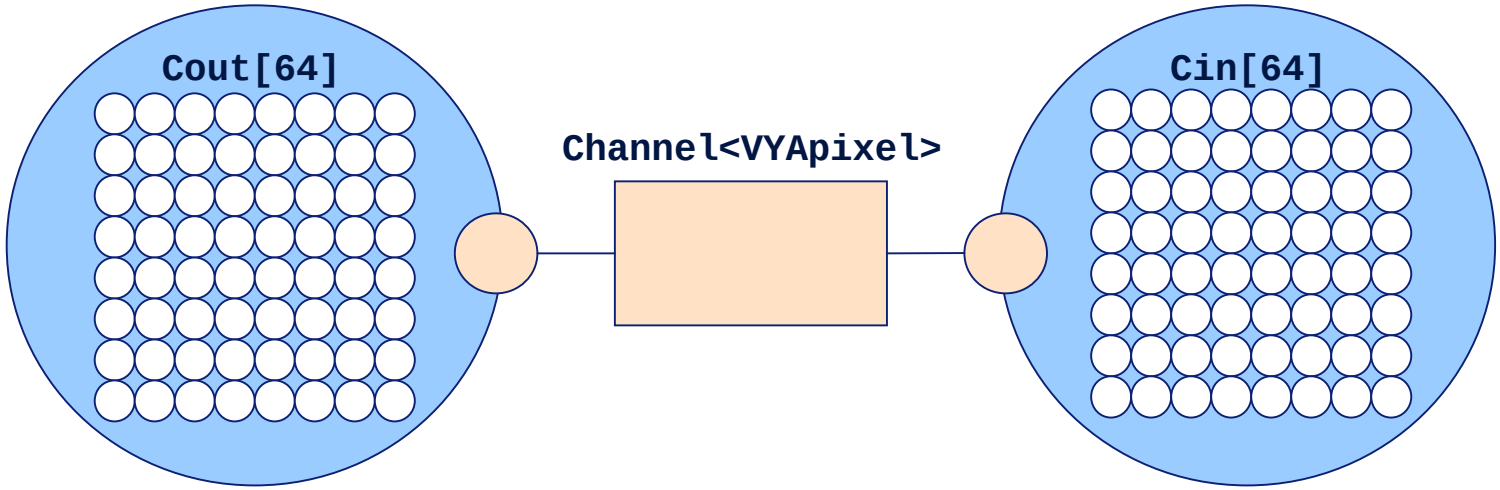
Transform IQ→IZZ Using RB (1)



Transform IQ→IZZ Using RB (2)

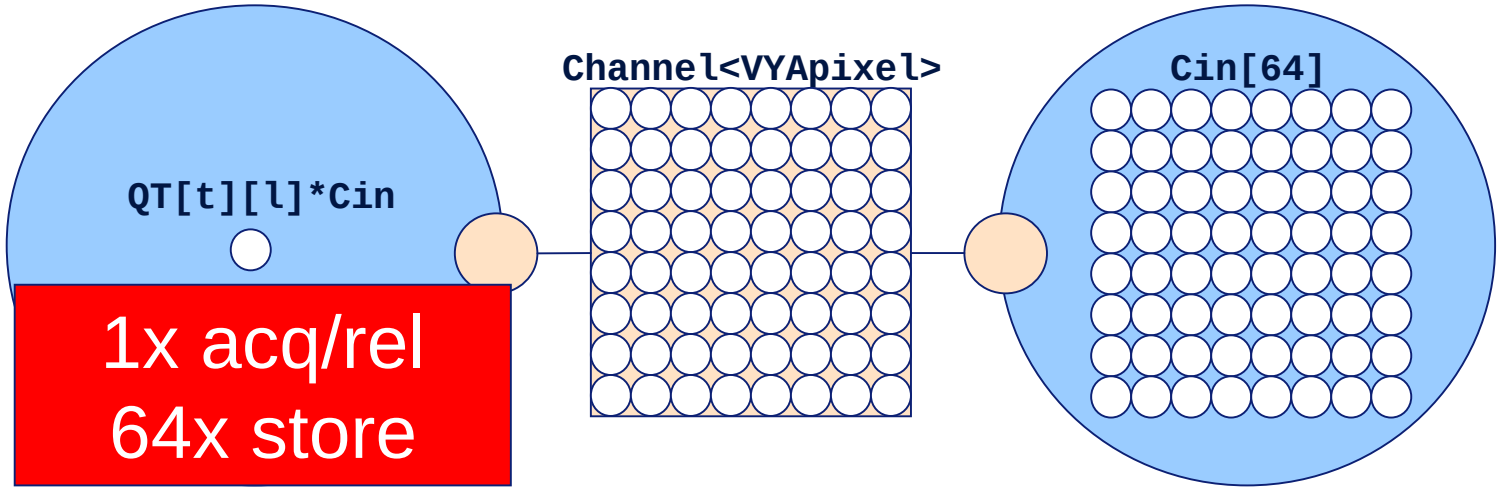
```
01 void IQ::main()
02   while (true)
03     for(int j=0; j<vi; j++)
04       for(int k=0; k<hi; k++)
05         VYApixel Cout[64];
06         for(int l=0; l<64; l++)
07           VYApixel Cin;
08           read(CinP, Cin);
09           Cout[l] = QT[t][l]*Cin;
10         write(CoutP, Cout, 64);
```

- remove declaration
- acquire 64 tokens
- add store operation
- release 64 tokens



Transform IQ→IZZ Using RB (3)

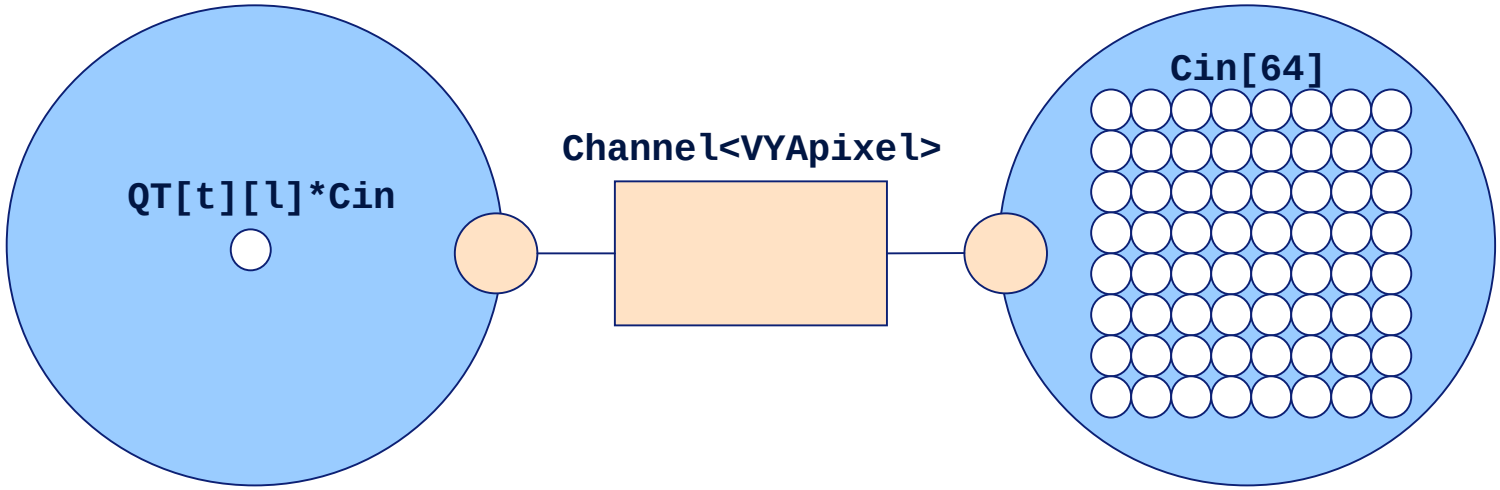
```
01 void IQ::main()
02  while (true)
03   for(int j=0; j<vi; j++)
04    for(int k=0; k<hi; k++)
05     reAcquireRoom(CoutP, 64);
06     for(int l=0; l<64; l++)
07      VYApixel Cin;
08      read(CinP, Cin);
09      store(CoutP, l, QT[t][l]*Cin);
10     releaseData(CoutP, 64);
```



Transform IQ→IZZ Using RB (4)

- remove declaration
- acquire 64 tokens
- load value of **`Cin[i]`**
- release 64 tokens

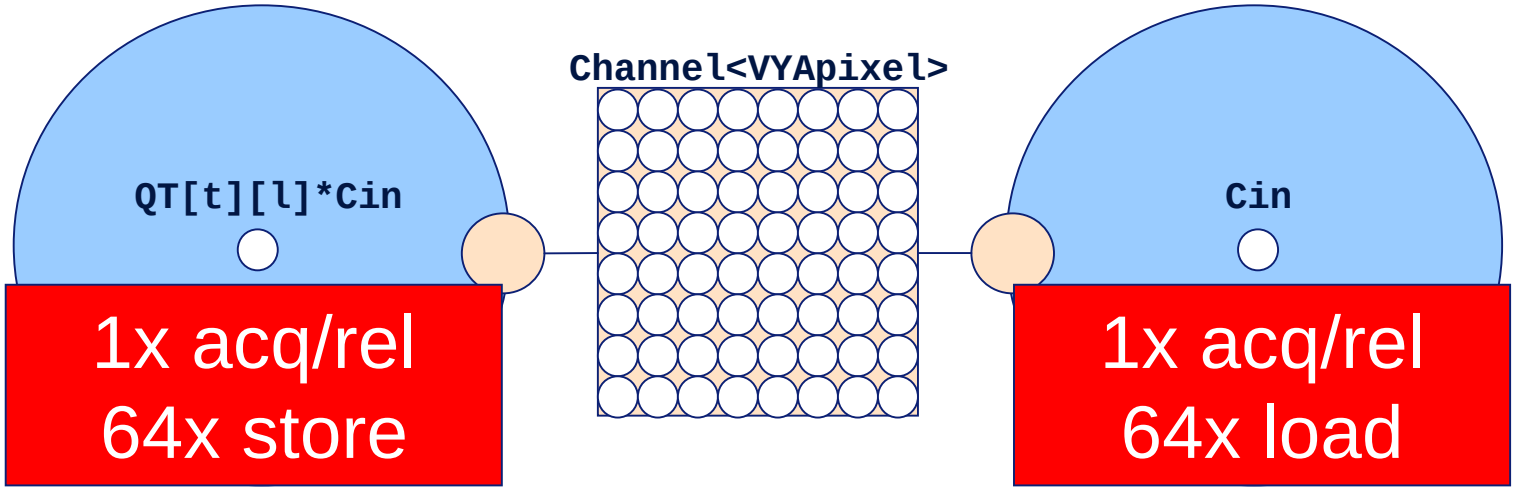
```
01 void IZZ::main()
02   while (true)
03     VYApixel Cin[64];
04     VYApixel Cout[64];
05     read(CinP, Cin, 64);
06     for(int i=0; i<64; i++)
07       Cout[zigzag[i]] = Cin[i];
08     write(CoutP, Cout, 64);
```



