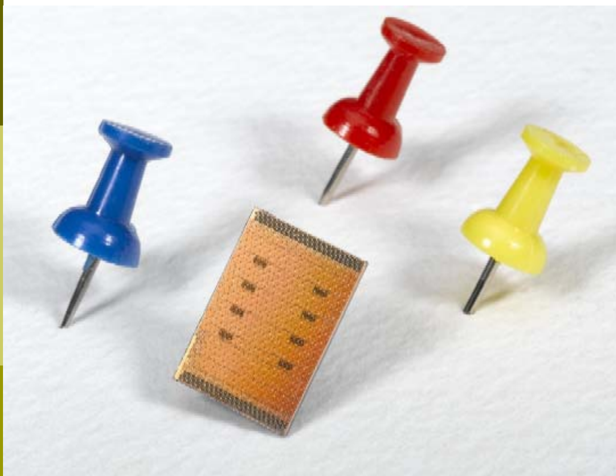
Cell



Cell STI BE

- A **multi-core** system architecture.
- It addresses:
  - Server applications:
    - Next-generation IBM Blade Servers.

  - High-performance embedded applications:
    - Gaming (Sony PS3).
    - Aerospace and defence.
    - Medical imaging.

3

Cell BE Solutions

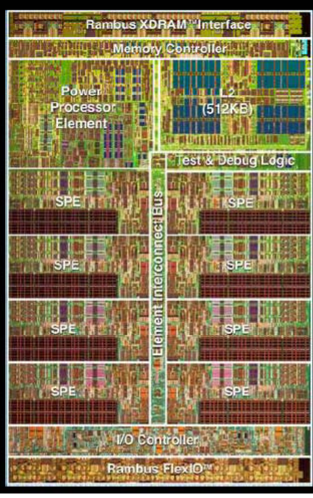
- Memory wall:
  - More slower threads
  - Asynchronous loads
- Frequency wall:
  - More slower threads
  - Specialized function
- Power wall:
  - Reduce transistor power
    - operating voltage
  - Reduce switching per function

INCREASE CONCURRENCY: Multi-Core

INCREASE SPECIALIZATION: Non-Homogeneous

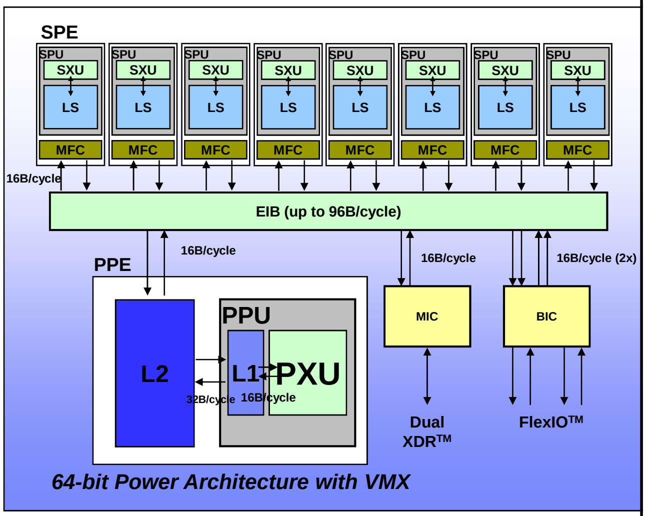
Highlights (3.2 GHz)

- **241M transistors**
- **235mm2**
- **9 cores, 10 threads**
- **>200 GFlops (SP)**
- **>20 GFlops (DP)**
- **Up to 25 GB/s memory B/W**
- **Up to 75 GB/s I/O B/W**
- **>300 GB/s EIB**
- **Top frequency >4GHz** (observed in lab)

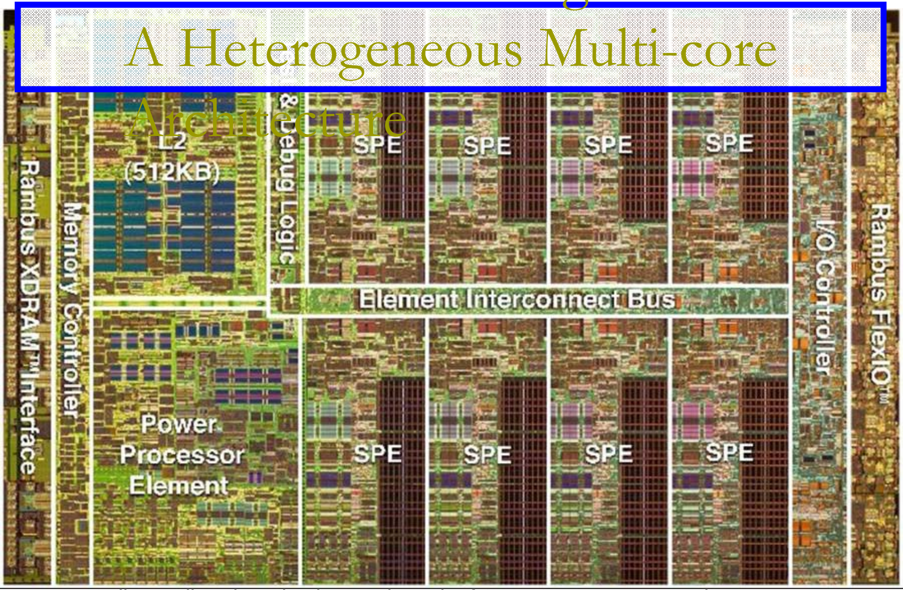


Cell System Features

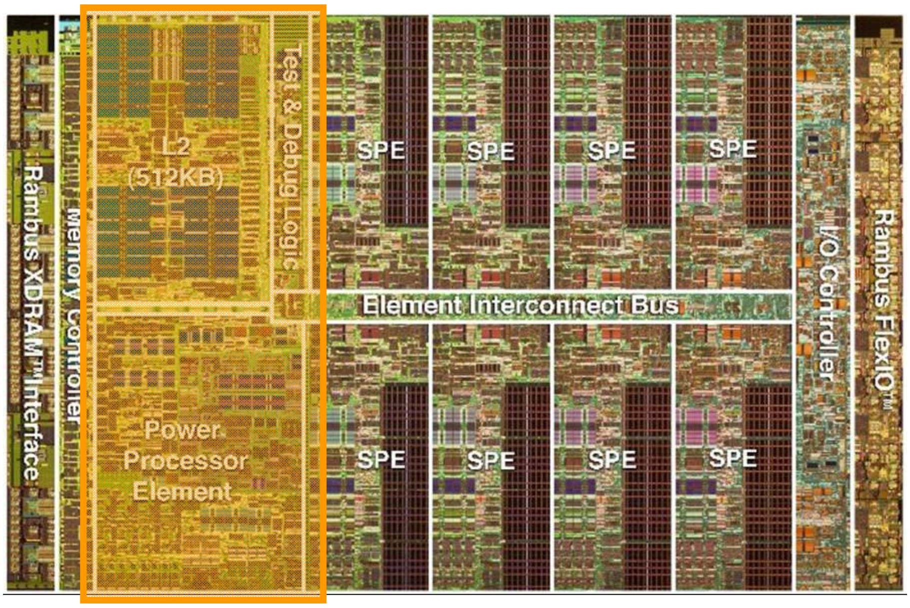
- Heterogeneous multi-core system architecture
  - Power Processor Element for control tasks
  - Synergistic Processor Elements for data-intensive processing
- Synergistic Processor Element (SPE) consists of
  - Synergistic Processor Unit (SPU)
  - Synergistic Memory Flow Control (MFC)
    - Data movement and synchronization
    - Interface to high-performance Element Interconnect Bus



