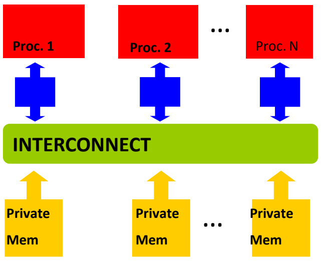
Supporting Software

Compilers and Autotuners
Virtual Machines
Middleware
Operating Systems

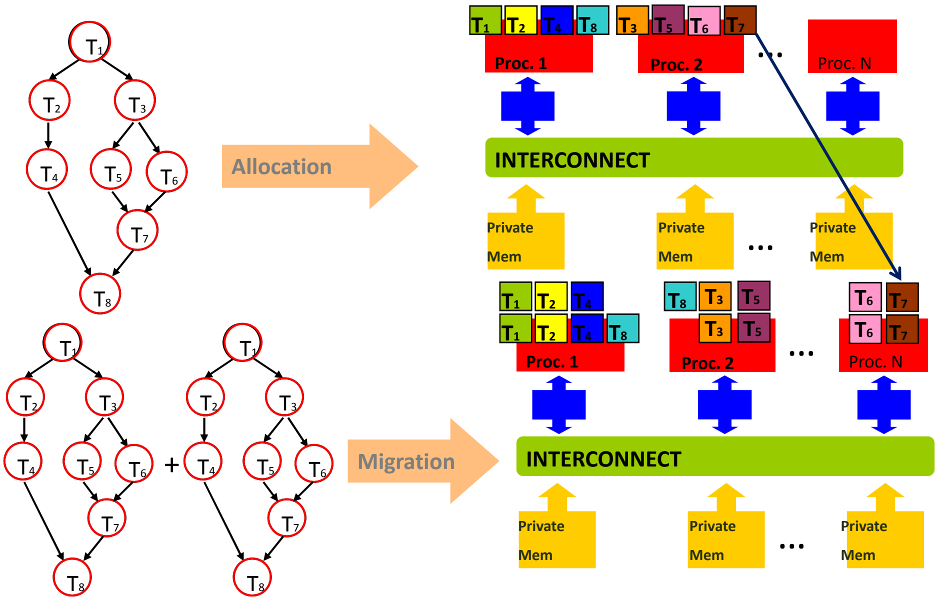
Task Migration on MPSoCs

- Needed for ***distributed memory not cache coherent MPSoCs***
  - Cache coherent SMP: move task status only…
  - Distributed memory MPSOC: move code and data too!!!
- Process migration to:
  - Workload balancing
  - Power consumption (with DVFS and SD)
  - facilitate thermal chip management
- Challenges for embedded systems
  - Low – overhead
  - Predictability

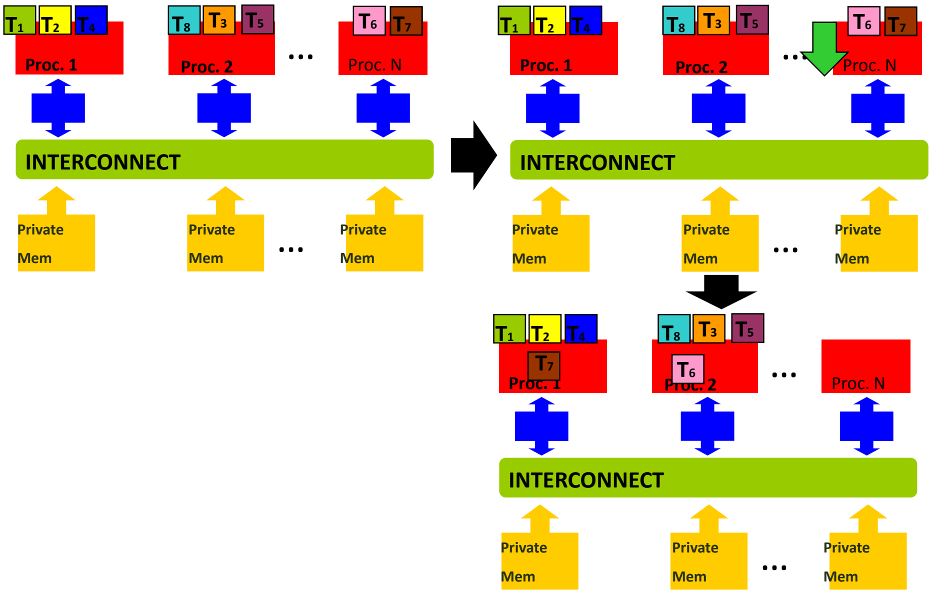


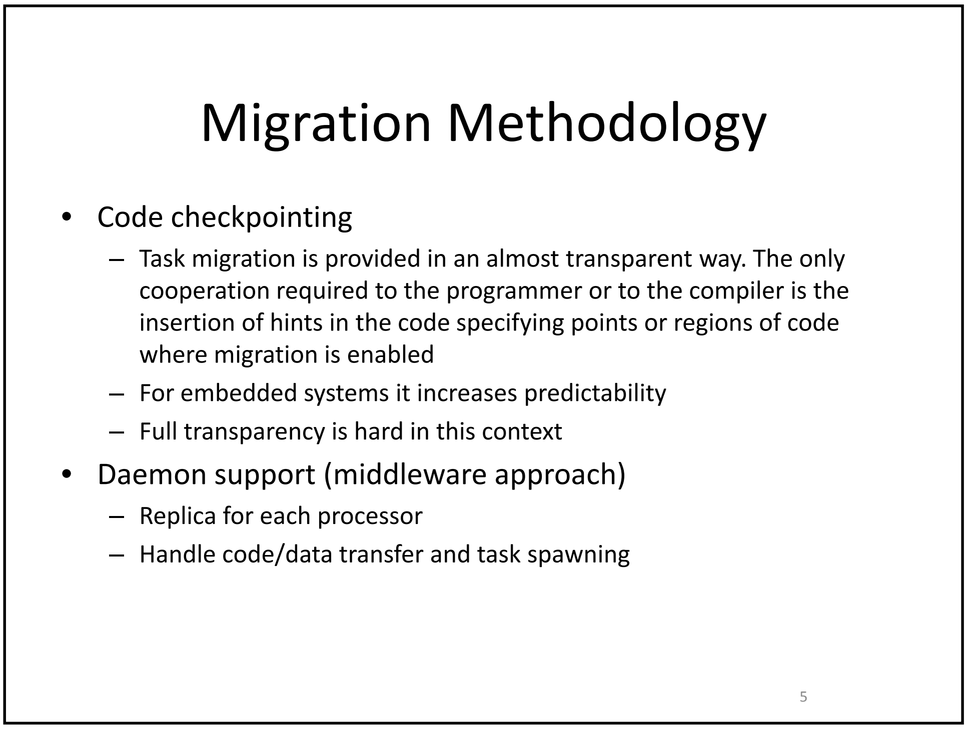
2

Task Graph Runtime Allocation



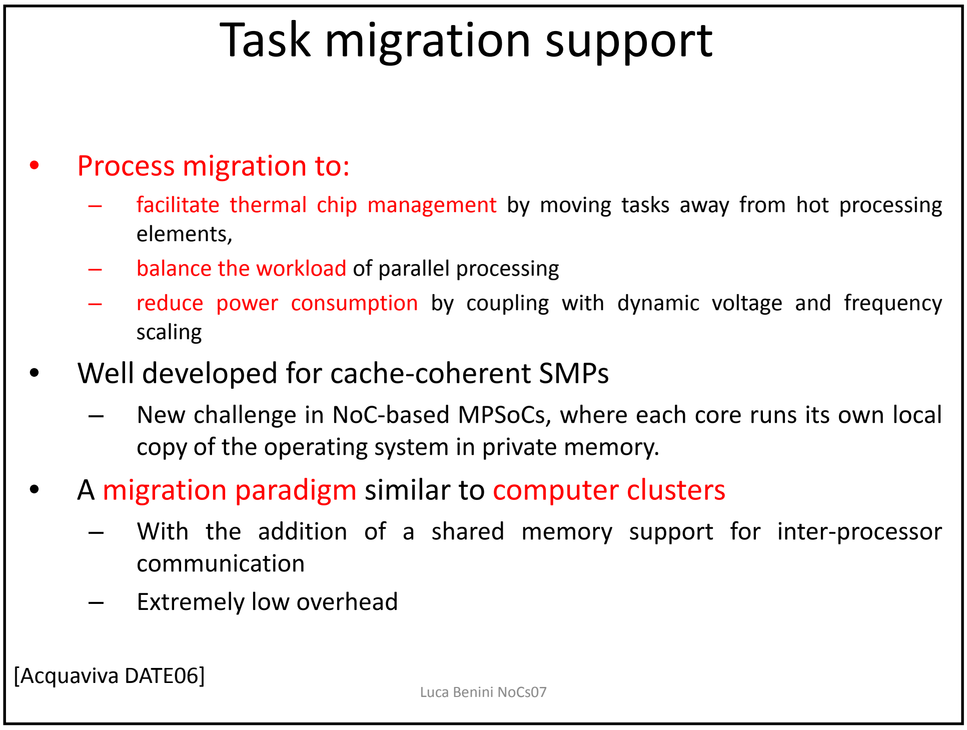
Migration for Variability





Migration Methodology

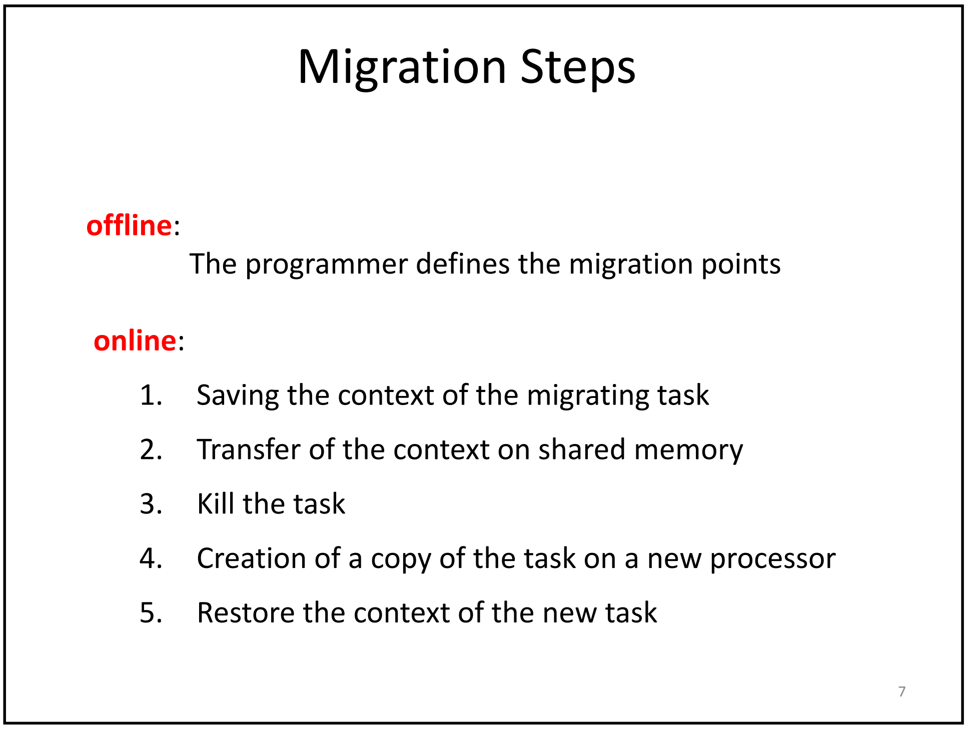
- Code checkpointing
  - Task migration is provided in an almost transparent way. The only cooperation required to the programmer or to the compiler is the insertion of hints in the code specifying points or regions of code where migration is enabled
  - For embedded systems it increases predictability
  - Full transparency is hard in this context
- Daemon support (middleware approach)
  - Replica for each processor
  - Handle code/data transfer and task spawning