Transform IQ→IZZ Using RB (5)

```
01 void IQ::main()
02  while (true)
03   for(int j=0; j<vi; j++)
04    for(int k=0; k<hi; k++)
05     reAcquireRoom(CoutP, 64);
06     for(int l=0; l<64; l++)
07      VYApixel Cin;
08      read(CinP, Cin);
09      store(CoutP, l, QT[t][l]*Cin);
10     releaseData(CoutP, 64);
```

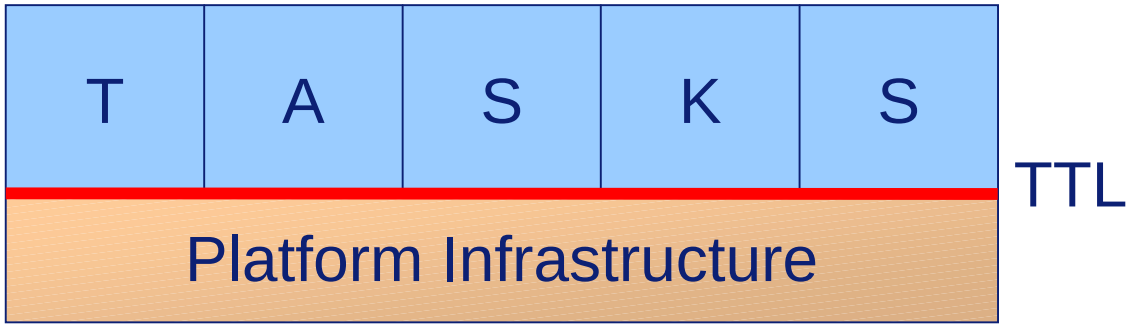
```
01 void IZZ::main()
02   while (true)
03     VYApixel Cout[64];
04     reAcquireData(CinP, 64);
05     for(int i=0; i<64; i++)
06       VYApixel Cin;
07       load(CinP, i, Cin);
08       Cout[zigzag[i]] = Cin;
09     write(CoutP, Cout, 64);
10     releaseRoom(CinP, 64);
```



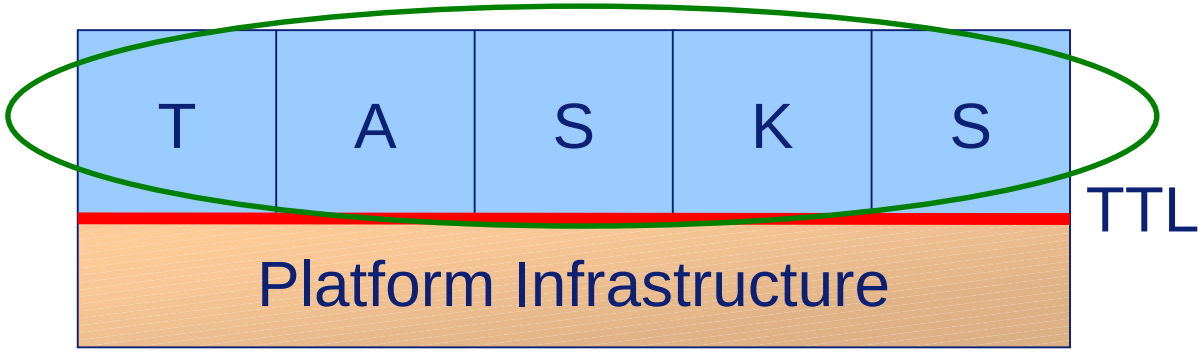
Outline

- Introduction
- Task Transaction Level interface: TTL
  - Abstract interface for streaming in MPSoCs
- Programming TTL multiprocessors
  - Constraint-driven code transformations
- Design cases
  - Sea-of-DSP
  - Smart Camera
  - Cake / Wasabi
- Conclusion