Cell Broadband Engine TM: A Heterogeneous Multi-core Architecture

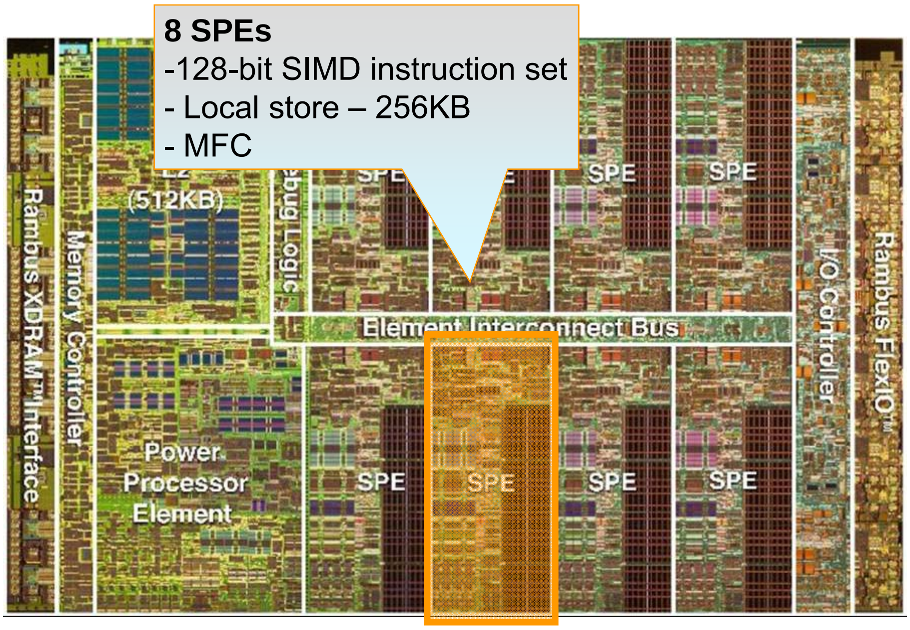


* Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc.



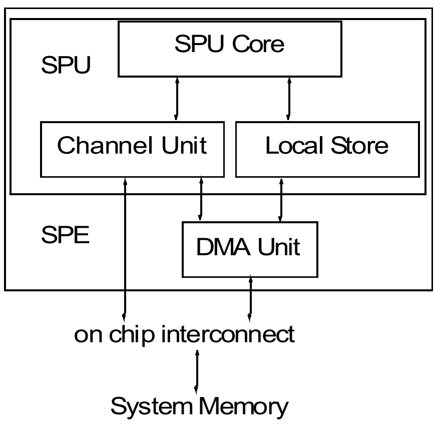
PPU Organization

Power Processor Element (PPE):
- General Purpose, 64-bit RISC Processor (PowerPC 2.02)
- 2-Way Hardware Multithreaded
- L1 : 32KB I ; 32KB D
- L2 : 512KB
- VMX (Vector Multimedia Extension)
- 3.2 GHz



SPE Organization

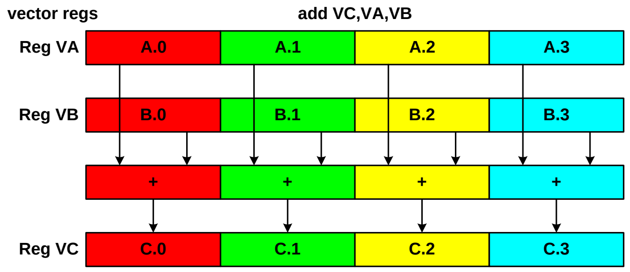
- Local Store is a private memory for program and data
- Channel Unit is a message passing I/O interface
- SPU programs DMA Unit with Channel Instructions
- DMA transfers data between Local Store and system memory



SIMD Architecture

- SIMD = "single-instruction multiple-data"
- SIMD exploits data-level parallelism
  - a single instruction can apply the same operation to multiple data elements in parallel
- SIMD units employ "vector registers"
  - each register holds multiple data elements
- SIMD is pervasive in the BE
  - PPE includes VMX (SIMD extensions to PPC architecture)
  - SPE is a native SIMD architecture (VMX-like)

A SIMD Instruction Example



- Example is a 4-wide add
  - each of the 4 elements in reg VA is added to the corresponding element in reg VB
  - the 4 results are placed in the appropriate slots in reg VC

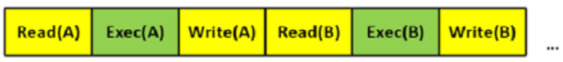
Local Store

- Never misses
  - No tags, backing store, or prefetch engine
  - Predictable real-time behavior
  - Less wasted bandwidth
  - Software managed caching
  - Can move data from one local store to another