Task migration support

- Process migration to:
  - facilitate thermal chip management by moving tasks away from hot processing elements,
  - balance the workload of parallel processing
  - reduce power consumption by coupling with dynamic voltage and frequency scaling
- Well developed for cache-coherent SMPs
  - New challenge in NoC-based MPSoCs, where each core runs its own local copy of the operating system in private memory.
- A migration paradigm similar to computer clusters
  - With the addition of a shared memory support for inter-processor communication
  - Extremely low overhead

[Acquaviva DATE06]



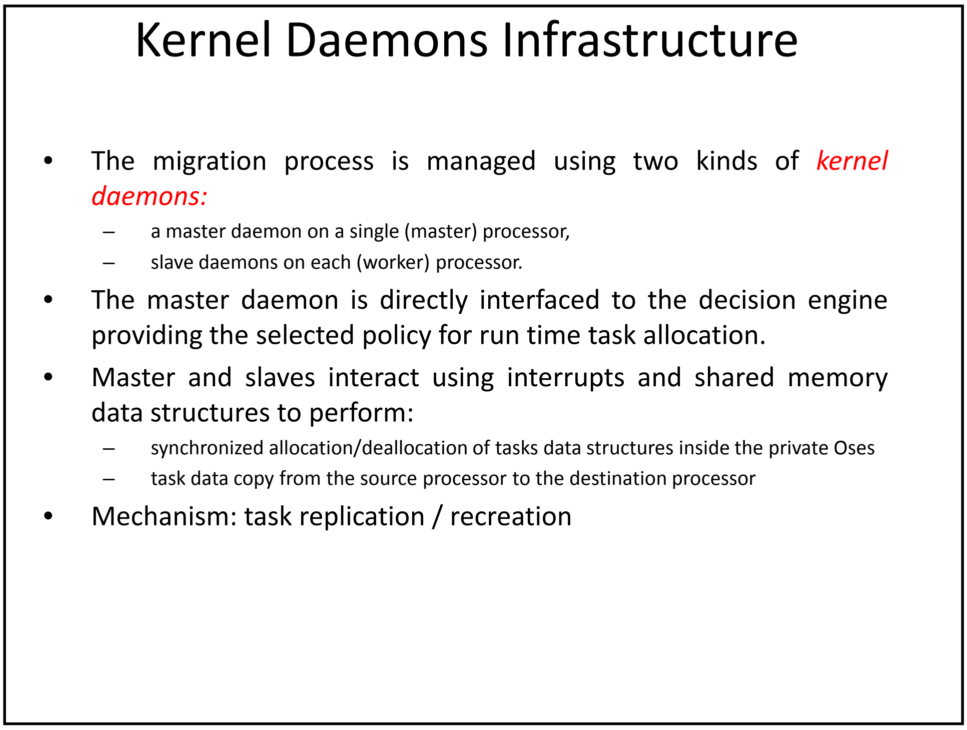
Migration Steps

offline:

The programmer defines the migration points

online:

1. Saving the context of the migrating task
2. Transfer of the context on shared memory
3. Kill the task
4. Creation of a copy of the task on a new processor
5. Restore the context of the new task



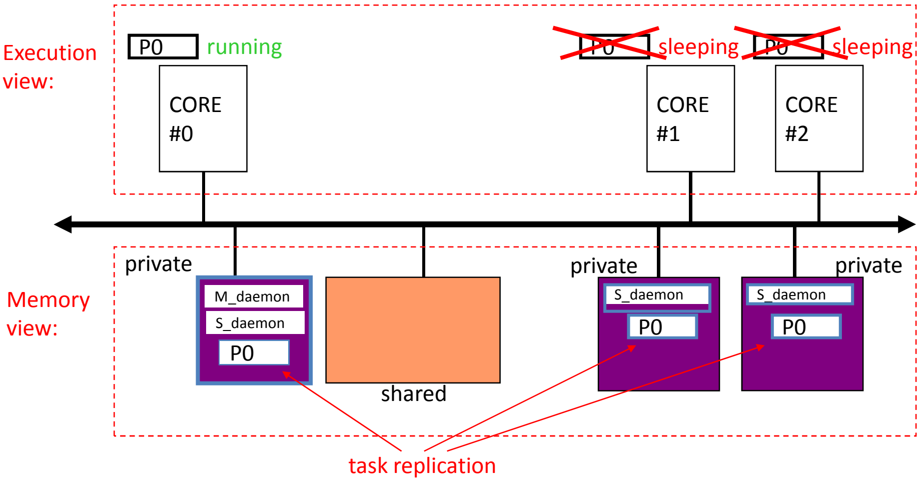
Kernel Daemons Infrastructure

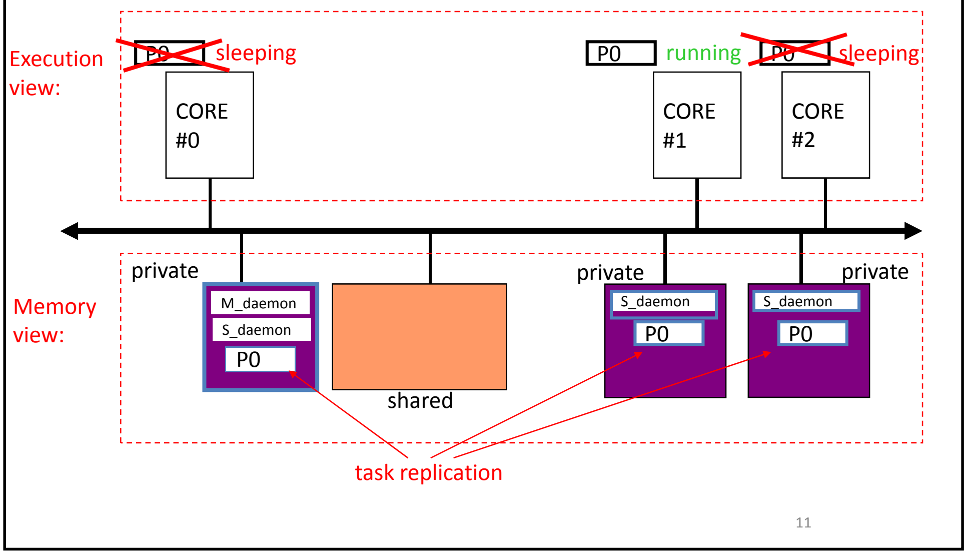
- The migration process is managed using two kinds of *kernel daemons:*
  - a master daemon on a single (master) processor,
  - slave daemons on each (worker) processor.
- The master daemon is directly interfaced to the decision engine providing the selected policy for run time task allocation.
- Master and slaves interact using interrupts and shared memory data structures to perform:
  - synchronized allocation/deallocation of tasks data structures inside the private Oses
  - task data copy from the source processor to the destination processor
- Mechanism: task replication / recreation

TM in Distributed Memory Architectures

- When moving app/kernel context pointers are not valid anymore unless data structures have the same position in memory
- Solutions
  - PIC (Position Independent Code)
  - Hardware support (replicated address space) but still need to keep the same position in memory (strong limitation)

Task replication





Execution view:

~~P0~~ sleeping

P0 running

~~P0~~ sleeping

CORE #0

CORE #1

CORE #2

Memory view:

private

M_daemon

S_daemon

P0

shared

private

S_daemon

P0

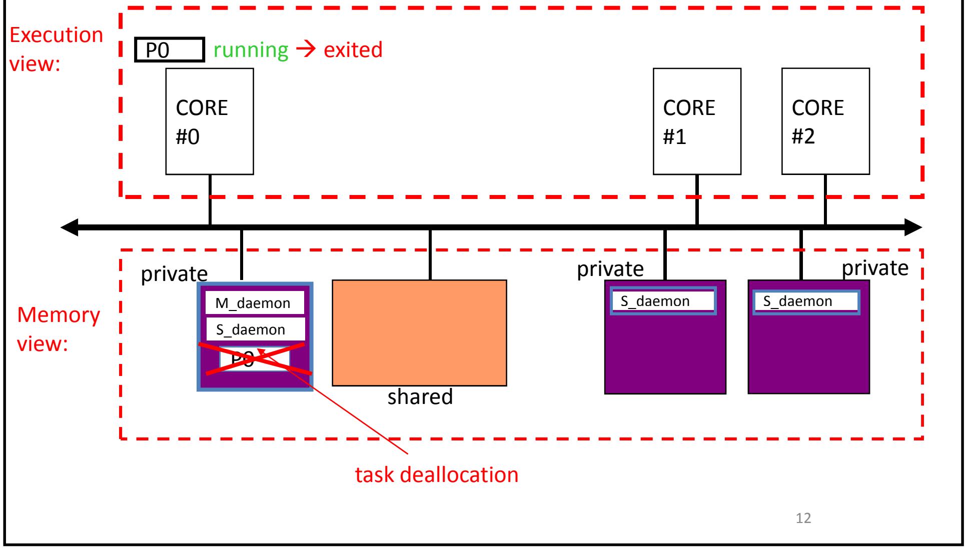
private

S_daemon

P0

task replication

Task re-creation



Execution view:

P0 running → exited

CORE #0

CORE #1

CORE #2

Memory view:

private

M_daemon

S_daemon

~~P0~~

shared

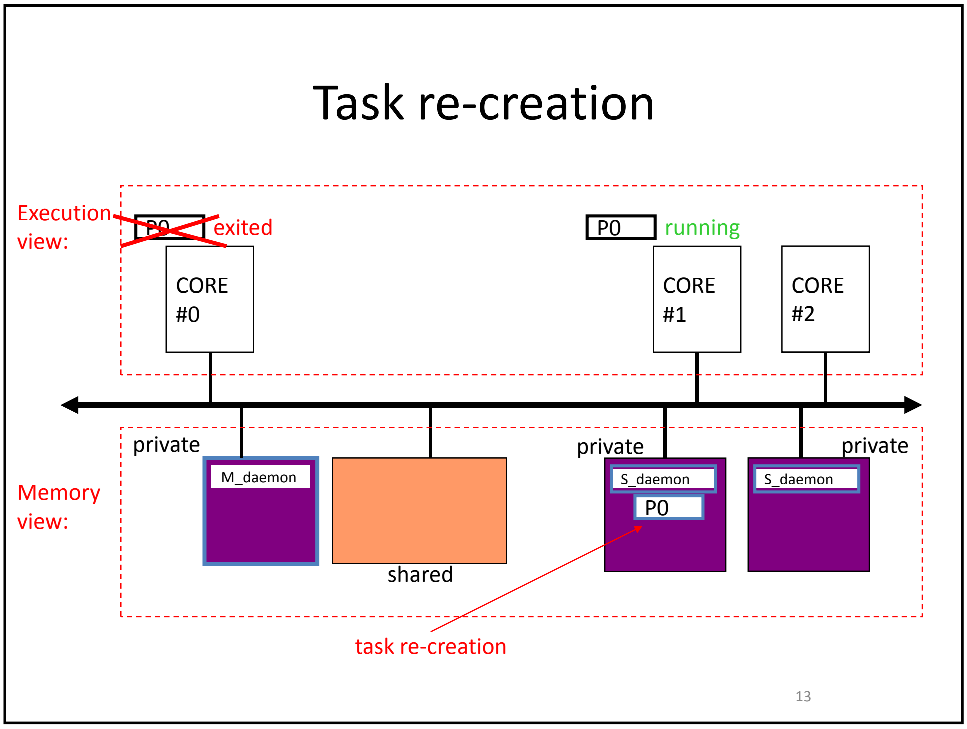
private

S_daemon

private

S_daemon

task deallocation



Task re-creation

Execution view:

~~P0~~ exited

P0 running

CORE #0

CORE #1

CORE #2

private

M_daemon

shared

private

S_daemon

P0

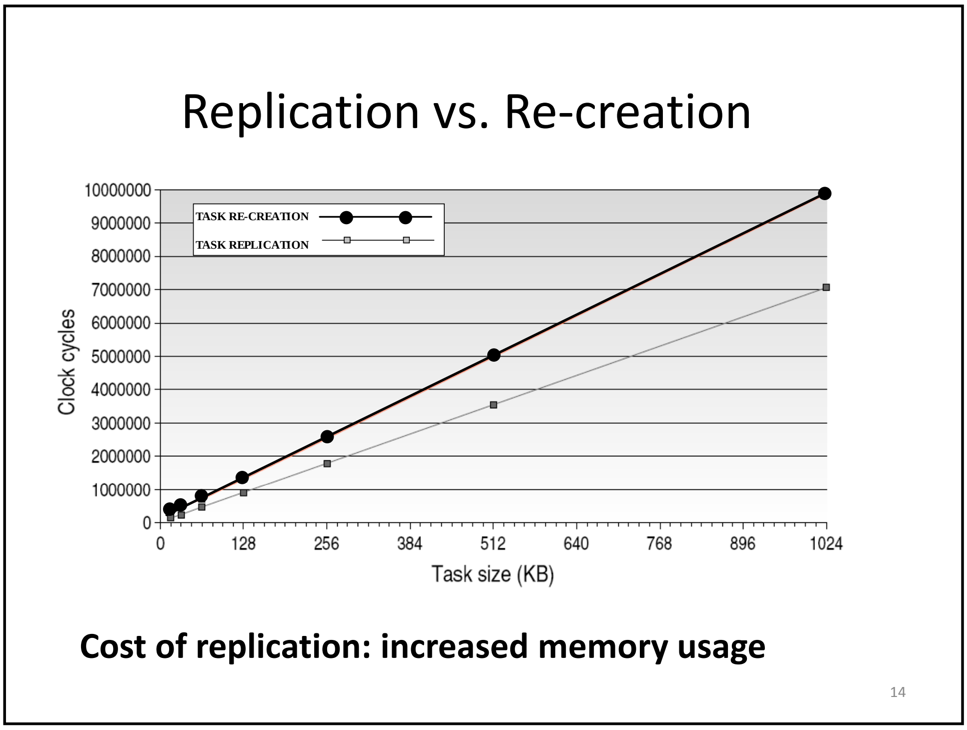
private

S_daemon

Memory view:

task re-creation

13

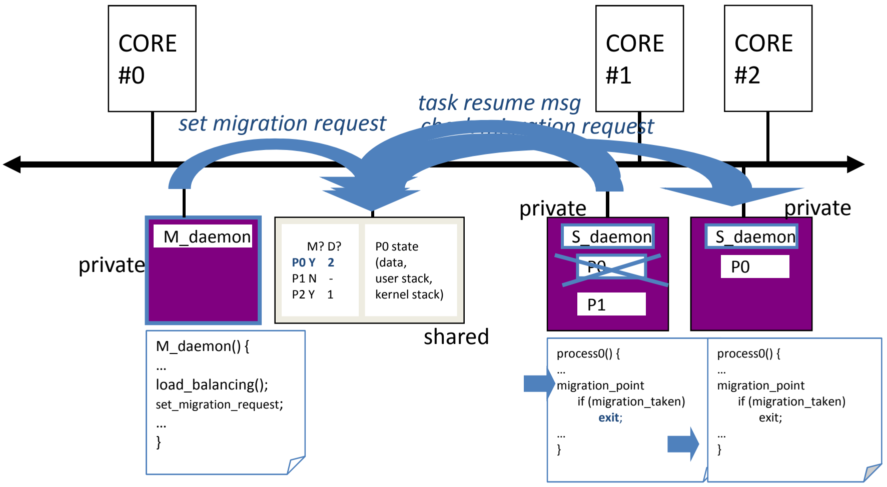


Replication vs. Re-creation

Cost of replication: increased memory usage

14

Migration mechanism



M? D?	P0 state
P0 Y 2	(data,
P1 N -	user stack,
P2 Y 1	kernel stack)

```
M_daemon() {
...
load_balancing();
set_migration_request;
...
}
```

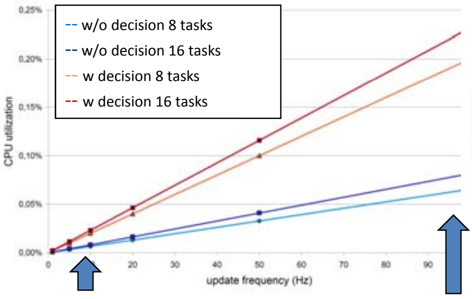
```
process0() {
...
migration_point
    if (migration_taken)
        exit;
...
}
```

```
process0() {
...
migration_point
    if (migration_taken)
        exit;
...
}
```

OS Implications

- Task replication is suitable for OS with no dynamic loading capabilities (eCos, RTEMS)
  - No need for dynamic process creation
  - User address space is statically assigned
  - Lower migration overhead but lower memory overhead
- Task re-creation requires dynamic loading capabilities (uClinux, Linux)
  - Exec is very expensive (address space copy)
  - Larger migration overhead but lower memory overhead

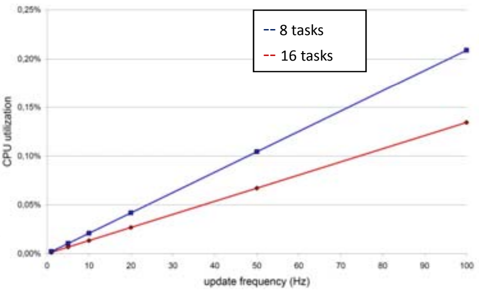
Deamon Overhead as a Function of Update Frequency



each 10 timeslices (100ms)

each timeslice (10ms)

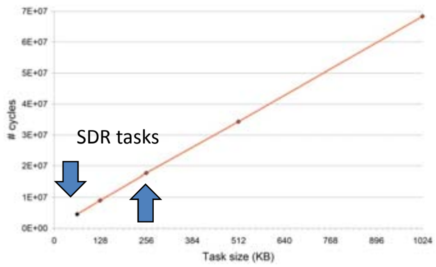
Master daemon



Slave daemon

- Master and slave daemons overhead as a function of update frequency
  - Reasonable update frequency lead to negligible overhead of middleware daemons
  - The overhead introduced by load balancing algorithm is still negligible

Deamon Overhead as a Function of Task Size



SDR tasks

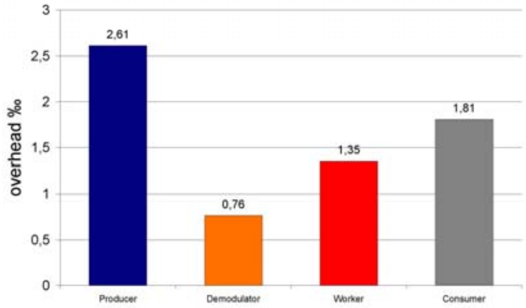
Note: minimum size is 64KBytes because of OS min allocation for processor address space

Ex: tasks size of Software Defined Radio application fits in 64Kbytes

- Migration overhead:
  - 2Mcycles for 256Kbytes task => roughly 10ms @ 200MHz
  - 0.5Mcycles for 64Kbytes task => roughly 2.5ms @ 200MHz
  - Migration overhead is on the order of one timeslice
- BUT: most of the accesses are to the shared memory => impact of bus contention must be taken into account which may impact predictability

Checkpoint Overhead: SW Radio Example

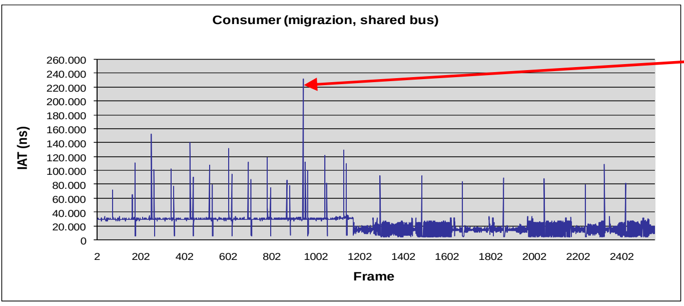
```
while(1) {

   checkpoint();

   infifo.get(&elem);
   for(q=0; q < FM_FRAME_SIZE; q++) {
      // demodulation
      temp = (elem.d[q] * lastElem);
      elem.d[q] = mGain *
            arctan(elem.d[q], 1.0/lastElem);
      lastElem = elem.d[q];
   }
   for(int w=0; w<n_workers; w++)  {
      outfifo[w]->put(&elem);
   }
}
```



User-Level Migration

- OS-assisted migration implies modification of the OS
  - User specifies migration points only
- User level approach is based on a user-level library
  - Portability
  - User is responsible of context definition
- Alternative:
  - Compiler-assisted context definition

Migration overhead



Daemon on master processor decides migration

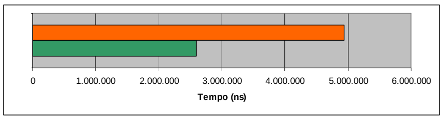
Before migration
Throughput 32165 Frames/sec