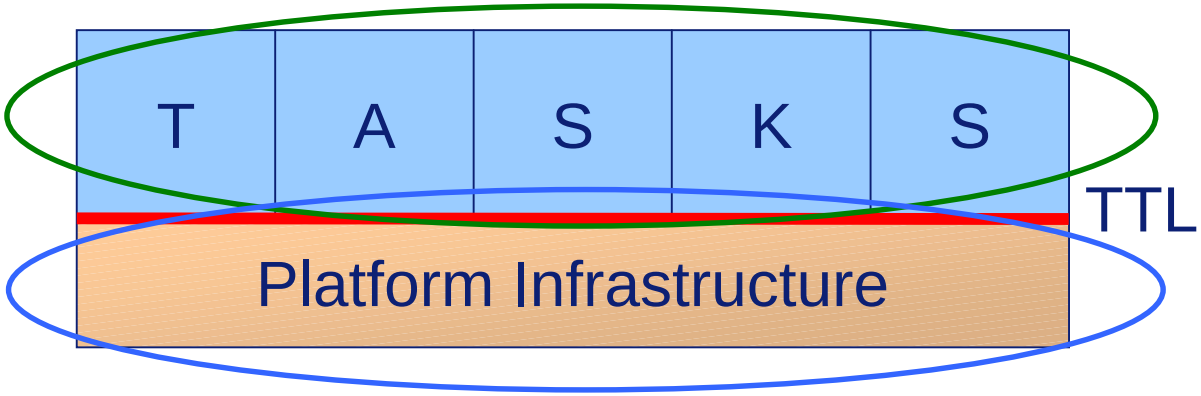
Implementation of TTL



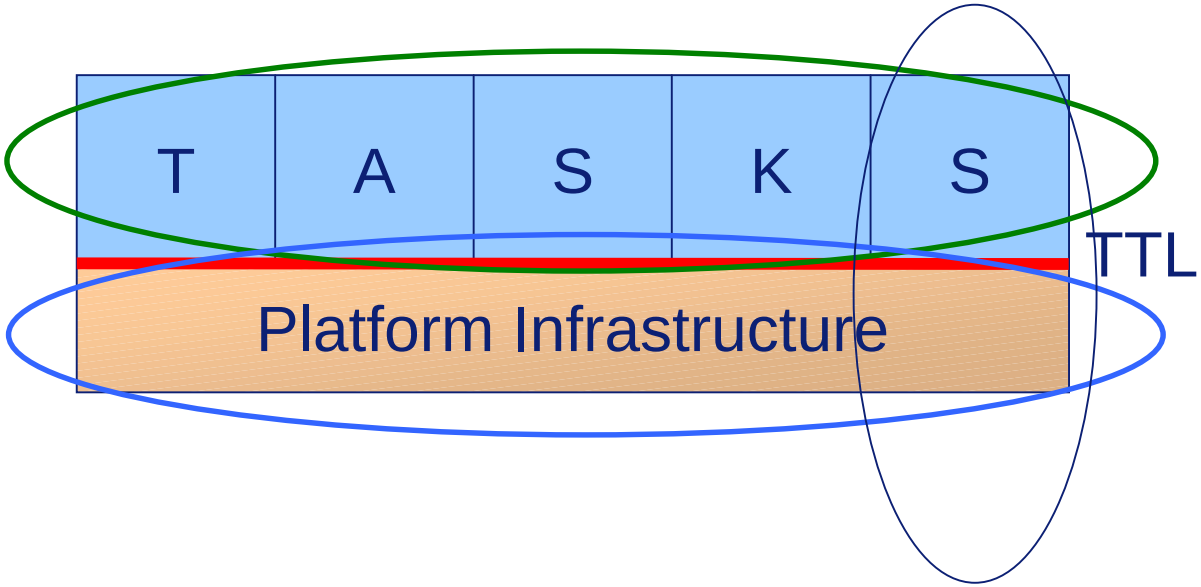
Implementation of TTL

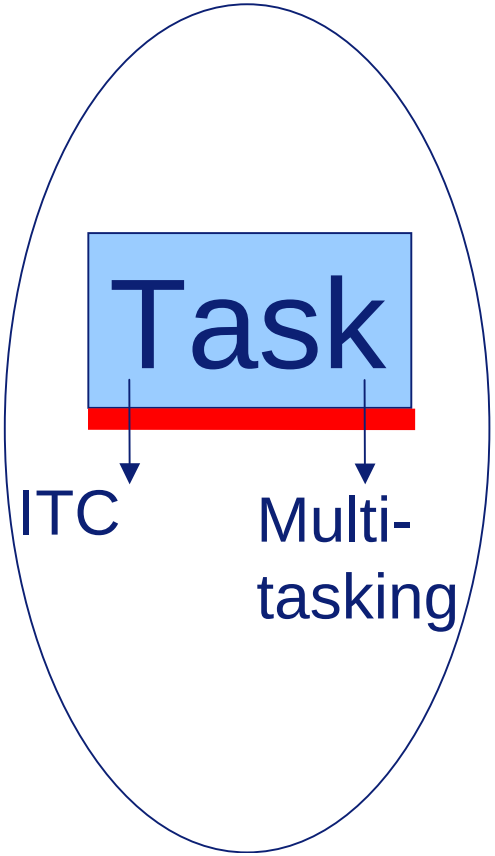


Implementation of TTL

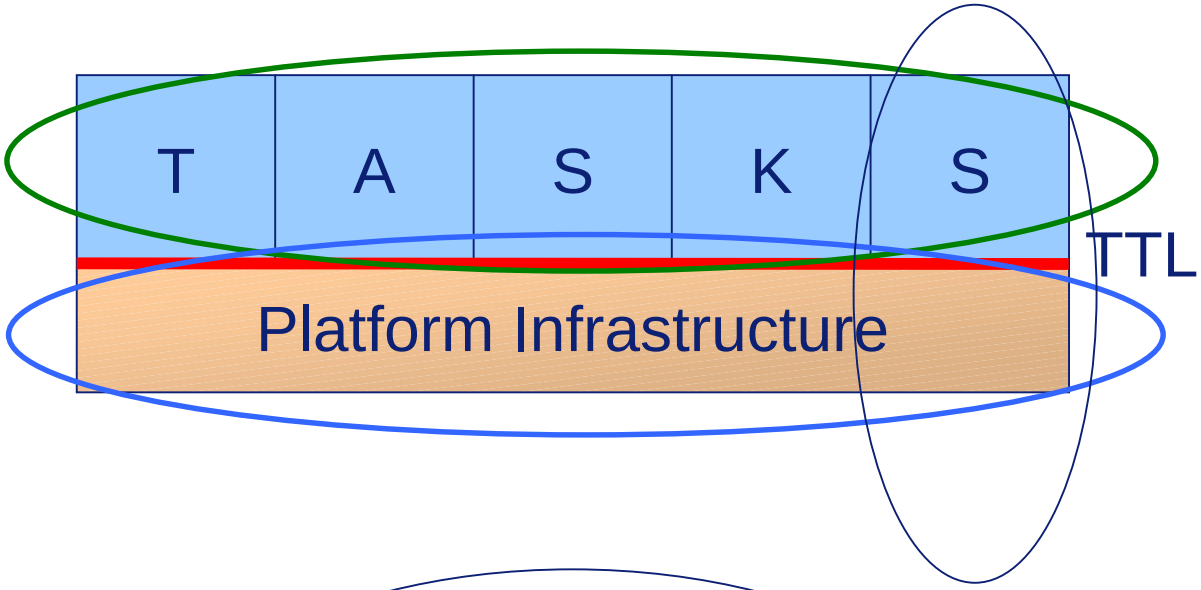


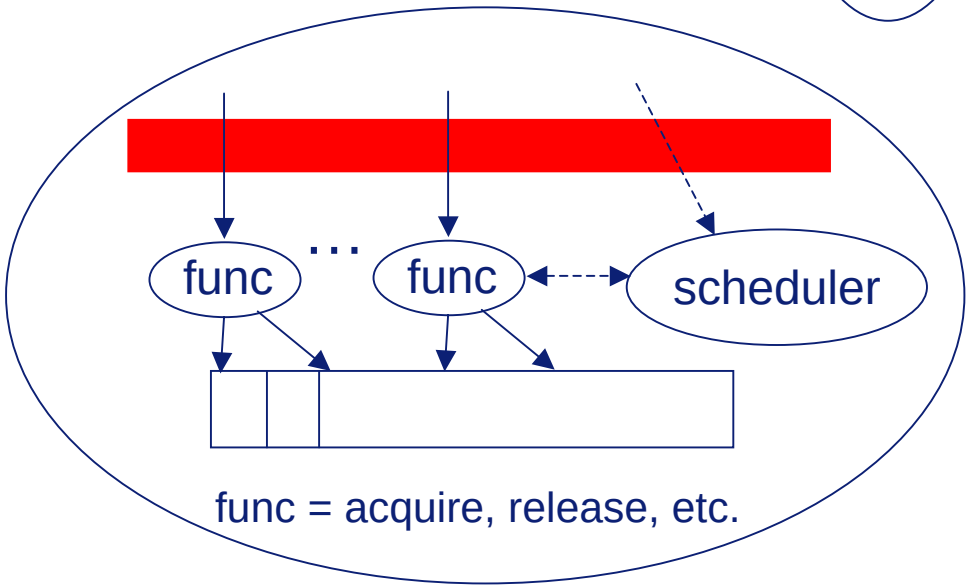
Implementation of TTL

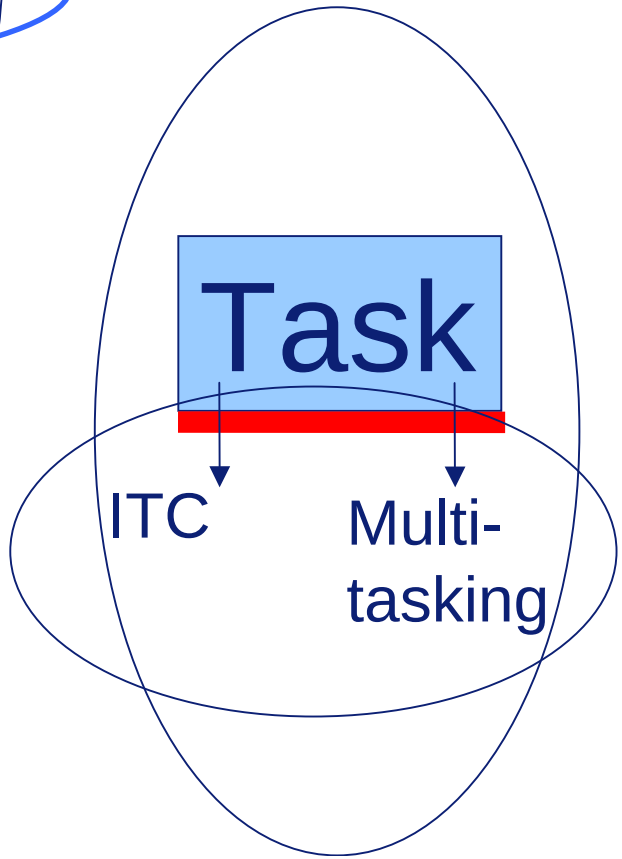




Implementation of TTL