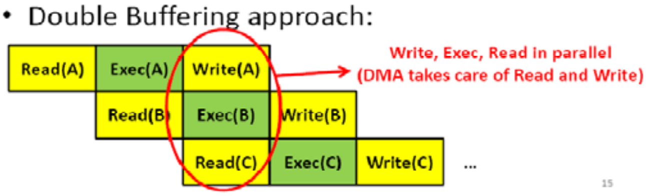
DMA & Multibuffering

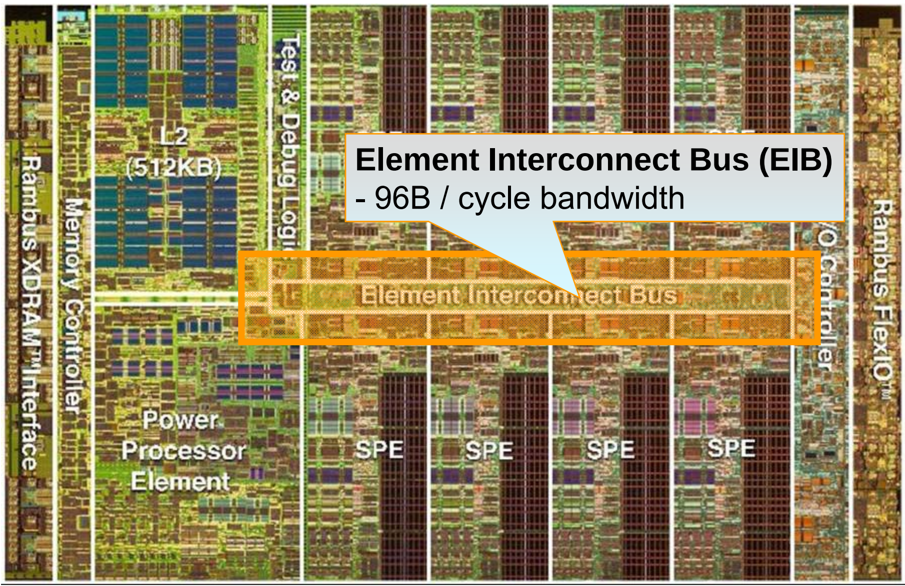
- DMA commands move data between system memory & Local Storage
- DMA commands are processed in parallel with software execution
  - Double buffering
  - Software Multithreading
- 16 queued commands

- Overlap data transfer and execution.
- Classical approach:



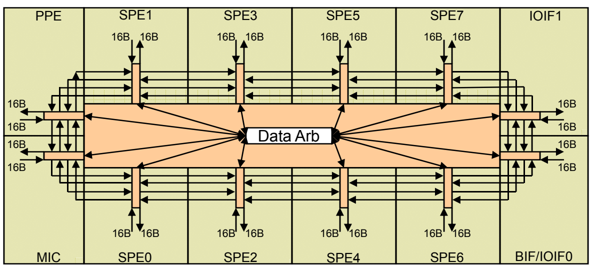
- Double Buffering approach:



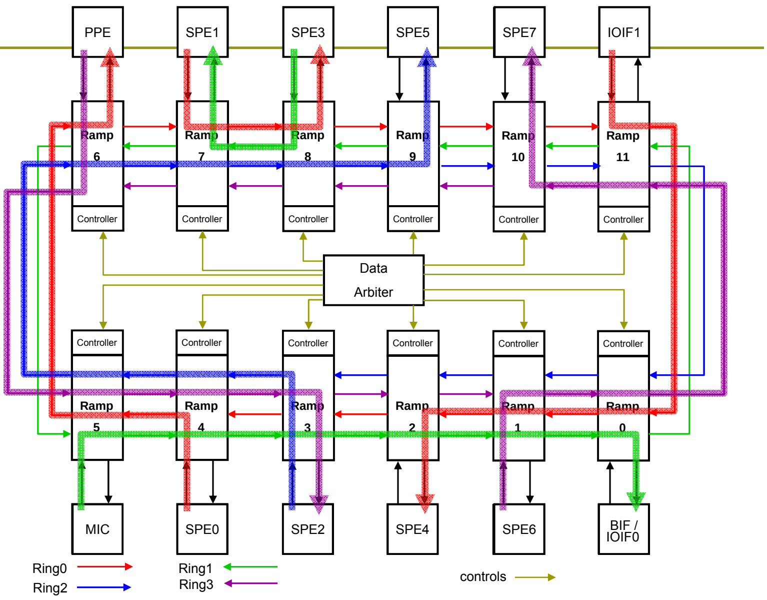


Element Interconnect Bus - Data Topology

- **Four** 16B **data rings** connecting 12 bus elements
- Physically overlaps all processor elements
- Central arbiter supports up to **3 concurrent transfers per data ring**
- Each element port simultaneously supports 16B in and 16B out data path
  - Ring topology is transparent to element data interface



Example of eight concurrent transactions



CELL Software Design Considerations

- Two Levels of Parallelism
  - Regular vector data that is SIMD-able
  - Independent tasks that may be executed in parallel
- Computational
  - SIMD engines on 8 SPEs and 1 PPE
  - Parallel sequence to be distributed over 8 SPE / 1 PPE
  - 256KB local store per SPE usage (data + code)
- Communicational
  - DMA and Bus bandwidth
    - DMA granularity – 128 bytes
    - DMA bandwidth among LS and System memory
  - Traffic control
    - Exploit computational complexity and data locality to lower data traffic requirement

Typical CELL Software Development Flow

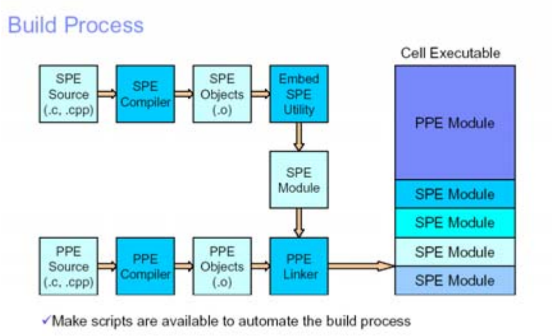
- Algorithm complexity study
- Data layout/locality and Data flow analysis
- Experimental partitioning and mapping of the algorithm and program structure to the architecture
- Develop PPE Control, PPE Scalar code
- Develop PPE Control, partitioned SPE scalar code
  - Communication, synchronization, latency handling
- Transform SPE scalar code to SPE SIMD code
- Re-balance the computation / data movement
- Other optimization considerations
  - PPE SIMD, system bottle-neck, load balance
- NOTES: Need to push all computational tasks to SPEs

Linux Kernel Support

- **PPE runs PowerPC applications and operating systems**
- **PPE handles thread allocation and resource management among SPEs**
- **PPE's Linux kernel controls the SPUs' execution of programs**
  - Schedule SPE execution independent from regular Linux threads
  - Responsible for runtime loading, passing parameters to SPE programs, notification of SPE events and errors, and debugger support
- **PPE code – Linux tasks**
  - a Linux task can initiate one or more "SPE threads"
- **SPE code – "local" SPE executables ("SPE threads")**
  - SPE executables are packaged inside PPE executable files
- **An SPE thread:**
  - is initiated by a task running on the PPE
  - runs asynchronously from initiating task
  - has a unique identifier known to both the SPE thread and the initiating task

PPE vs SPE

- **Both PPE and SPE execute SIMD instructions**
  - PPE processes SIMD operations in the VXU within its PPU
  - SPEs process SIMD operations in their SPU
- **Both processors execute different instruction sets**
- **Programs written for the PPE and SPEs must be compiled by different compilers**