After migration
Throughput 62955 Frames/sec

95.7% improvement

Migration time (from command to restore): 7.54 ms

Treaction
Texec



65.6%
34.4%

Optimizations are needed to reduce migration overhead

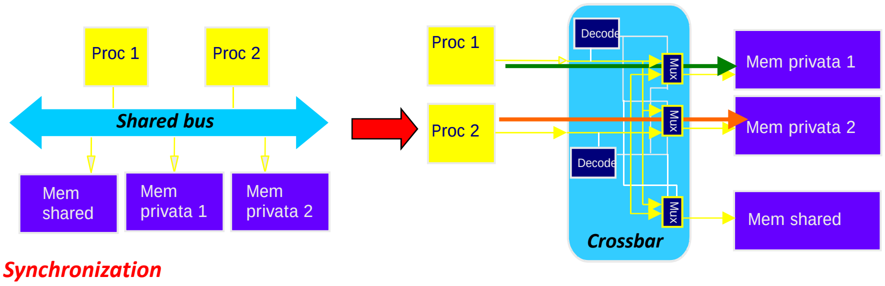
Optimization of Migration Costs

- Architectural/infrastructural optimizations
  - Interconnection
  - Interrupts
  - Scheduling effects
- Software optimizations
  - Migration context reduction
  - Efficient use of memory hierarchy

Communication and Synchronization Optimizations

Communication architecture

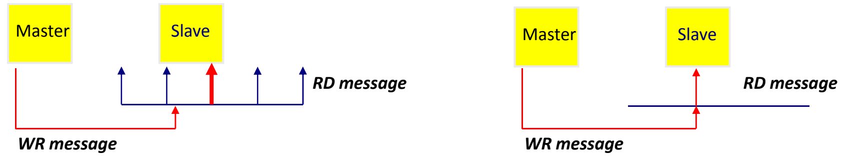
Core access to their private memory without contention largely optimizes ***exec*** overhead



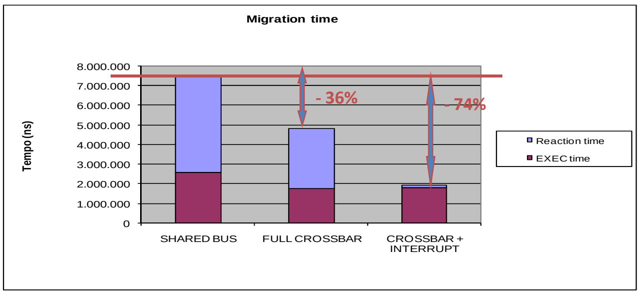
Synchronization

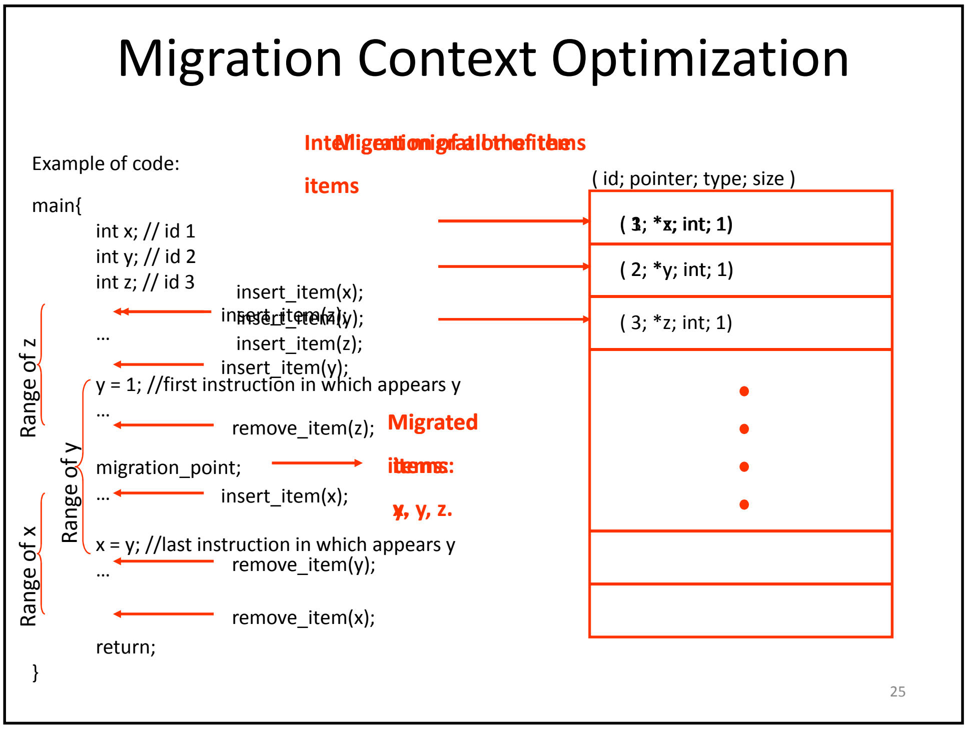
Synchronization between deamons using ***interrupts*** improves reaction time

Polling ***Interrupt***



Results

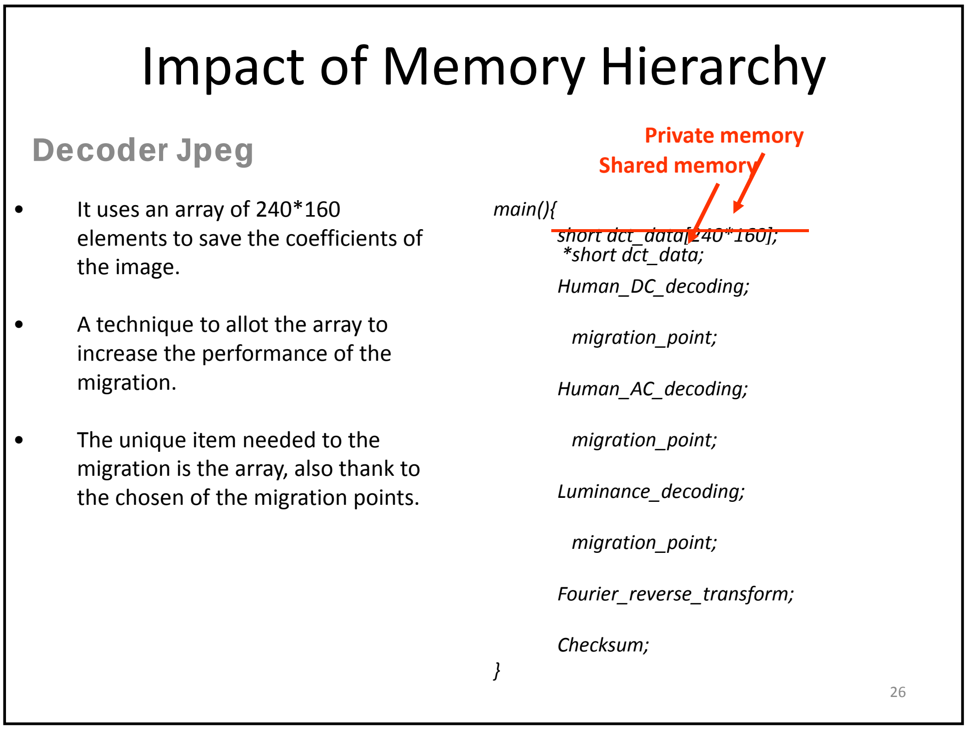




Migration Context Optimization

Example of code:

```
main{
        int x; // id 1
        int y; // id 2
        int z; // id 3
        ...
        y = 1; //first instruction in which appears y
        ...
        migration_point;
        ...
        x = y; //last instruction in which appears y
        ...
        return;
}
```

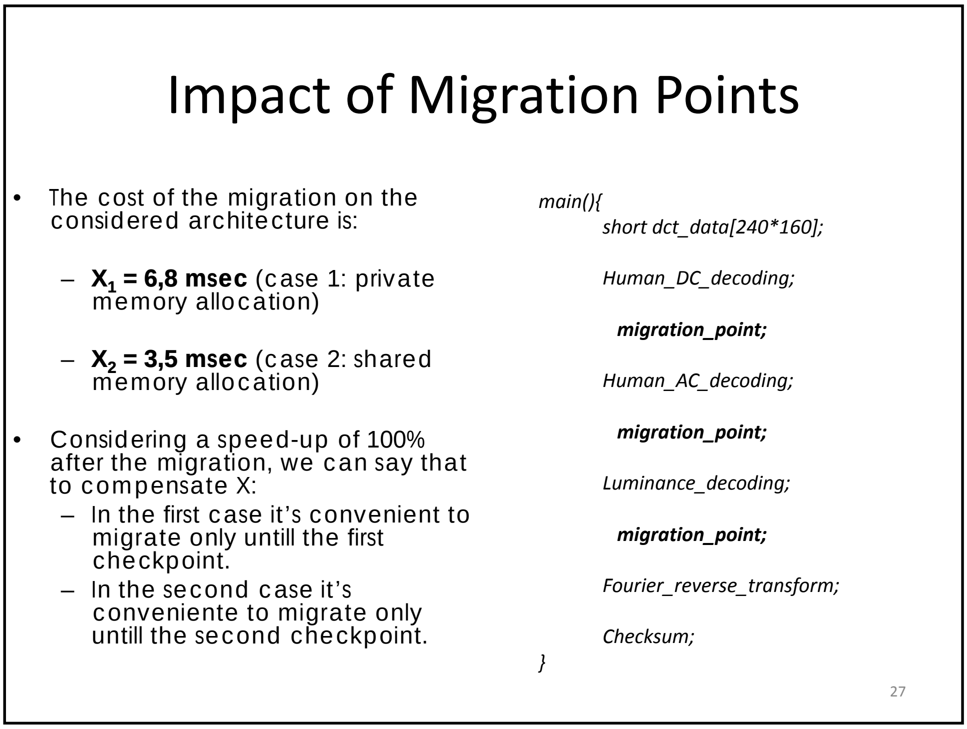


Impact of Memory Hierarchy

Decoder Jpeg

- It uses an array of 240*160 elements to save the coefficients of the image.
- A technique to allot the array to increase the performance of the migration.
- The unique item needed to the migration is the array, also thank to the chosen of the migration points.