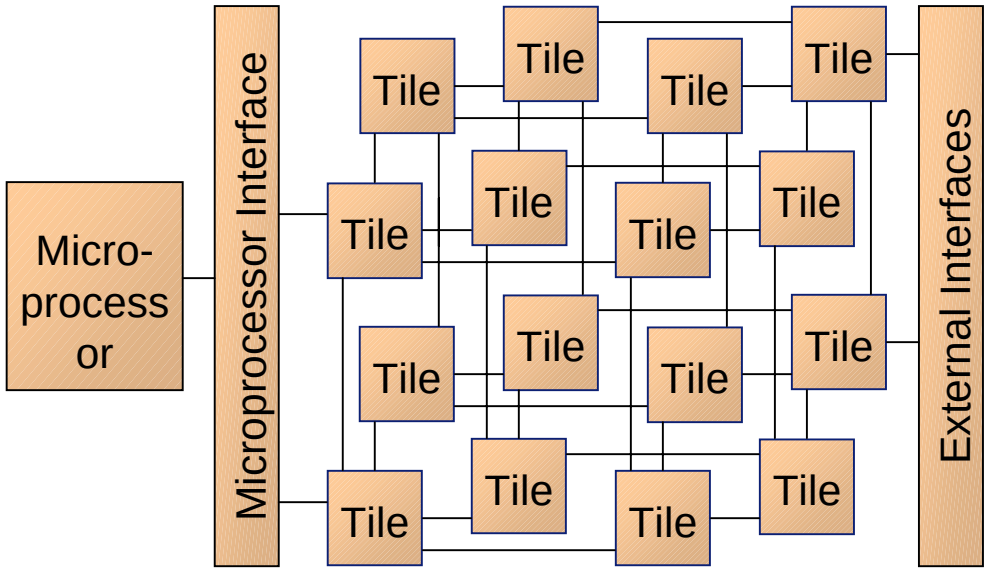




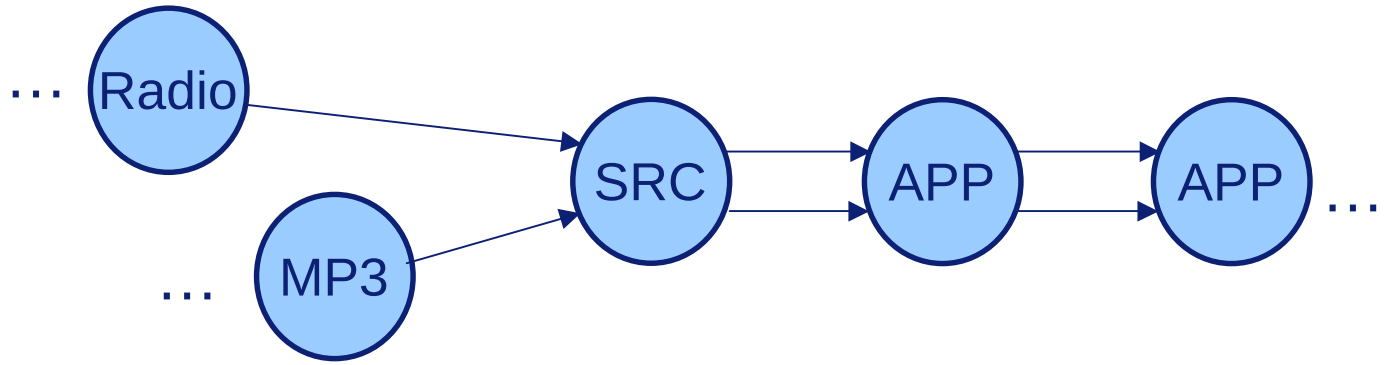


Sea of DSP Architecture

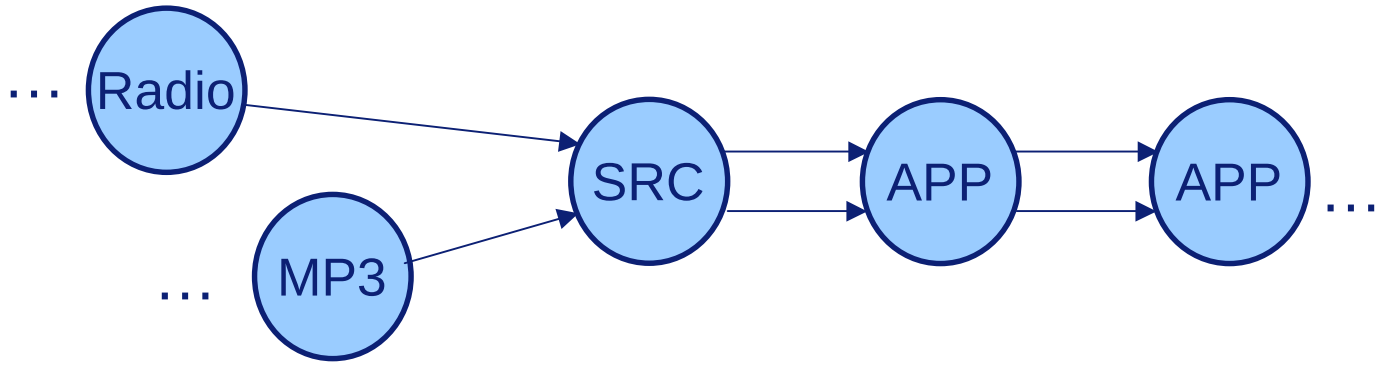
- Scalable and power-efficient
- Tile = DSP + Memory + DMA + inter-tile communication
- Any number of tiles is possible
- Memory mapped write-only inter-tile communication
- No general shared memory
- No OS on tiles