Communication Between the PPE and SPEs

- **PPE communicates with SPEs through MMIO registers supported by the MFC of each SPE**
- **Three primary communication mechanisms between the PPE and SPEs**
  - **Mailboxes**
    - **Queues** for exchanging 32-bit messages
    - **Two** mailboxes are provided for sending messages **from the SPE to the PPE**
      - SPU Write Outbound Mailbox
      - SPU Write Outbound Interrupt Mailbox
    - **One** mailbox is provided for sending messages **to the SPE**
      - SPU Read Inbound Mailbox
  - **Signal notification registers**
    - Each SPE has two 32-bit signal-notification **registers**, each has a corresponding memory-mapped I/O (MMIO) register into which the signal-notification data is written by the sending processor
    - **Signal-notification** channels, or *signals*, are inbound (to an SPE) registers
    - They can be used by other SPEs, the PPE, or other devices to send information, such as a buffer-completion synchronization flag, **to an SPE**
  - **DMAs**
    - To **transfer data** between main storage and the LS

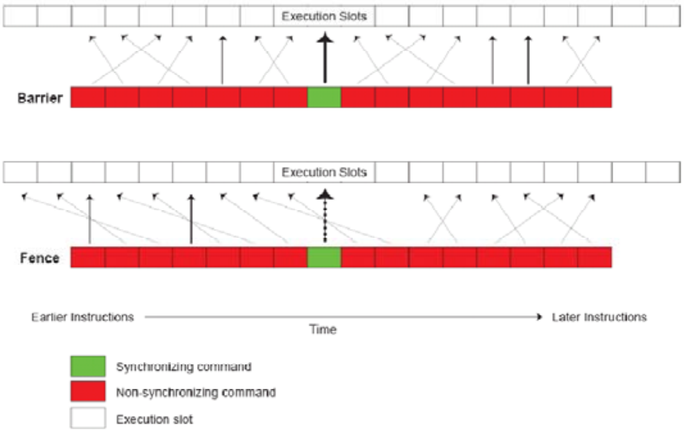
PPE and SPE MFC Command Differences

- **Code running on the SPU issues an MFC command by executing a series of writes and/or reads using channel instructions**
- **Code running on the PPE or other devices issues an MFC command by performing a series of stores and/or loads to memory-mapped I/O (MMIO) registers in the MFC**
- **Data-transfer direction for MFC DMA commands is always referenced from the perspective of an SPE**
  - get: transfer data into an SPE (from main storage to local store)
  - put: transfer data out of an SPE (from local store to main storage)

Data Transfer Between Main Storage and LS Domain

- **An SPE or PPE performs data transfers between the SPE's LS and main storage primarily using DMA transfers controlled by the MFC DMA controller for that SPE**
- **Channels**
  - Software on the SPE's SPU interacts with the MFC through channels, which enqueue DMA commands and provide other facilities, such as mailboxes, signal notification, and access auxiliary resources
- **DMA transfer requests contain both an LSA and an EA**
  - → Thus, they can address both an SPE's LS and main storage
- **Each MFC can maintain and process multiple in-progress DMA command requests and DMA transfers**
- **Each DMA command is tagged with a 5-bit Tag Group ID. This identifier is used to check or wait on the completion of all queued commands in one or more tag groups**

Barriers and Fences



Uses of Mailboxes

- **To communicate messages up to 32 bits in length, such as buffer completion flags or program status**
  - e.g., When the SPE places computational results in main storage via DMA. After requesting the DMA transfer, the SPE waits for the DMA transfer to complete and then writes to an outbound mailbox to notify the PPE that its computation is complete
- **Can be used for any short-data transfer purpose, such as sending of storage addresses, function parameters, command parameters, and state-machine parameters**

Mailboxes - Characteristics

Each MFC provides **three mailbox queues** of 32 bit each:

1. PPE ("SPU write outbound") mailbox queue
   - SPE writes, PPE reads
   - 1 deep
   - SPE stalls writing to full mailbox

2. PPE ("SPU write outbound") interrupt mailbox queue
   - like PPE mailbox queue, but an interrupt is posted to the PPE when the mailbox is written

3. SPU ("SPU read inbound") mailbox queue
   - PPE writes, SPE reads
   - 4 deep
   - can be overwritten

Hands-on: Example

- A simple hello world program

Example 1a – code sample

ppu program

hello.c

```
#include <stdio.h>
#include <libspe.h>
extern spe_program_handle_t hello_spu;
int main (void)
{
        speid_t speid;
        int status;
        speid = spe_create_thread (0, &hello_spu, NULL, NULL, -1, 0);
        spe_wait(speid, &status, 0);
        return 0;
}
```

spu program

hello_spu.c

```
#include <stdio.h>
int main(unsigned long long speid, unsigned long long argp, unsigned long long
        envp)
{
        int a=0;
        printf("Hello world (0x%llx),(0x%llx),(0x%llx)\n", speid, argp, envp);
        return a;
}
```

ppu Makefile

```
DIRS                = spu
PROGRAM_ppu     = hello
IMPORTS         = -lspe spu/hello_spu.a
include $(CELL_TOP)/make.footer
```

spu Makefile

```
PROGRAM_spu     := hello_spu
LIBRARY_embed   := hello_spu.a
include $(CELL_TOP)/make.footer
```

Hands-on: DMA

- Example 1b - Develop an SPU program to create a buffer for a "hello" message that it will DMA in, modify, and DMA back to the PPU program

Hands-on Exercise – Example 1b

- Adding to Example 1a … the following
  - Develop a PPU program that
    - Creates a buffer containing a character string "Good morning!"
    - Creates an spu thread that contains the above buffer as one of its arguments
  - Develop an SPU program that
    - Creates a local buffer to contain the data to be dma'ed in
    - Initiates a dma transfer to receive the buffer
    - Modifies the buffer into "Guten Morgen!"
    - Transfers the buffer back to the ppu

ppu program

hello.c

```
#include <stdio.h>
#include <libspe.h>
#include <libmisc.h>
#include <string.h>
extern spe_program_handle_t example1b_spu;
int main (void)
{
    speid_t speid;
    int status;
    char * buffer;
    buffer    = malloc_align(128,7);
    strcpy (buffer, "Good morning!");
    // create SPU threads
    speid = spe_create_thread (0, &example1b_spu, (void *)buffer,
        128, -1, 0);
    spe_wait(speid, &status, 0);
    printf("New modified buffer is %s\n", buffer);
    return 0;
}
```

spu program

example1bspu.c

```
#include <stdio.h>
#include <string.h>
#include <libmisc.h>
#include <spu_mfcio.h>
#define wait_on_mask(x)   mfc_write_tag_mask(x);
      mfc_read_tag_status_any();
int main(unsigned long long speid, unsigned long long argp, unsigned
      long long envp)
{
      char * buffer;
      int tag = 1, tag_mask = 1<<tag;
      // we should probably check here to make sure envp/size is a
      valid DMA size but for the sake
      // of readability I'm going to assume it is....
      printf("ARGP = 0x%llx, ENVP = %d\n", argp, envp);
      buffer = malloc_align(envp, 7);
      // receive buffer
      mfc_get(buffer, (unsigned int)argp, envp, tag, 0, 0);
      wait_on_mask(tag_mask);
      printf("SPE 0 received buffer \"%s\"\n", buffer);
      // modify buffer
      strcpy (buffer, "Guten Morgen!");
      // DMA out buffer to SPE 1
      mfc_put(buffer, (unsigned int)argp, envp, tag, 0, 0);
      wait_on_mask(tag_mask);
      return 0;
}
```