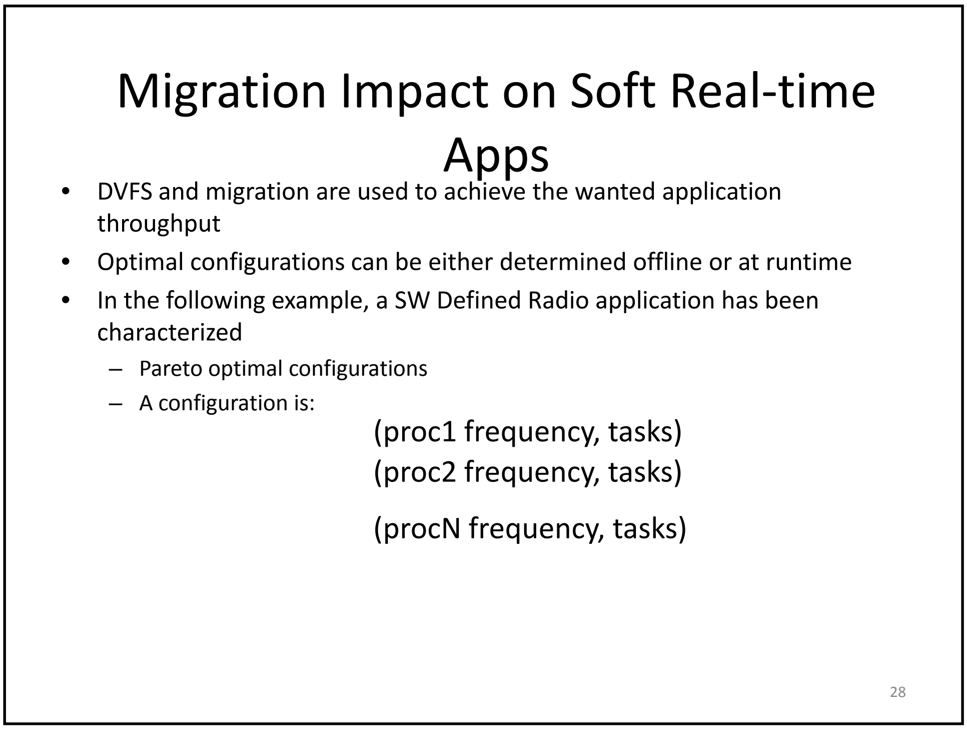
main(){
*~~short dct_data[240*160];~~*
**short dct_data;*
Human_DC_decoding;
migration_point;
Human_AC_decoding;
migration_point;
Luminance_decoding;
migration_point;
Fourier_reverse_transform;
Checksum;
}



Impact of Migration Points

- The cost of the migration on the considered architecture is:
  - **X_1 = 6,8 msec** (case 1: private memory allocation)
  - **X_2 = 3,5 msec** (case 2: shared memory allocation)
- Considering a speed-up of 100% after the migration, we can say that to compensate X:
  - In the first case it’s convenient to migrate only untill the first checkpoint.
  - In the second case it’s conveniente to migrate only untill the second checkpoint.

```
main(){
        short dct_data[240*160];

        Human_DC_decoding;

          migration_point;

        Human_AC_decoding;

          migration_point;

        Luminance_decoding;

          migration_point;

        Fourier_reverse_transform;

        Checksum;
}
```



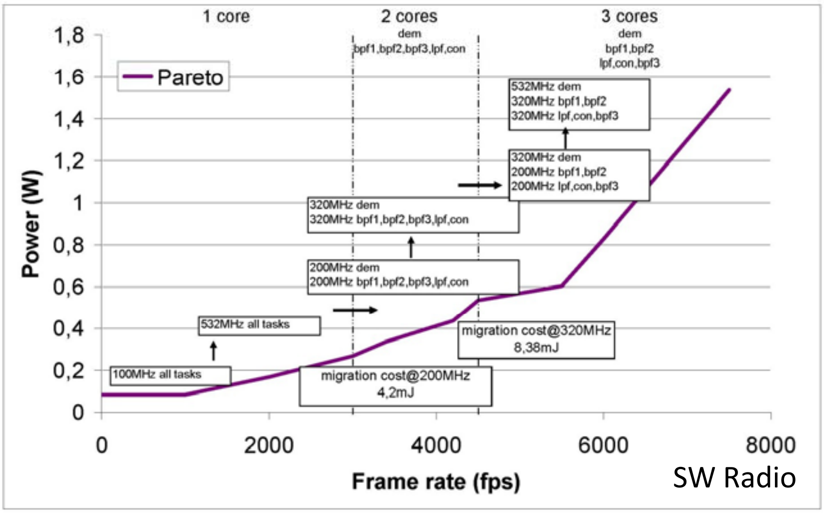
Migration Impact on Soft Real-time Apps

- DVFS and migration are used to achieve the wanted application throughput
- Optimal configurations can be either determined offline or at runtime
- In the following example, a SW Defined Radio application has been characterized
  - Pareto optimal configurations
  - A configuration is:

(proc1 frequency, tasks)
(proc2 frequency, tasks)

(procN frequency, tasks)

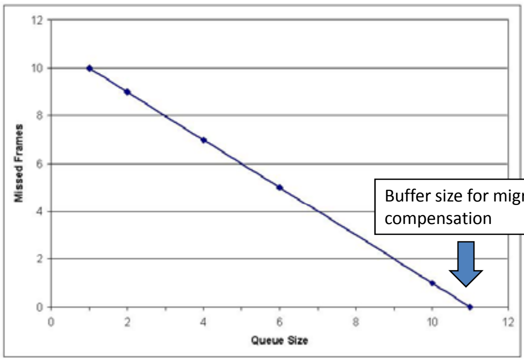
Migration to handle Runtime Performance Requirements



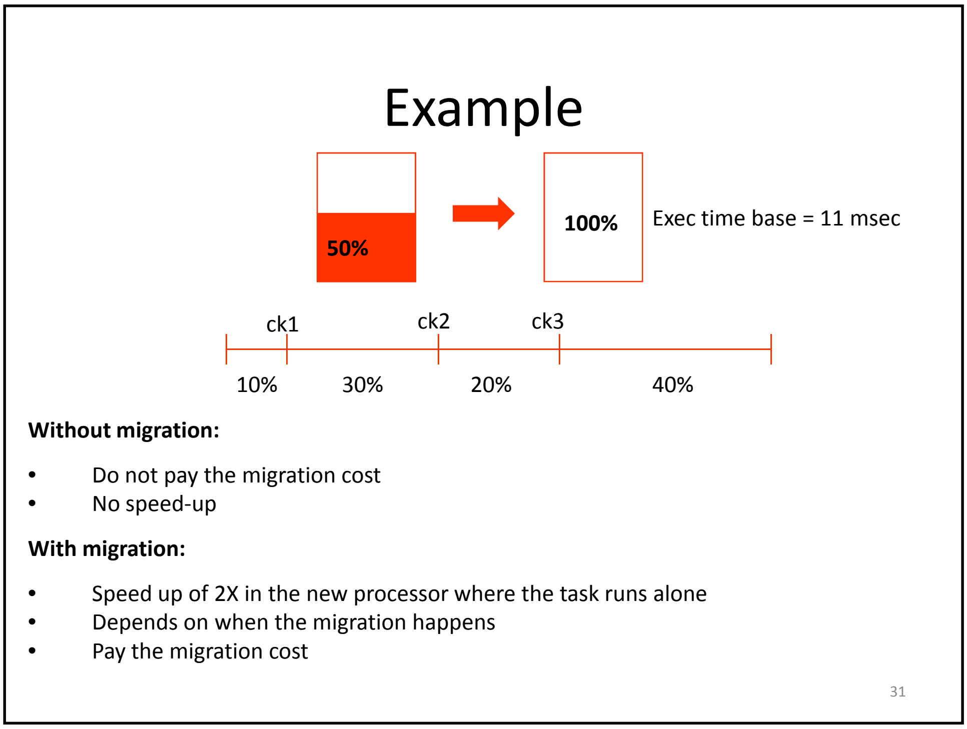
SW Radio

Migration Impact on Streaming Applications

- Deadline misses due to migration as a function of queue size
- Migration is a sporadic event: it can be handled with suitable buffer design



Buffer size for migration cost compensation



Example

50%

100%

Exec time base = 11 msec

ck1 ck2 ck3

10% 30% 20% 40%

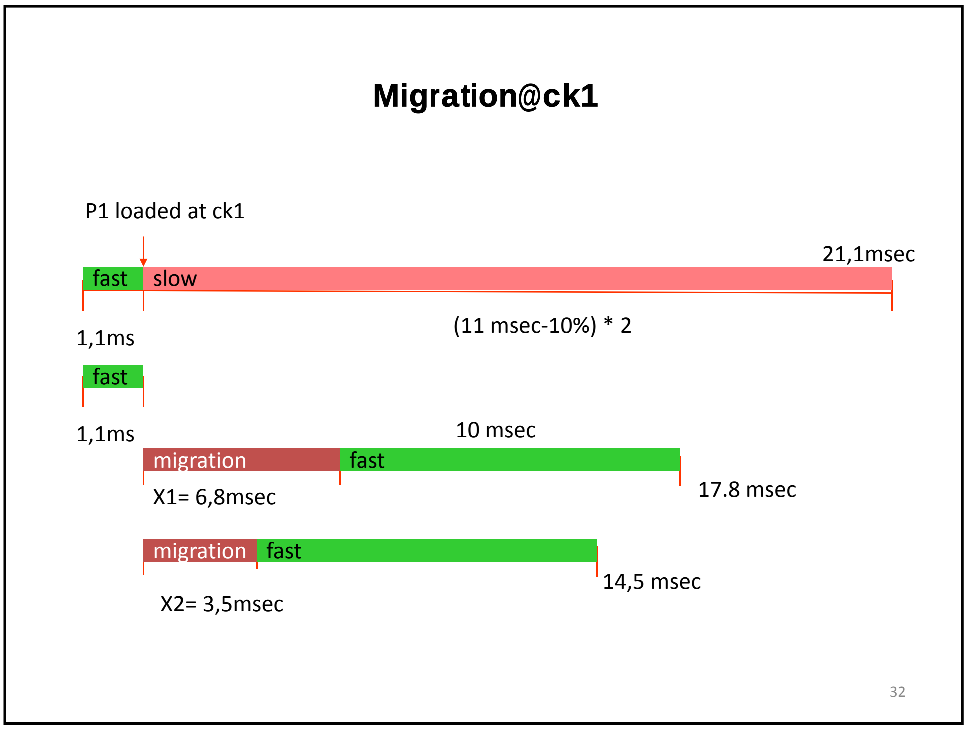
Without migration:

- Do not pay the migration cost
- No speed-up

With migration:

- Speed up of 2X in the new processor where the task runs alone
- Depends on when the migration happens
- Pay the migration cost