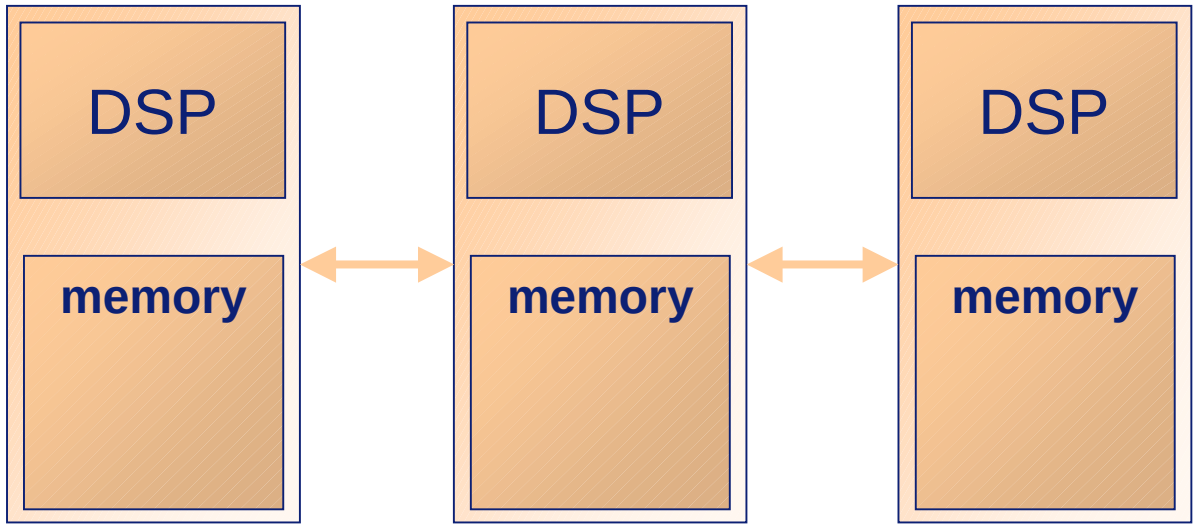


Mapping on Sea of DSP

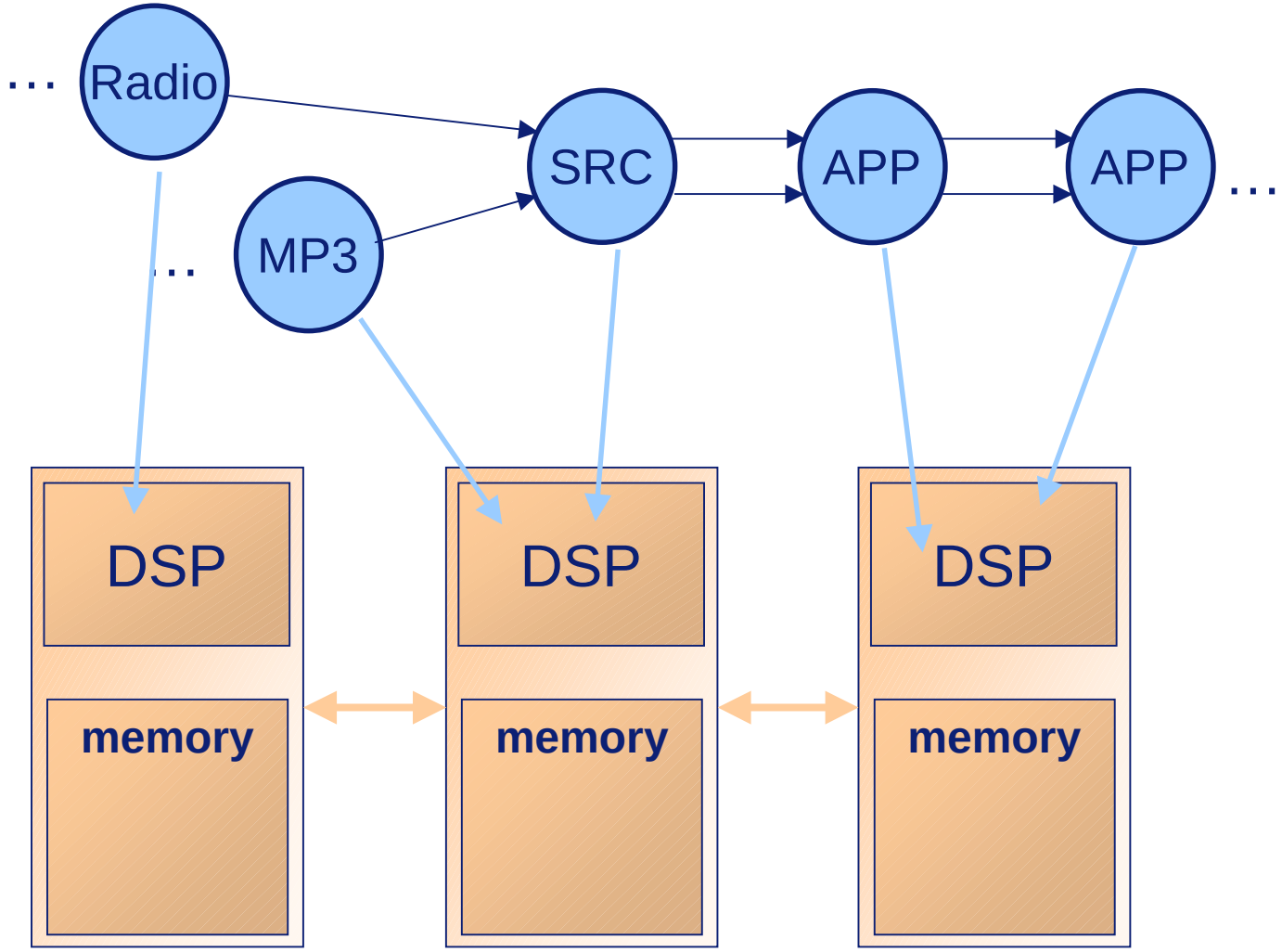


Mapping on Sea of DSP

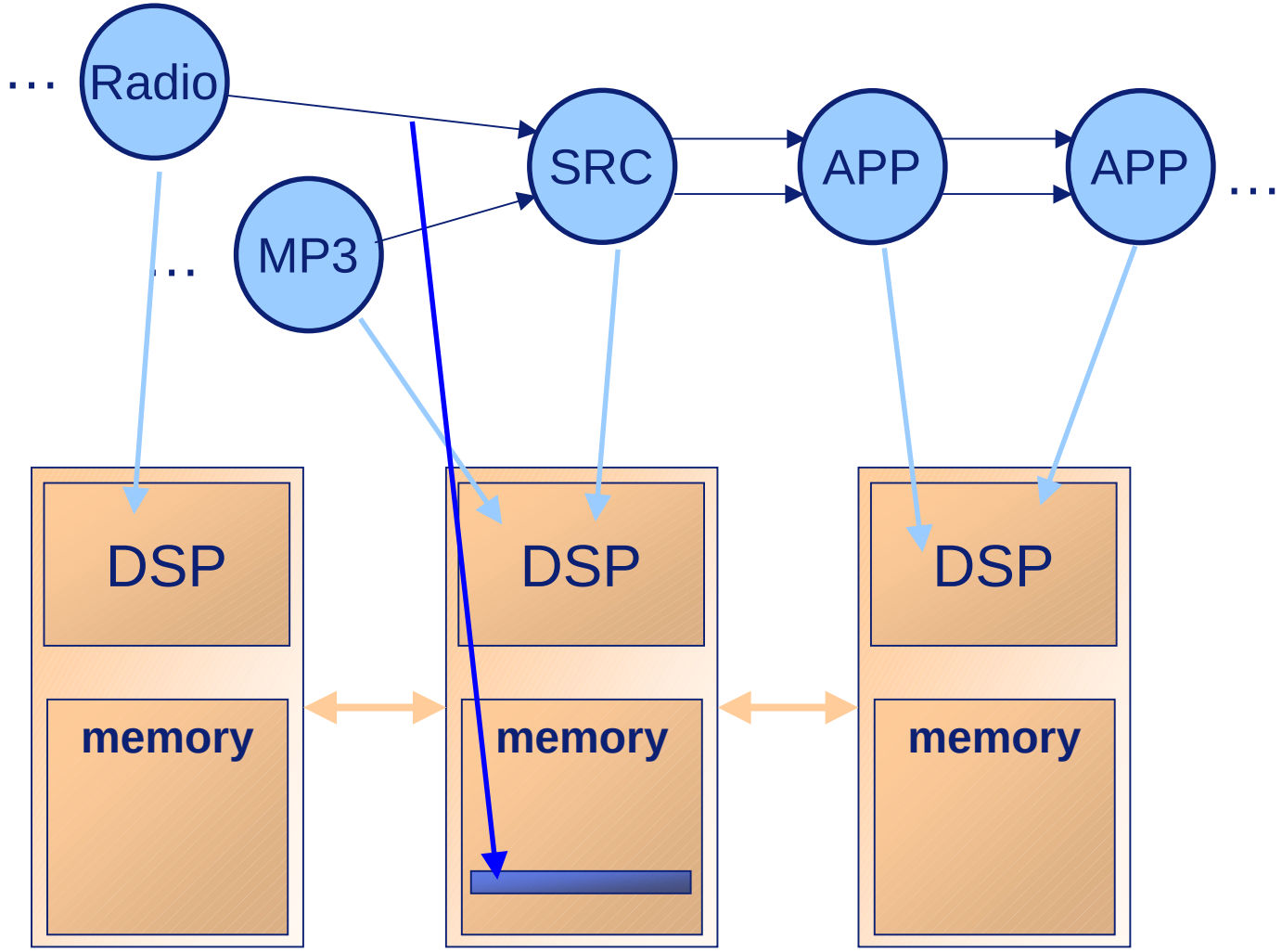




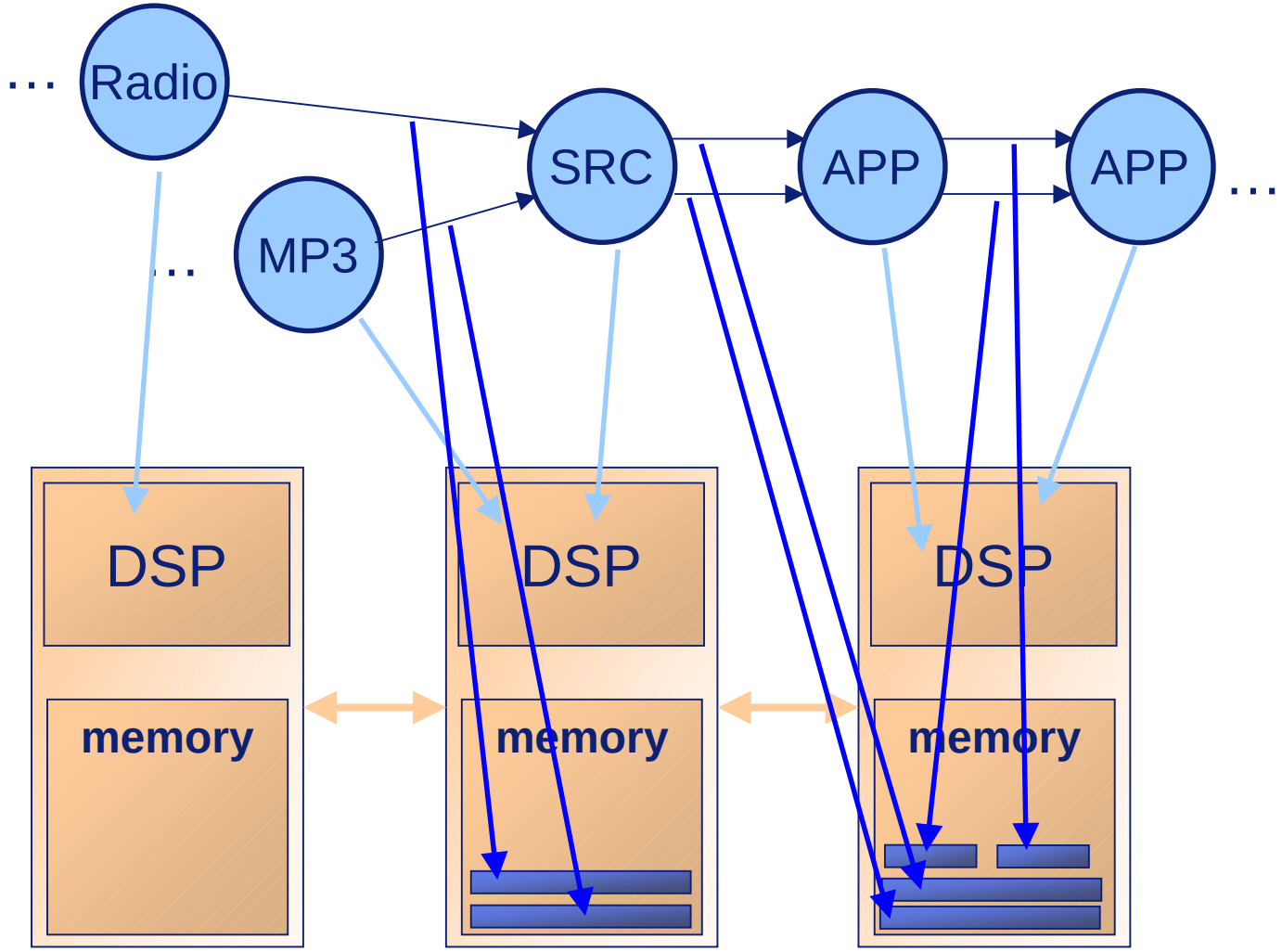
Mapping on Sea of DSP



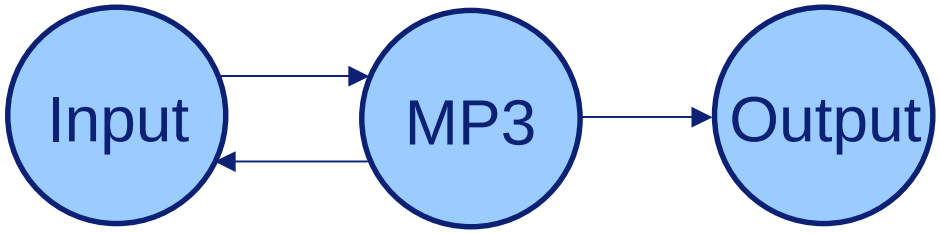
Mapping on Sea of DSP



Mapping on Sea of DSP

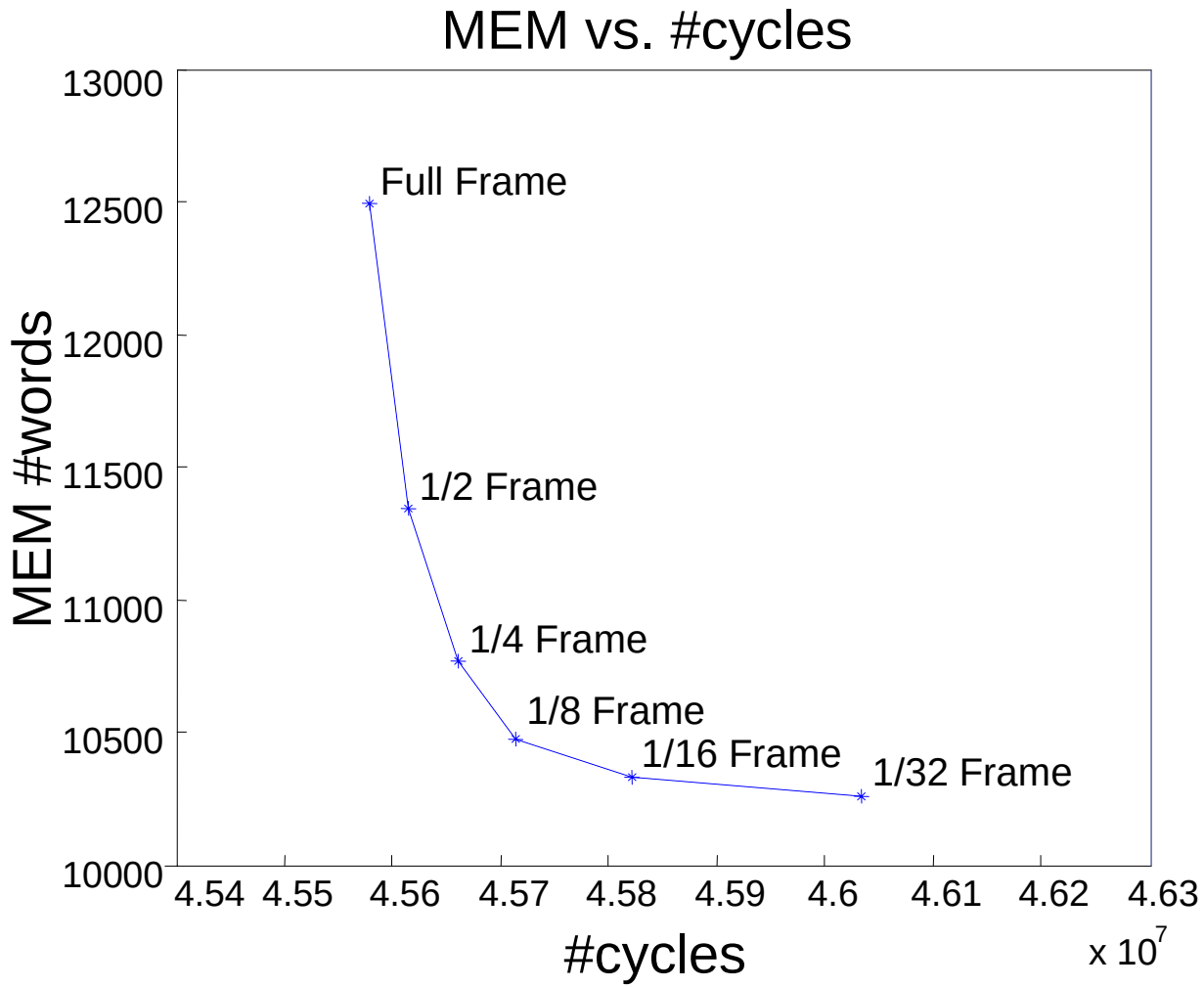


Results for Different Interface Types



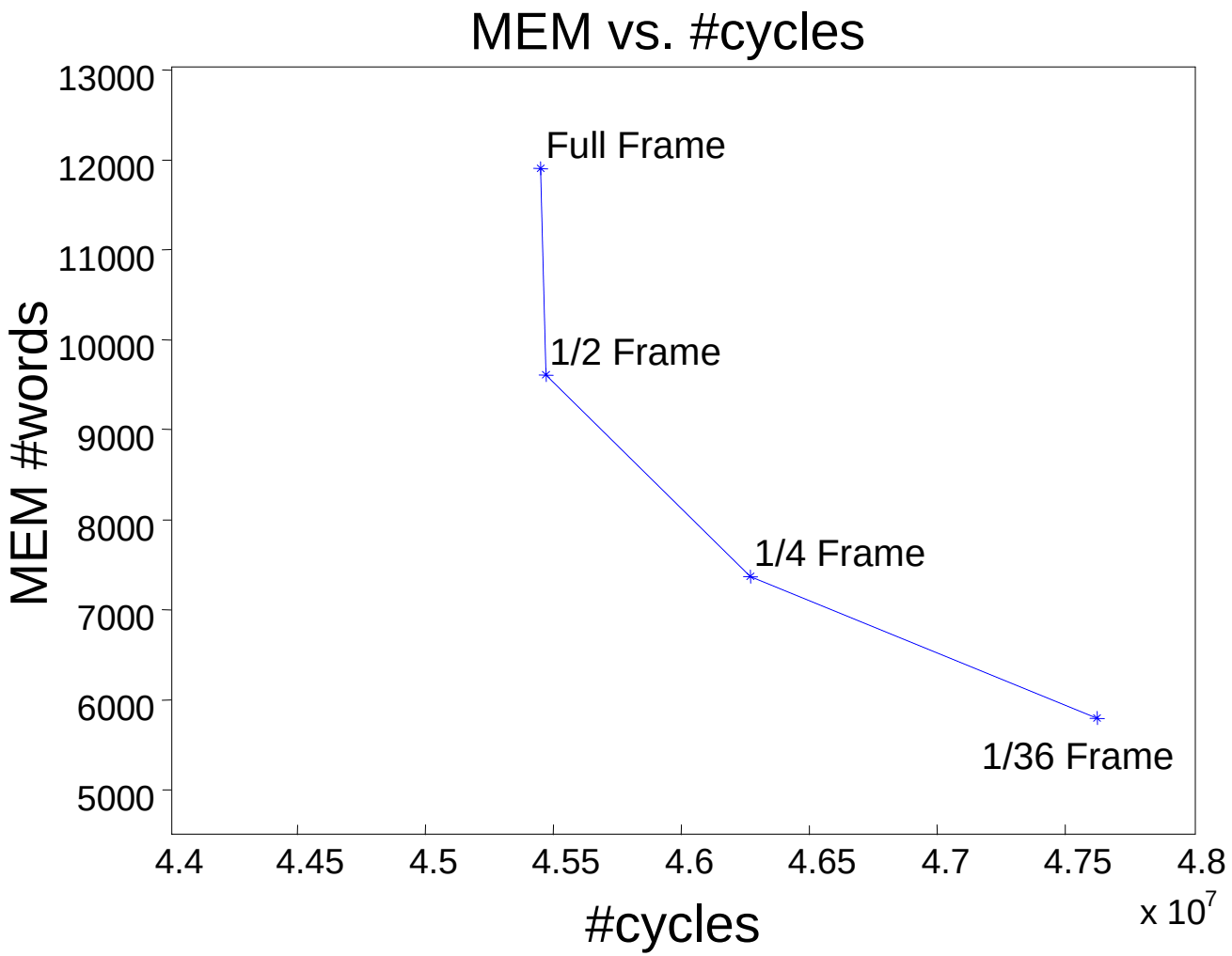
TTL IF Type	#Cycles	Part in TTL	#Memory words
CB	45579603	2.9%	12493
RB	45551243	2.8%	12494
RN	45505950	2.2%	12365
DBI	45152454	1.1%	9162
DNI	45108086	0.5%	9041

Results for Varying Channel Size (CB)



- Task code not modified
- Possible with CB
- Only channel buffer has been reduced in size

Results: Sub-frame Decoding (RN)



- Channel buffer and private buffers are reduced in size
- Task code must be modified
- Possible with all interface types