Parallel Application Development toolkit with Run-Time Support for MPSoC Processors

Martino Ruggiero
martino.ruggiero@unibo.it

Luca Benini
luca.benini@unibo.it

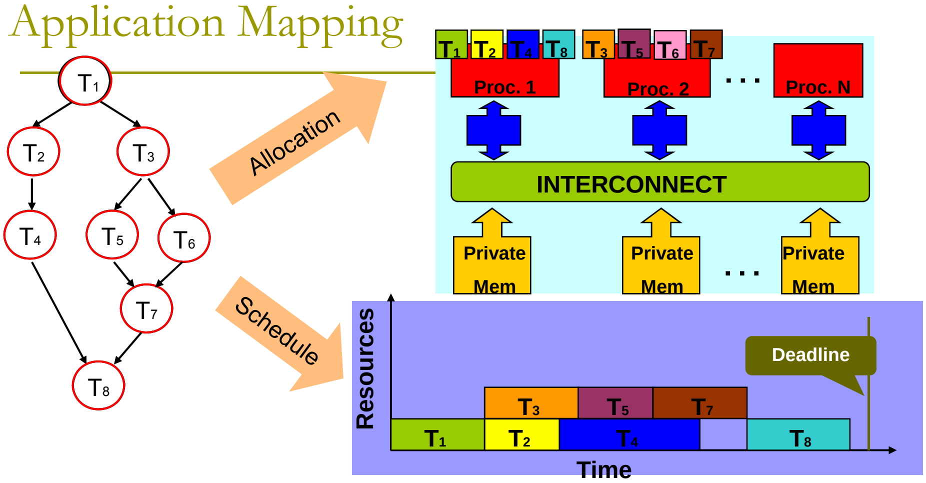
University of Bologna
Italy

Enhancing Programmability Efficiency on Multi Processor System-on-Chip Platforms

- Providing support for multimedia applications on MPSoC platforms remains a significant research challenge.
- Challenges in:
  - The design of applications
  - Programmability efficiency

- New tools for efficient developing and mapping of applications onto hardware platforms
  - Easy to use
  - Flexibility
- The problem of allocating and scheduling task graphs on processors in a distributed real-time system is NP-hard.

Application Mapping



- New tool flows for efficient developing and mapping of multi-task applications onto hardware platforms
- The problem of allocating and scheduling task graphs on processors in a distributed real-time system is NP-hard.

Related Work

Main approaches:

- **Incomplete:**
  - Low computational cost;
  - No guarantees about the quality of the final solution;
- **Complete:**
  - Mainly based on Integer Linear Programming;
  - High computational cost;
  - Suitable for small problems instances;
- **Problem decomposition:**
  - Good way to tackle problem complexity;
  - Divide up the problem into sub-problems & leverage their structures;
  - Mainly heuristic approach.

Our approach

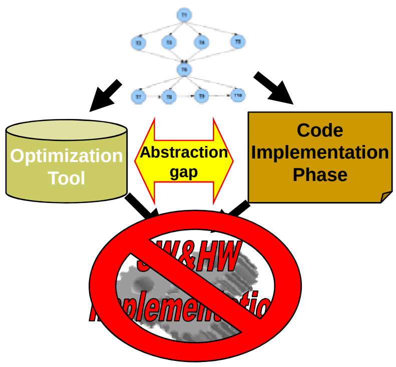
Our Focus:
- **Statically Scheduled Task Graph Applications**

Our Objective:
- **Complete approach to allocation and scheduling;**
- **High computational efficiency w.r.t. commercial solvers;**
- **High accuracy of generated solutions;**

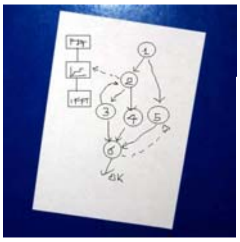
Our Methodology:
- **Problem decomposition;**
- **Allocation Sub-problem:**
  - Integer Programming.
- **Scheduling Sub-problem:**
  - Constraint Programming.

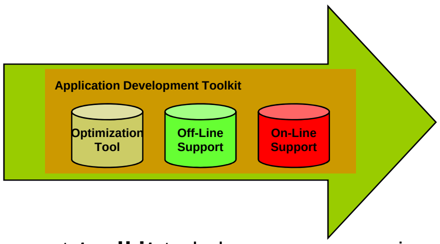
Application Developing: Design Flow

- Optimization tools need abstraction to model:
  - Application
  - Programming model
  - Hardware platform
- The abstraction gap between high level optimization tools and standard application programming models can introduce unpredictable and undesired behaviours.



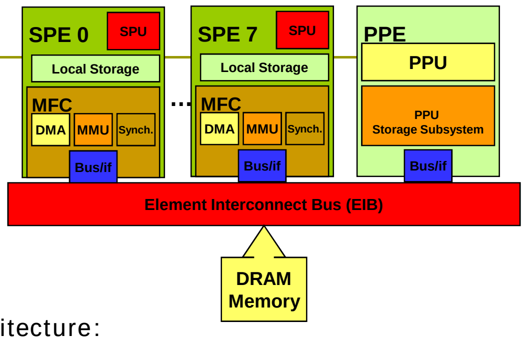
The main idea





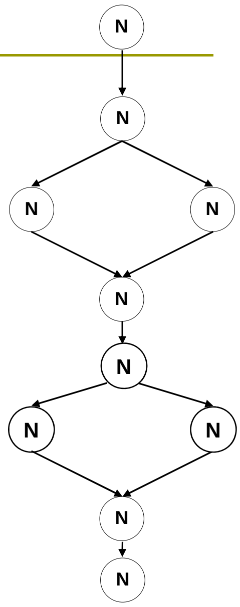
- A software development **toolkit** to help programmers in software implementation
- Starting from a high level task and data flow graph, software developers can easily and quickly build their **application infrastructure**.
- Programmers can intuitively translate high level representation into C-code using our facilities and library
  - a generic customizable **application template → OFFLINE SUPPORT**;
  - a set of high-level **APIs → ONLINE SUPPORT in RT-OS**
- The main goals are:
  - guarantees on **high performance** and **constraint satisfaction**;
  - **predictable** application execution after the optimization step.