31



Migration@ck1

P1 loaded at ck1

fast slow

21,1msec

(11 msec-10%) * 2

1,1ms

fast

1,1ms

10 msec

migration fast

X1= 6,8msec

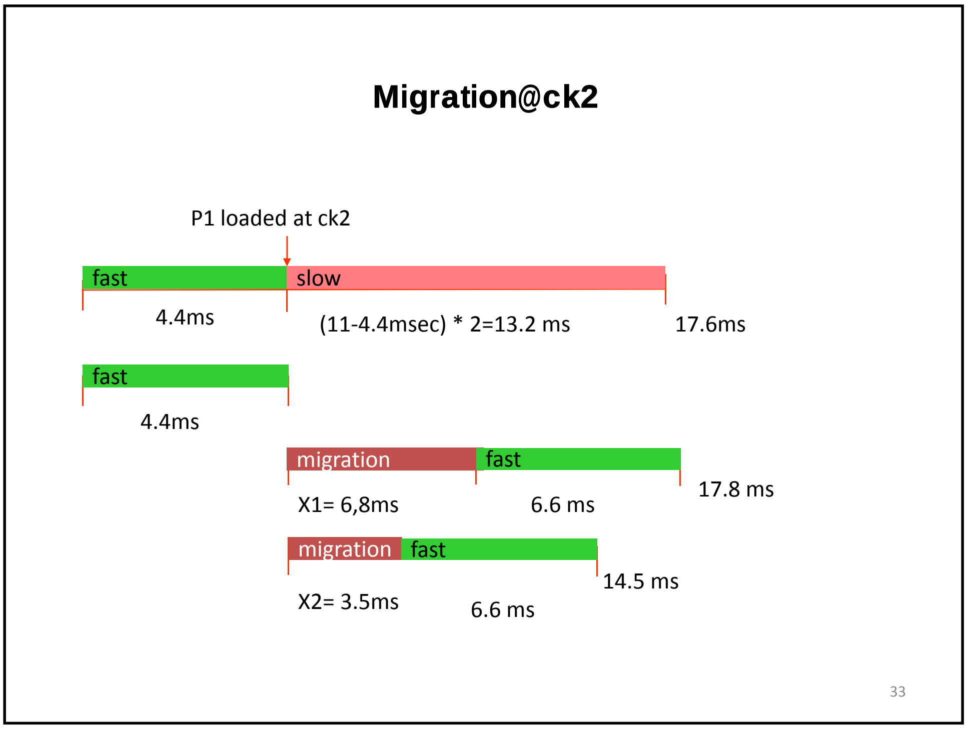
17.8 msec

migration fast

14,5 msec

X2= 3,5msec

32



Migration@ck2

P1 loaded at ck2

fast

slow

4.4ms

(11-4.4msec) * 2=13.2 ms

17.6ms

fast

4.4ms

migration

fast

X1= 6,8ms

6.6 ms

17.8 ms

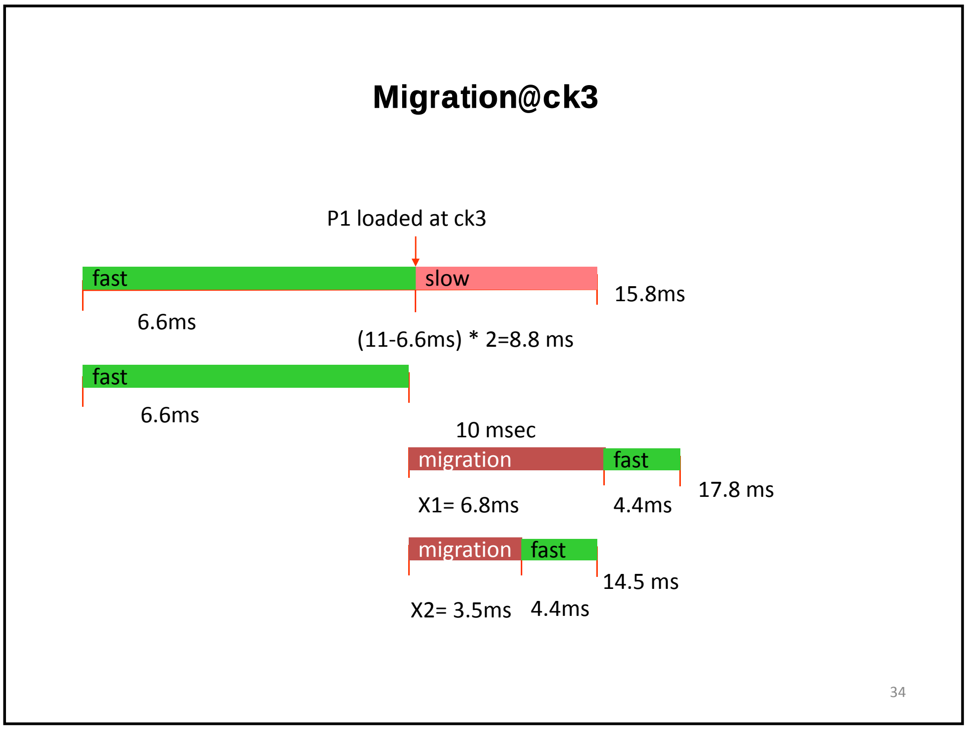
migration

fast

14.5 ms

X2= 3.5ms

6.6 ms



Migration@ck3

P1 loaded at ck3

fast

slow

15.8ms

6.6ms

(11-6.6ms) * 2=8.8 ms

fast

6.6ms

10 msec

migration

fast

17.8 ms

X1= 6.8ms

4.4ms

migration

fast

14.5 ms

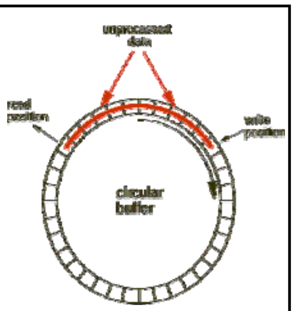
X2= 3.5ms

4.4ms

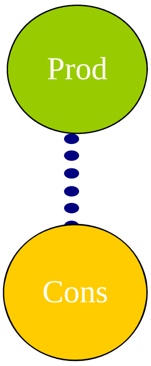
Message Passing

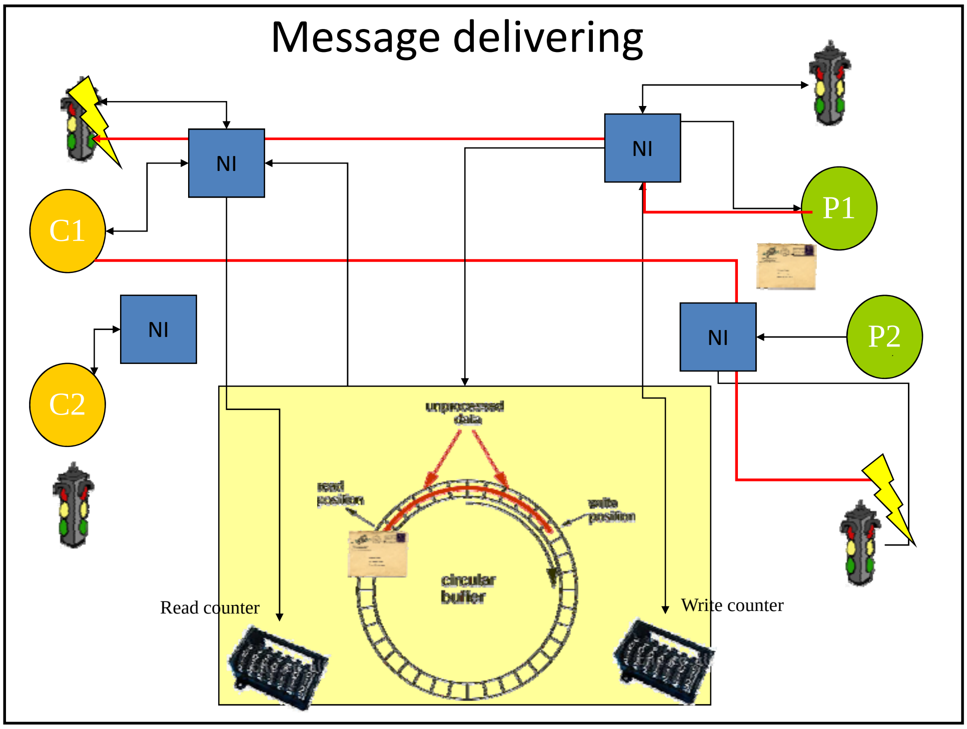
- Message passing (from Alessandro)
- Remote objects (from Luca)

MP-Queue library

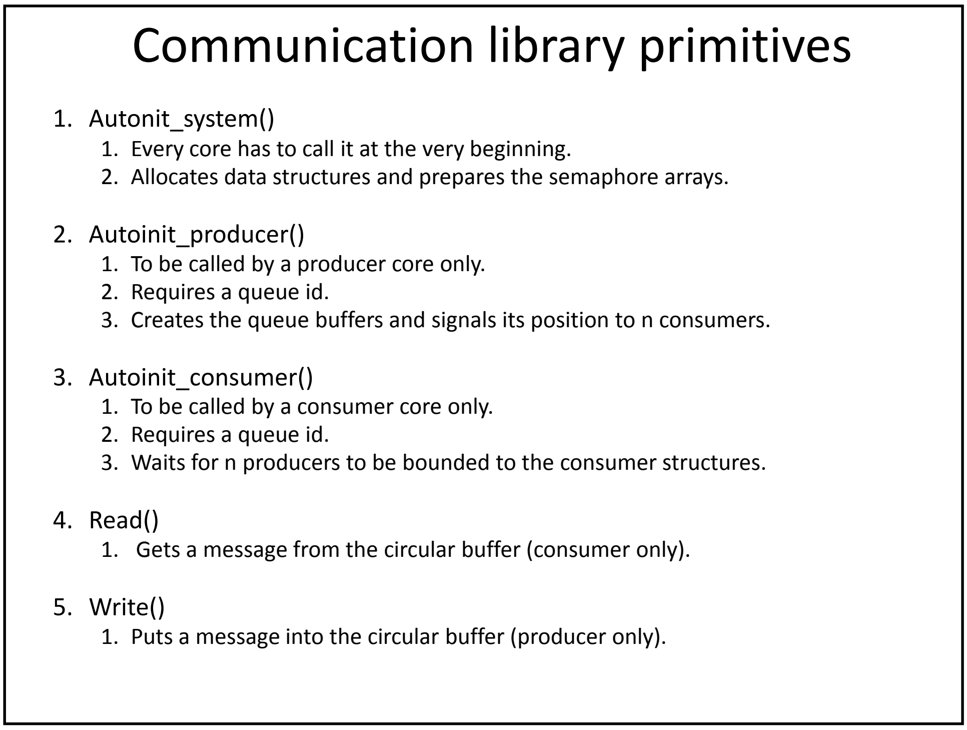


- **Distributed queue middleware**:
  - Support library for streaming channels in a MPSoC environment.
  - Producer / consumer paradigm.
  - FIFOs are circular buffers.
- Three main contributions:
  1. Configurability: MP-Queue library matches several heterogeneous architectural templates.
  2. An architecture independent efficiency metric is given.
  3. It achieves both high efficiency and portability:
     - synch operations optimized for minimal interconnect utilization.
     - data transfer optimized for performance through analyses of disassembled code.
     - portable C is the final implementation language.