Smart Cameras Application Areas

Surveillance

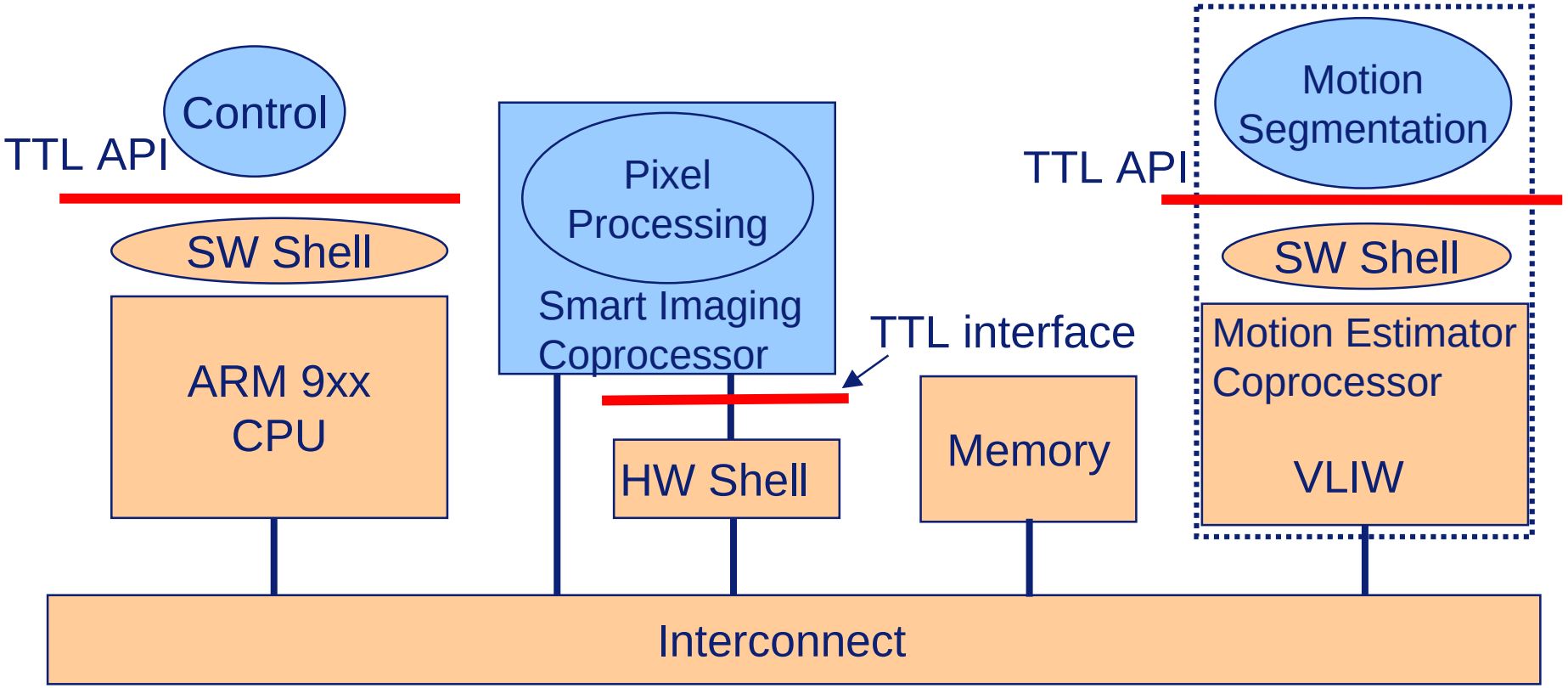
Consumer

Automotive

Mobile

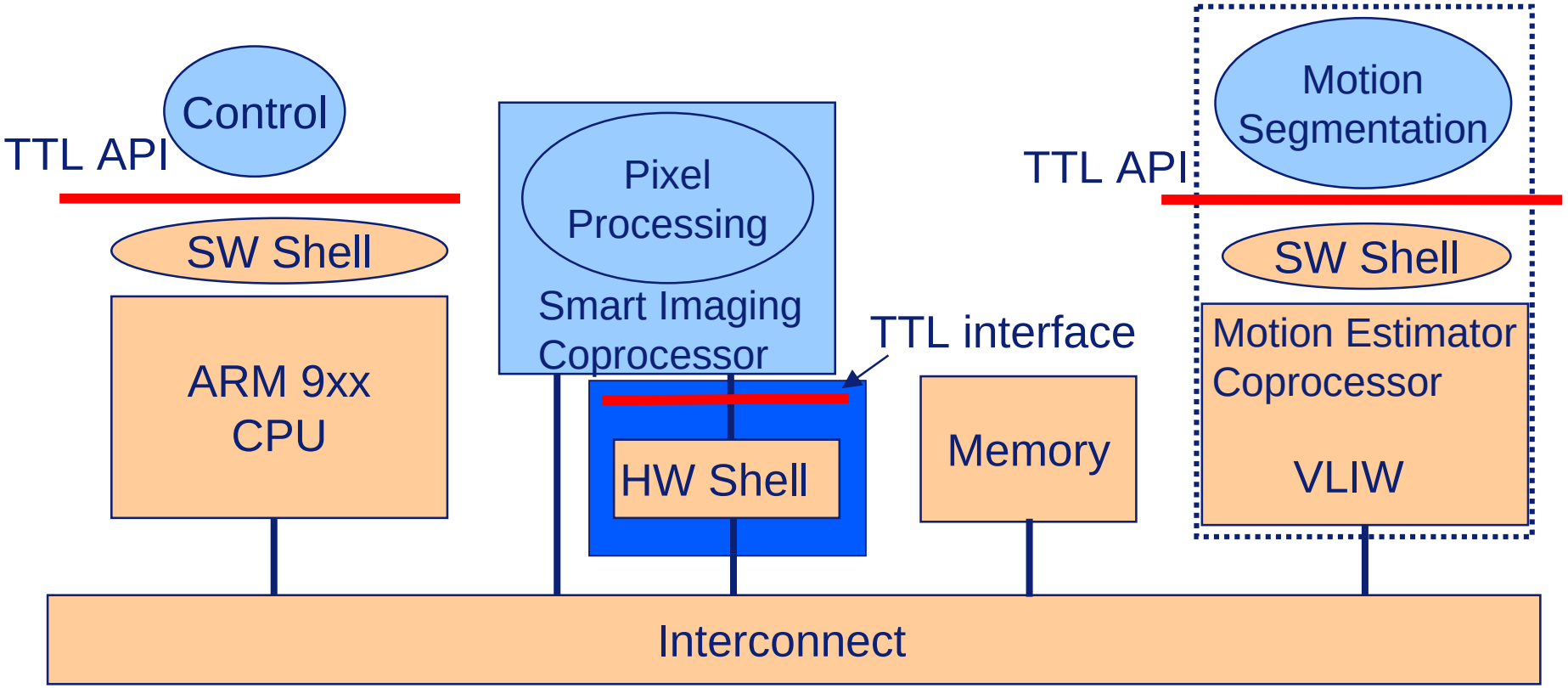
EC funded CAMELLIA project (IST-34410)

Architecture of Smart Imaging Core



- Enable efficient software – hardware communication
- Make all processors “self-synchronizing”

Architecture of Smart Imaging Core

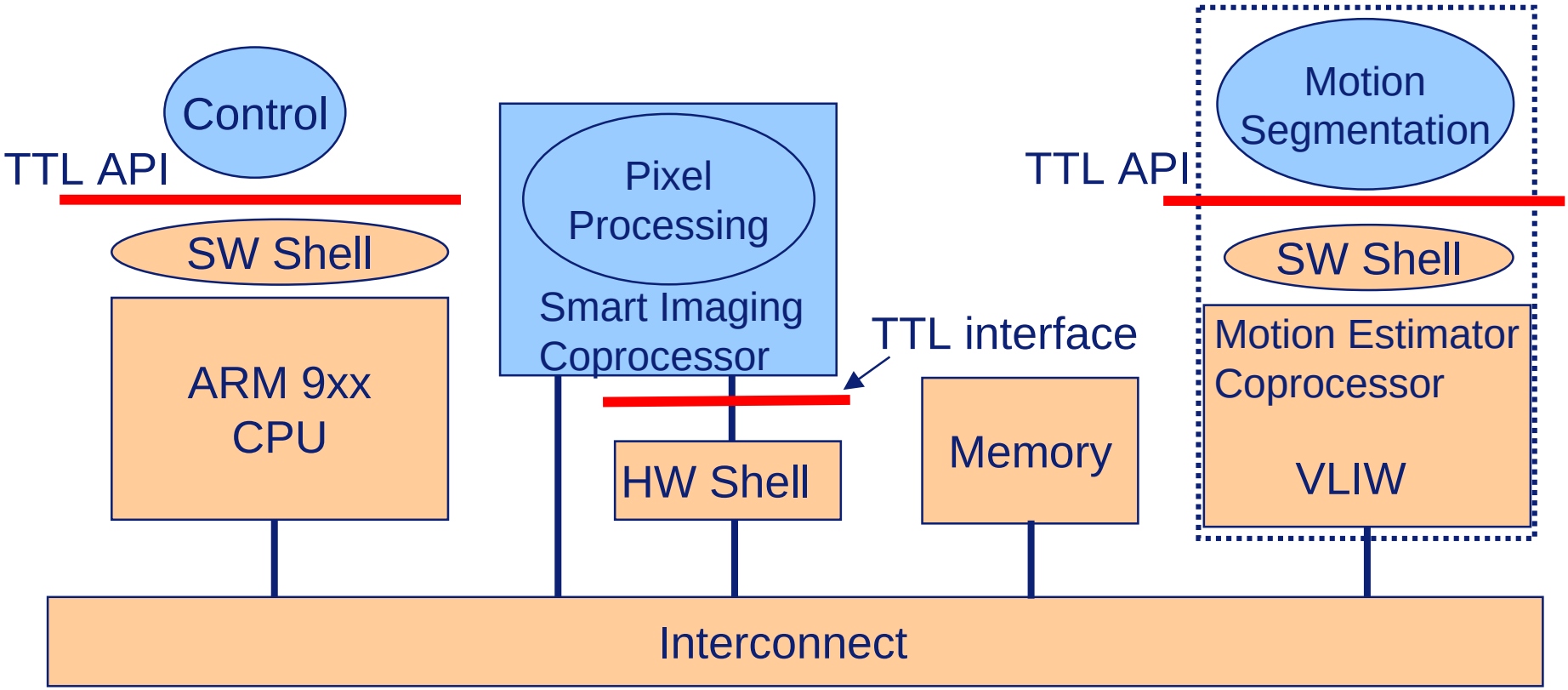


- Enable efficient software – hardware communication
- Make all processors “self-synchronizing”

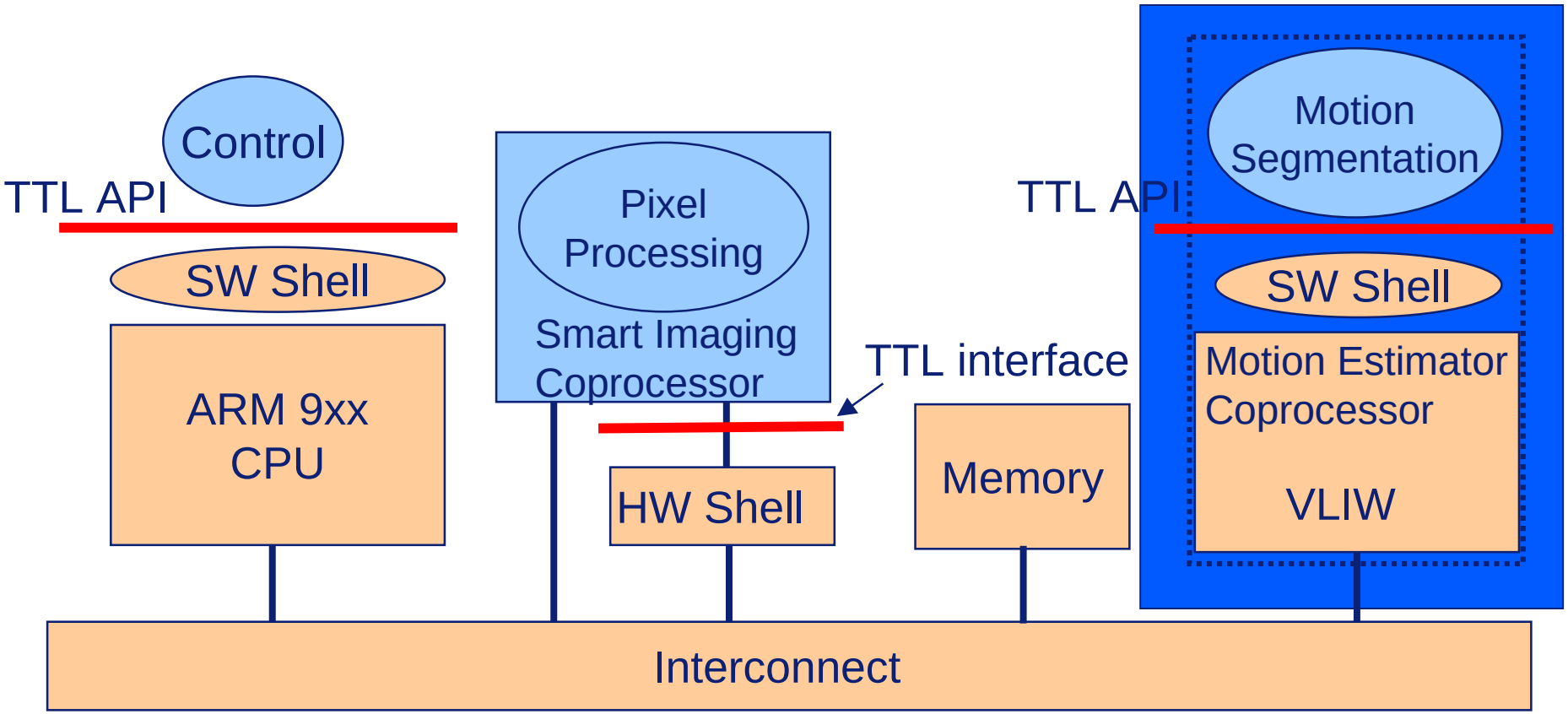
TTL shell performance

- HW Shell (channel administration local)
  - reAcquireRoom/Data 5 cycles
  - releaseRoom/Data 7 cycles
  - load 5 + 2n cycles
  - store 5 + n cycles