Target architecture



- **Heterogeneous** system architecture:
  - One 64-bit Power Processor Element (PPE) → Unary Resource
  - 8 Synergistic Processing Elements (SPEs) → Unary Resource
  - Element Interconnect Bus → Cumulative Resource
  - Limited Local Memory → Cumulative Resource
- **Several MPSoC platforms available on the market match this template:**
  - Data processing engine similar to our homogeneous tile vector with explicit memory management

Target Application: Task Graph (TG)

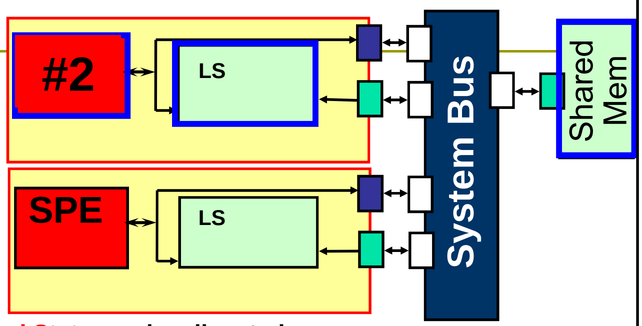


- Statically scheduled Task Graph Applications:
  - Explicit parallelism;
  - Message Passing Communication;
  - Single-token Communication.
- A **TG** is a couple **<T,A>**, where:
  - **T** is the set of **nodes** modelling generic tasks (e.g. elementary operations, subprograms, ...);
  - **A** the set of **arcs** modelling precedence constraints (e.g. due to data communication);
  - WCET for Comp. & Comm. Modelling.

Task memory requirements

Each task has three kinds of memory requirements:

- Program Data;
- Internal State;
- Communication queues.



Program Data & Internal State can be allocated:

- On the local LS;
- On the remote Shared Memory.

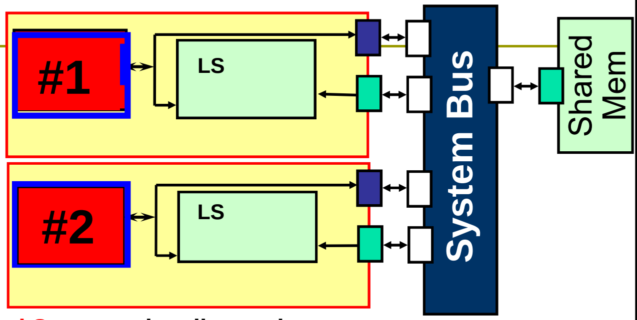
The communication task might run:

- On the same SPE → negligible communication cost
- On a remote SPE → costly read or write procedure
- On Shared Memory → costly message exchange procedure

- Communication queues in LS → more efficient message passing
  - **Memory size limit!**

Task memory requirements

Each task has **three** kinds of **memory requirements**:
- Program Data;
- Internal State;
- Communication queues.



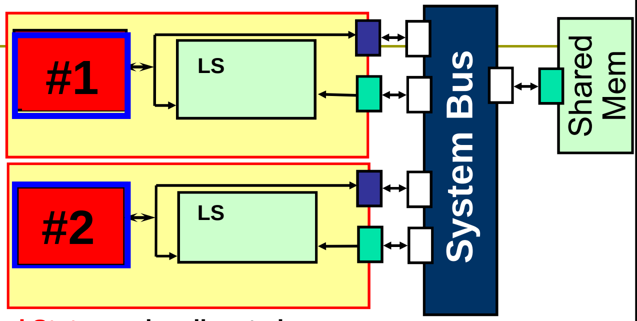
Program Data & Internal State can be allocated:
- On the local LS;
- On the remote Shared Memory.

The communication task might run:
- On the same SPE → negligible communication cost
- On a remote SPE → costly read or write procedure
- On Shared Memory → costly message exchange procedure

- Communication queues in LS → more efficient message passing
  - **Memory size limit!**

Task memory requirements

Each task has **three** kinds of **memory requirements**:
- Program Data;
- Internal State;
- Communication queues.

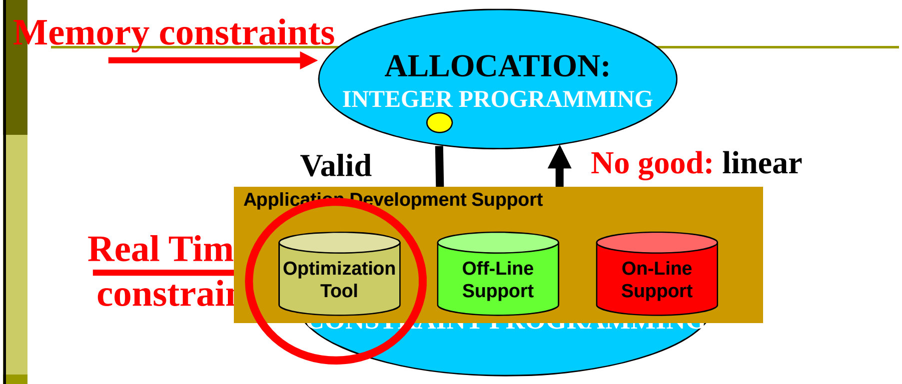


Program Data & Internal State can be allocated:
- On the local LS;
- On the remote Shared Memory.

The communication task might run:
- On the same SPE → negligible communication cost
- On a remote SPE → costly read or write procedure
- On Shared Memory → costly message exchange procedure

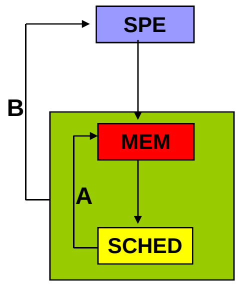
- Communication queues in LS → more efficient message passing
  - **Memory size limit!**

Logic Based Benders Decomposition



- Decomposes a problem into 2 sub-problems:
  - Allocation → IP
  - Scheduling → CP
- The process continues until the master problem and sub-problem converge providing the same value.
- Methodology has been proven to converge to the optimal solution [J.N.Hooker and G.Ottosson].

Multi-stage Benders Decomposition



- When the SCHED problem is solved (no matter if a solution has been found), one or more cuts (labelled A) are generated to forbid (at least) the current memory device allocation and the process is restarted from the MEM stage;
- if the scheduling problem is feasible, an upper bound on the value of the next solution is also posted.
- When the MEM & SCHED sub-problem ends (either successfully or not), more cuts (labelled B) are generated to forbid the current task-to-SPE assignment.
- When the SPE stage becomes infeasible the process is over converging to the optimal solution for the problem overall.