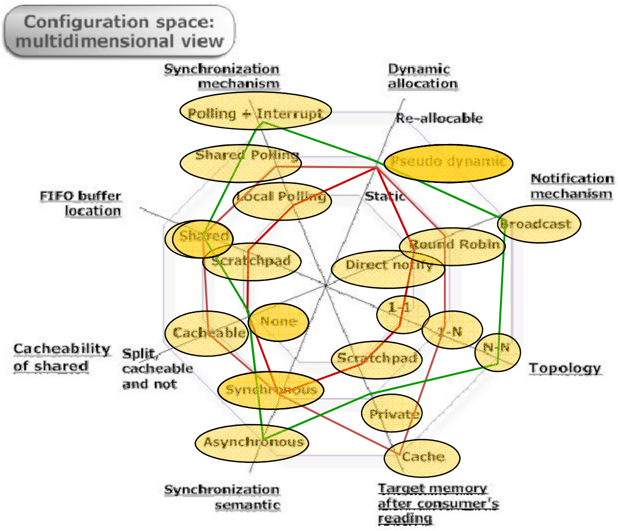
Message delivering

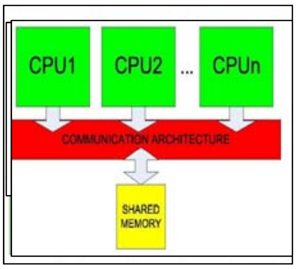


Communication library primitives

1. Autonit_system()
   1. Every core has to call it at the very beginning.
   2. Allocates data structures and prepares the semaphore arrays.
2. Autoinit_producer()
   1. To be called by a producer core only.
   2. Requires a queue id.
   3. Creates the queue buffers and signals its position to n consumers.
3. Autoinit_consumer()
   1. To be called by a consumer core only.
   2. Requires a queue id.
   3. Waits for n producers to be bounded to the consumer structures.
4. Read()
   1. Gets a message from the circular buffer (consumer only).
5. Write()
   1. Puts a message into the circular buffer (producer only).

Architectural Flexibility

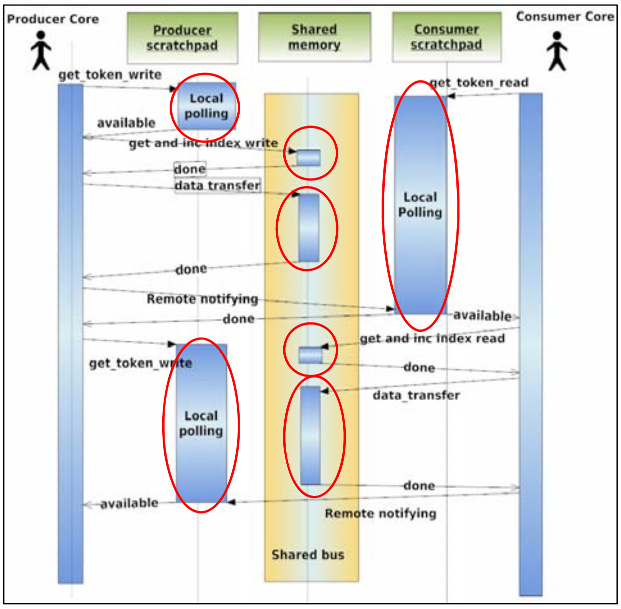




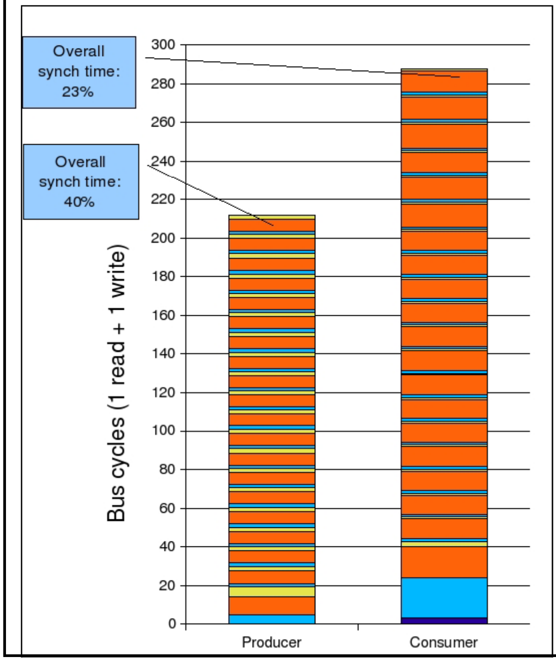
1. Multi core architectures with distributed memories (scratchpad).
2. Purely shared memory based architectures.
3. Hybrid platforms (MPARM cycle accurate simulator, different settings).

Transaction Chart

- Shares bus accesses are minimized as much as possible:
  - Local polling on scratchpad memories.
- Insertion and extraction indexes are stored into shared memory and protected by mutex.
- Data transfer section involves shared bus
  - Critical for performance.



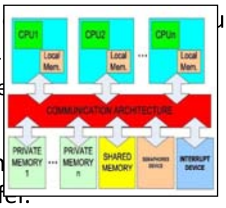
Sequence diagrams



Overall synch time: 23%

Overall synch time: 40%

- Parall… ucer and 1…
  - de…

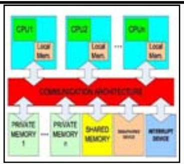


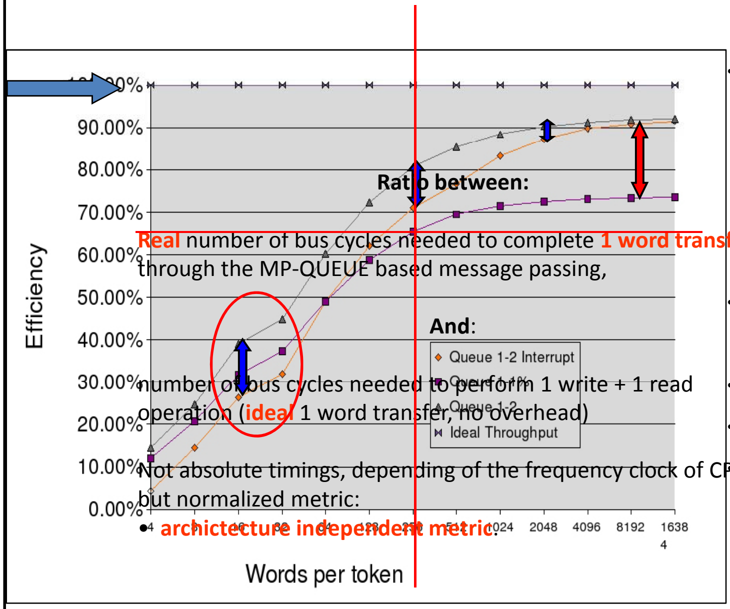
- Synch… transfer:
  - Local polling onto scratch semaphore: 🟦
  - Signaling to remote core scratch: 🟨
  - “Pure” data transfer to and from FIFO buffer in shared memory: 🟧

Message size 64 WORDS

Message size 8 WORDS

Communication efficiency





Ratio between:

Real number of bus cycles needed to complete **1 word transfer** through the MP-QUEUE based message passing,

And:

number of bus cycles needed to perform 1 write + 1 read operation (**ideal** 1 word transfer, no overhead)

Not absolute timings, depending of the frequency clock of CPU, but normalized metric:

- **architecture independent metric**.

- Comparison against ideal point-to-point communication:
  - 1-1 queue,
  - 1-N queue,
  - Interrupt based queue.

- For small messages the library synchronization overhead is prevailing,
- While for a 256 words size we reach good performance.

- Monotonicity, difference in absyntotic behaviour.
- Abrupt variation of the curve slope.

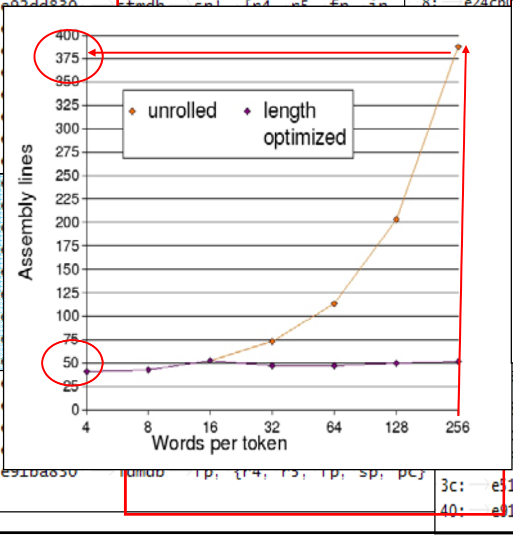
Interrupt based notification adds overhead (~15%).

Low-level optimizations are critical!

16 words per token

32 words per token

```
00000000 <main>:
   0:   e1a0c00d   mov ip, sp
   4:
   8:
   c:
  10:
  14:
  18:
  1c:
  20:
  24:
  28:
  2c:
  30:
  34:
  38:
  3c:
  40:
  44:
  48:
  4c:
  50:
  54:
  58:   e91ba830   ldmdb   fp, {r4, r5, fp, sp, pc}
```

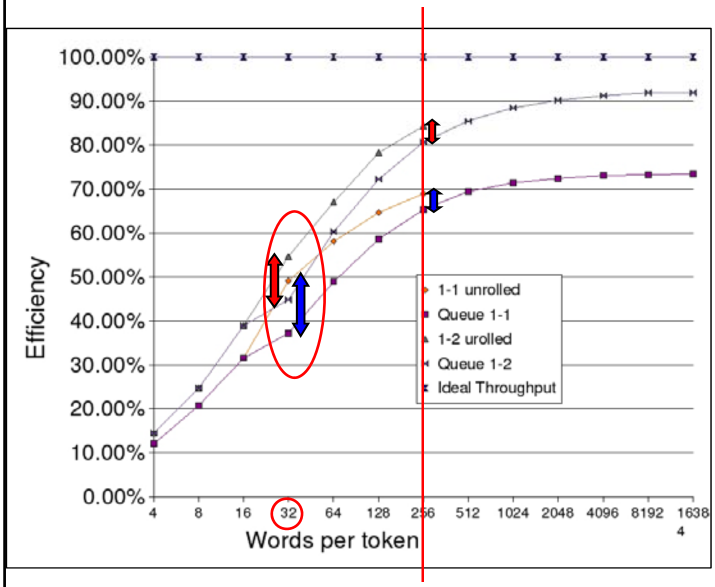


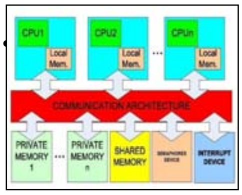
```
   0:   e1a0c00d   mov ip, sp
   4:   e92dd830   stmdb   sp!, {r4, r5, fp, ip, lr, pc}
   8:   e24cb0
```