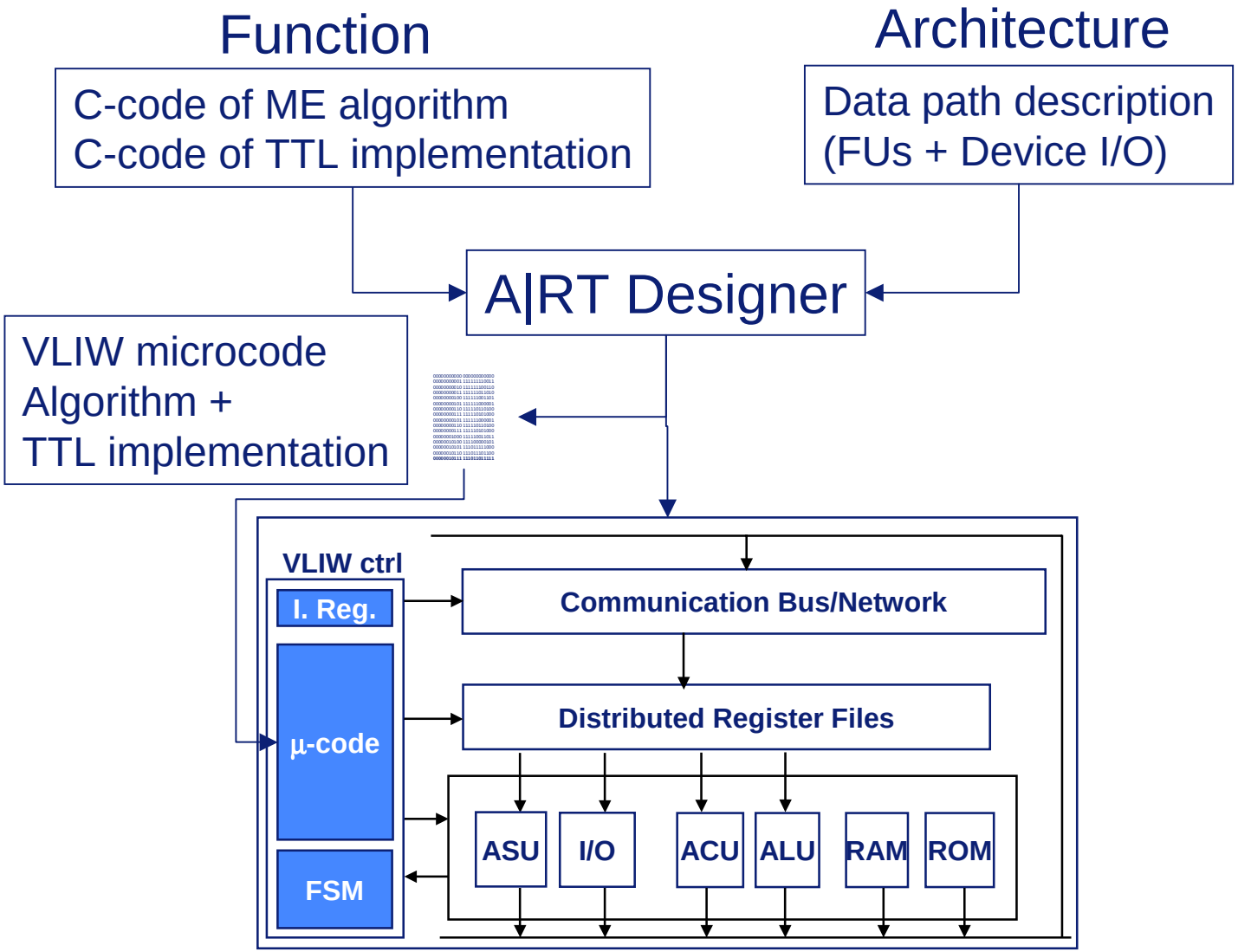
Architecture of Smart Imaging Core



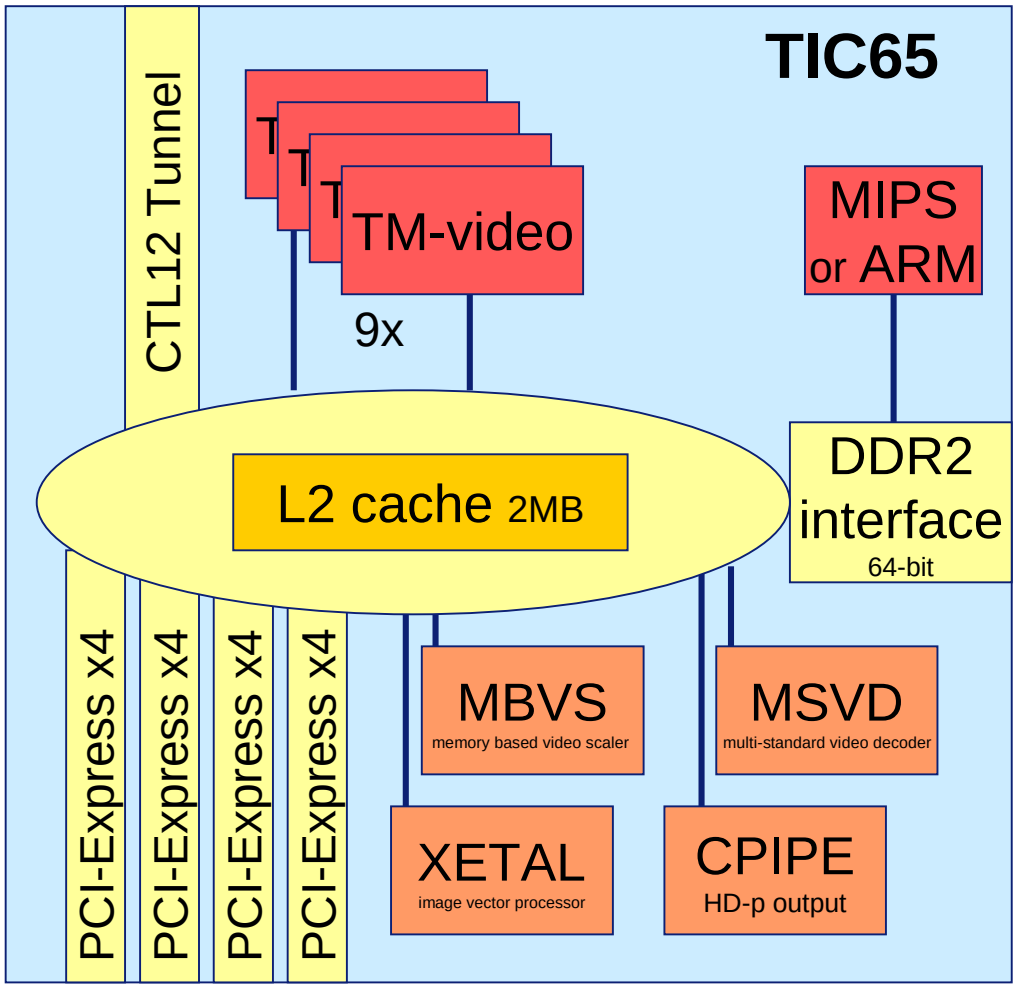
Architecture of Smart Imaging Core



TTL Implementation for ME



Cake / Wasabi



- Hybrid multiprocessor with homogeneous bias
- First silicon 2006

TTL Implementation on Cake / Wasabi

	MIPS	Trimedia
Cycles per sync operation (TTL on top of TRT run-time system)	20 (MIPS - MIPS)	20 (TM - TM)
Code size TTL (CB + DBI)	5 kB	14 kB
Lines of code TTL (CB + DBI)	773	773
Code size TTL (all IF types)	12 kB	29 kB
Lines of code TTL (all IF types)	1529	1529

Task-Level Interface Standardization

Industry-wide standardization needed

- Reuse of function-specific hardware and software IP
  - Enable eco-system of IP providers
- EDA for system-level design
  - Support development of function-specific IP
  - Support integration of IP

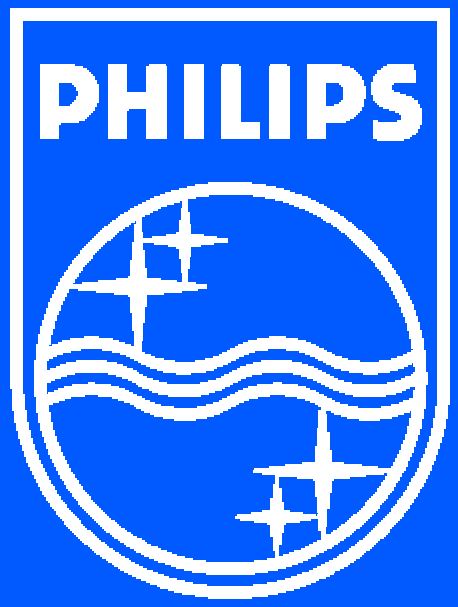
See also:

- Codes+ISSS'04, Inter-Task Communication and Multi-Tasking
- Codes+ISSS'05, Dynamic Reconfiguration

Conclusion

TTL supports structured and efficient design and integration of hardware and software tasks in MPSoCs

- High-level interface for ease of programming
  - Decreases design effort for task programmer
  - Facilitates reuse and integration of IP
  - Provides implementation freedom for platform infrastructure
- Enabler for automated mapping
  - Automated transformations support design optimizations
  - Closes gap between specification and implementation
  - Decreases design effort for system integrator
- Efficient implementation on range of platforms
  - Different architectures
  - In hardware and software
- Need for standardization


PHILIPS