SPE Allocation

Given a graph with **n** tasks, **m** arcs and a platform with **p** processing Elements

$$\min z$$

$$s.t.$$

$$z \geq \sum_{i=0}^{n-1} T_{ij} \forall j = 0, \ldots, p-1$$

Needed to express the objective function

$p-1$

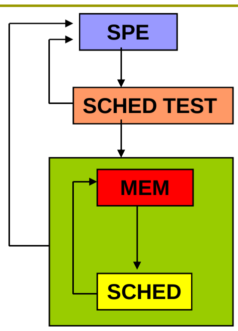
The **makespan** objective function depends only on scheduling decision variables.

We adopt an ***heuristic* objective function**:
to spread tasks as much as possible on different SPEs, which often provides good makespan values pretty quickly.
It forces the objective variable z to be
greater than the number of tasks allocated on any PE.

DMIN(i) is the minimum possible duration of task i and dline is a deadline

Constraints on the total duration of tasks on a single SPE.
To discard trivially infeasible solutions.

Schedulability test



- SPE allocation choices are by themselves very relevant:
  - a bad SPE assignment is sometimes sufficient to make the scheduling problem unfeasible.
- if the given allocation with **minimal task durations** is already infeasible for the scheduling component, then it is useless to complete it with the memory assignment that cannot lead to any feasible solution overall.

Memory device allocation

$M_i \in \{0,1\} \qquad \forall i = 0,\dots,n-1$

$W_r \in \{0,1\} \qquad \forall r = 0,\dots,m-1$

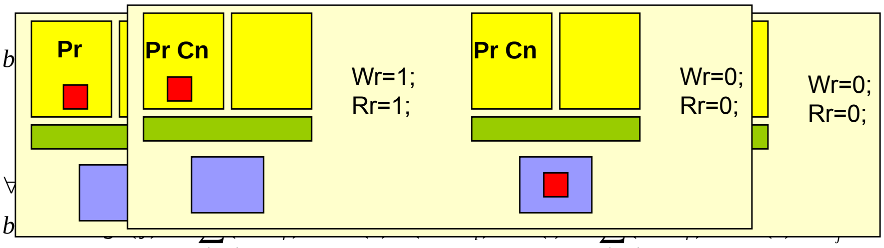
$R_r \in \{0,1\} \qquad \forall r = 0,\dots,m-1$

Mi = 1 if task i allocates its computation data on the local memory of the SPE it is assigned to

Wr = 1 if the communication buffer is on SPE pe(h) (that of the producer), Rr = 1 if the buffer is on SPE pe(k) (that of the consumer).

$R_r + W_r$ …

$R_r = W_r \qquad if \;\; pe(h) = pe(k)$



Memory device allocation

$M_i \in \{0,1\} \qquad \forall i = 0,\dots,n-1$

$W_r \in \{0,1\} \qquad \forall r = 0,\dots,m-1$

$R_r \in \{0,1\} \qquad \forall r = 0,\dots,m-1$

$R_r + W_r$ …

$R_r =$ …

mem(i) is the amount of memory required to store internal data of task I;
comm(r) is the size of the communication buffer associated to arc r.
The **base_usage(j)** expression is the amount of memory needed to store all data **permanently** allocated on the local device of processor j.

$$base\_usage(j) = \sum_{\substack{a_r=(-,t_h)\\ pe(h)=j}} comm(r)R_r + \sum_{i=0}^{p-1} mem(i)Mi + \sum_{\substack{a_r=(t_h,t_k)\\ pe(k)=j\\ pe(h)\neq pe(k)}} comm(r)W_r$$

$$\forall j = 0,\dots,p-1; \qquad \forall i \quad s.t. \quad pe(i) = j:$$

$$base\_usage(j) + \sum_{a_r=(-,t_h)} (1-R_r)comm(r) + (1-M_i)mem(i) + \sum_{a_r=(-,t_h)} (1-W_r)comm(r) \le C_j$$

Scheduling subproblem

Each communication buffer must be written before it can be read.

$$\forall l = 0,\ldots,h-2 \qquad end(rd_{rl}) = start(rd_{rl+1})$$

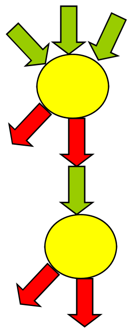
$$end(rd_{rh-1}) = start(exec_i)$$

$$end(exec_i) = start(wr_{rh})$$

$$\forall l = h,\ldots,k-2 \qquad end(wr_{rl}) = start(wr_{rl+1})$$

$\forall r = 0,.$

- An activity for each:
  - execution phase (exec)
  - buffer reading/writing operation (rd,wr).
- Task are not preemptive;



Rd	Rd	Rd	Exec	Wr	Wr				
						Rd	Exec	Wr	Wr

Computational Efficiency

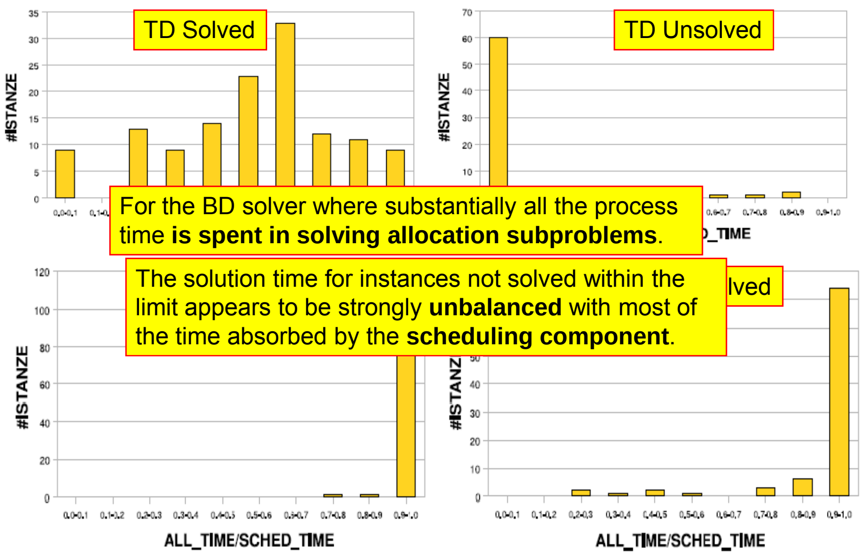
	TD			BD		Timed out		
ntasks	SPE it.	MEM it.	time	PM it.	time	$TD \wedge BD$	$\neg TD \wedge BD$	$TD \wedge \neg BD$
10-11	12	13	3.95	12	71.10	0	0	0
12-13	17	21	11.59	13	151.38	0	1	0
14-15	19	28	14.78	14	145.19	0	0	0
16-17	29	38	42.61	18	388.89	0	2	0
18-19	46	70	245.17	28	863.00	1	5	0
20-21	70	90	665.35	23	1291.90	4	8	0
22-23	33	69	1304.92	19	1686.00	12	6	1
24-25	29	42	1486.15	8	1623.00	11	4	3
26-27	18	41	1523.50	4	1701.67	12	4	3
28-29	13	19	1800.00	3	1721.00	19	0	1

Table 1. Performance tests

- Up to the 20 − 21 group, TD is much more efficient than BD.
- Starting from group 22−23, the high number of timed out instances biases the average execution time.
- TD is doing considerably better until group 24 − 25.
  - After that, most instances are not solved within the time limit by any of the approaches
- TD has a lower execution time, despite it generally performs more iterations than BD:
  - TD works by solving many easy sub-problems
  - BD performs fewer and slower iterations.