- Gcc compiler avoids to insert the multiple load /multiple store loop from 32 words on.
- Code size would be exponentially rising.
- Up to 256 words per token we can tolerate the code size growing.

```
  fffffe   bl 0 <main>
  54074    str r4, [r5, #116]
  54004    strb    r4, [r5, #4]
  fffffe   bl 0 <main>
  3c:   e51b0094   ldr r0, [fp, -#148]
  40:   e91ba830   ldmdb   fp, {r4, r5, fp, sp, pc}
```

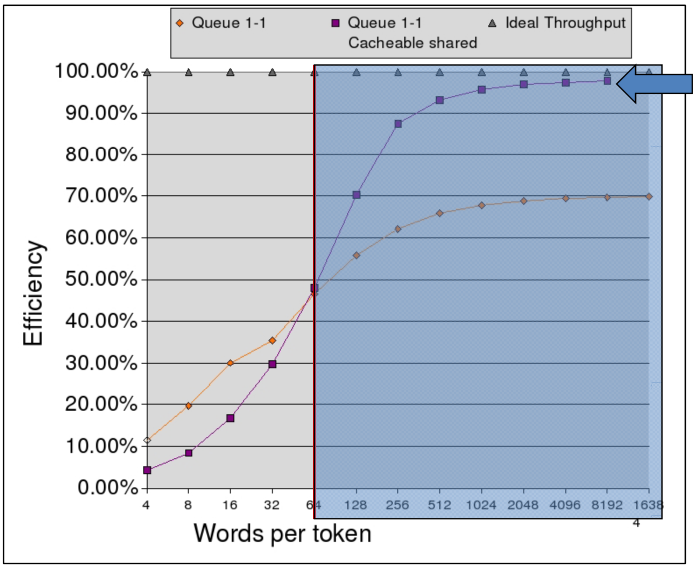
Compiler-aware optimization benefits



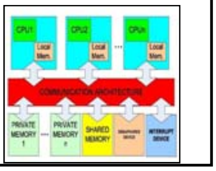


- About 15% improvement with 32 word sized messages.
- A Typical JPEG 8x8 block is encoded in a 32 word struct.
  - 8x8x16 bit data.
- Above 256 words, we don’t get any improvement by using “loop unrolling”.

Shared cacheable memory management with flush

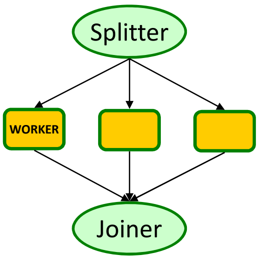


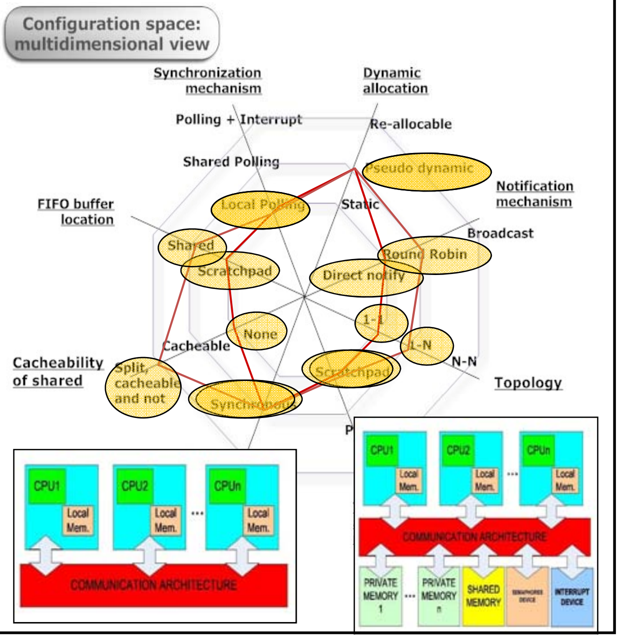
- Flushing is needed to avoid “thrashing”if no cache coherency support is given.
- With small messages, flush compromises performances.
- 64 words is the break-even size for cacheable-shared based communication.
- Efficiency is asymptotically rising to 98% !



JPEG Decoding parallelized through MP-Queue

- Starting from a sequencial JPEG decoding:
  1. Huffman extraction.
  2. Inverse DCT.
  3. Dequantization.
  4. Frame reconstruction.
- Split –Join parallelization.
- After step 1, the reconstructed 240 x 160 image is:
  – split by master core into slices;
  – delivered to N worker cores through MP-Queue messages.
- 2 different architectural templates are explored.

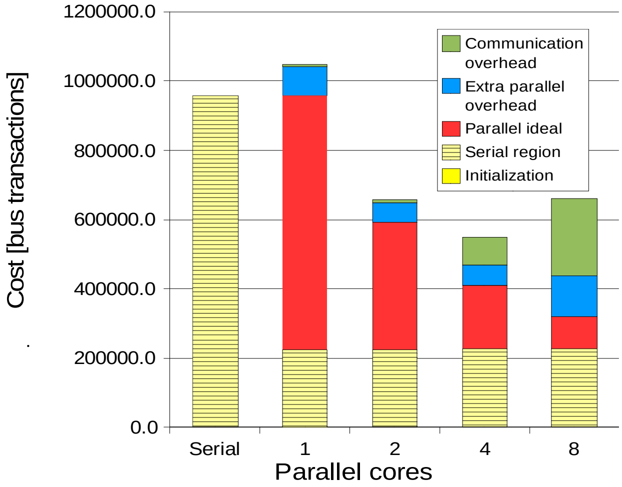


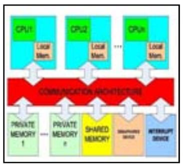


Experimental part: metrics

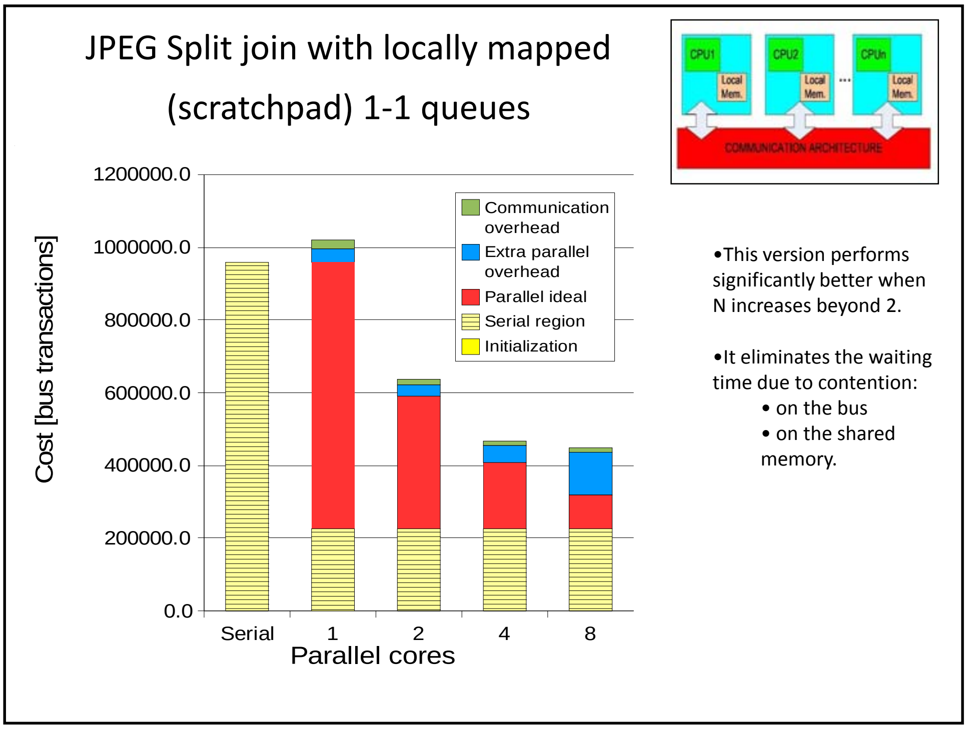
Parallel cores

JPEG Split join with one 1-N buffer in shared memory (cachable)



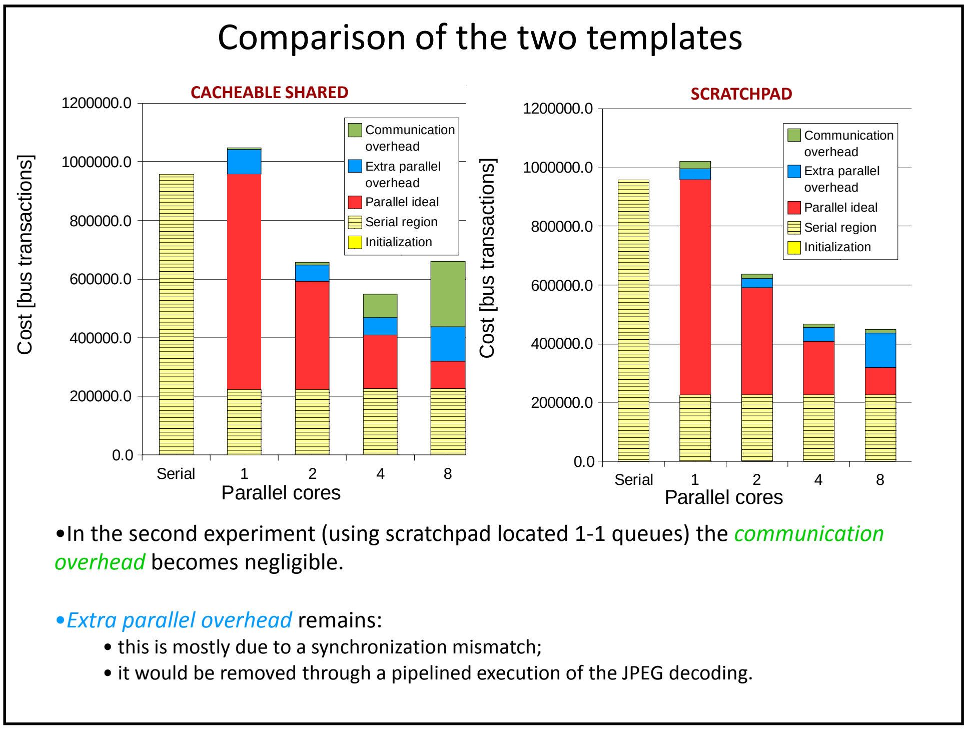


- We explored configurations with 1, 2, 4 and 8 workers (parallel cores).
- Execution time (cost in terms of bus transaction) scales down.
- Shared bus becomes quite busy while using 8 parallel cores: destination bottleneck negates speed up.



JPEG Split join with locally mapped (scratchpad) 1-1 queues

- This version performs significantly better when N increases beyond 2.
- It eliminates the waiting time due to contention:
  - on the bus
  - on the shared memory.



Comparison of the two templates

- In the second experiment (using scratchpad located 1-1 queues) the *communication overhead* becomes negligible.
- *Extra parallel overhead* remains:
  - this is mostly due to a synchronization mismatch;
  - it would be removed through a pipelined execution of the JPEG decoding.