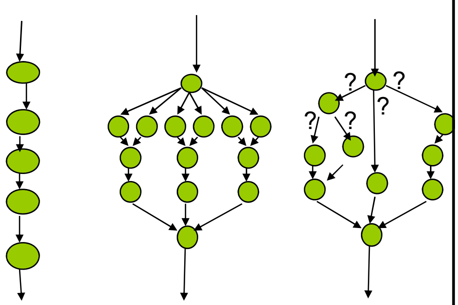
Computational Efficiency:

distribution of the allocation/scheduling time ratio for the solvers

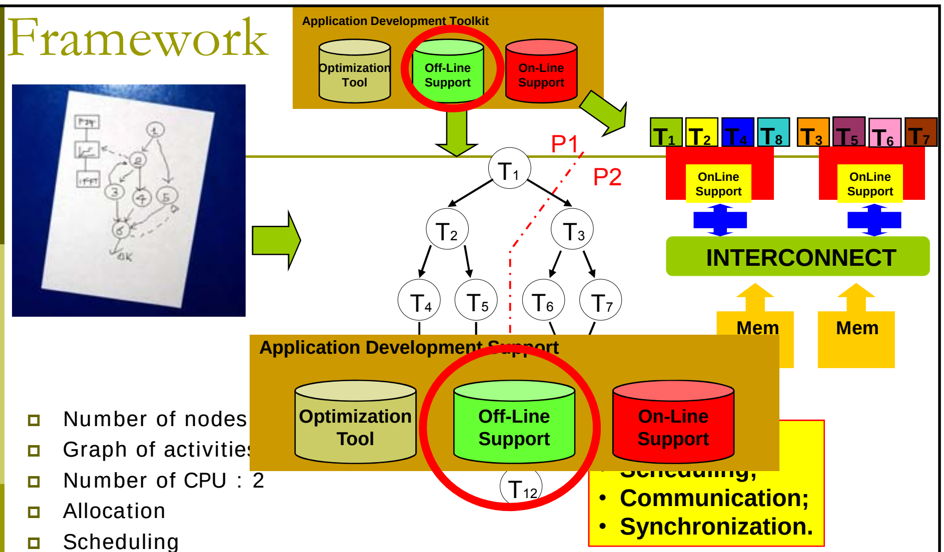


Optimization Models

- Application model:
  - Pipeline
  - Generic Task Graph
  - Conditional Task graph
- Objective Function:
  - Throughput
  - Makespan
  - Power
- Constraints:
  - Memory occupation
  - Frequency Switching
  - Bus Traffic



Framework



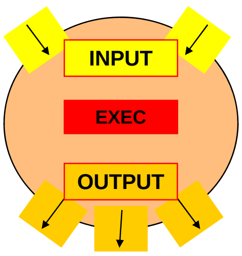
- Number of nodes
- Graph of activities
- Number of CPU : 2
- Allocation
- Scheduling

uint queue_consumer [ ] [ ] = {

Available scheduling engines:
- **optimum** static scheduler based on integrated IP and CP solvers (Benders decomposition)
- fast & **suboptimal** list-based scheduler for dynamic scheduling

Task Computational Model Vs Generated Code

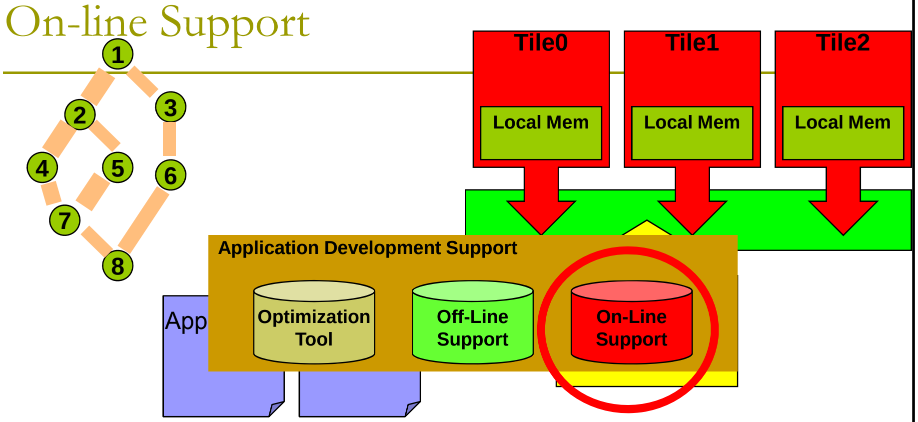
INPUT EXEC OUTPUT



//Initialization
Init_task_structures(); //Task State and Private Data Init.
Init_queues(); //Buffers and Semaphores Init.
…
//Task Core
//Reading phase
Read_input();
….

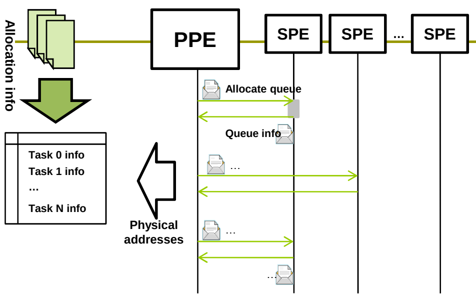
//Task Execution
Exec(); //The only section which is up to //the application programmer
….
//Writing phase
Write_output();

On-line Support

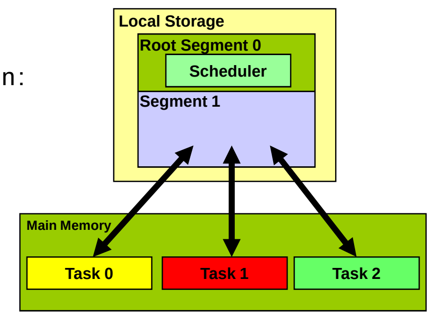


- The **Task Graph** and the **allocation** are described by data structures:
  - At application boot time, the allocation support reads these data structures and allocates resources to tasks and communication buffers.

On-line Support

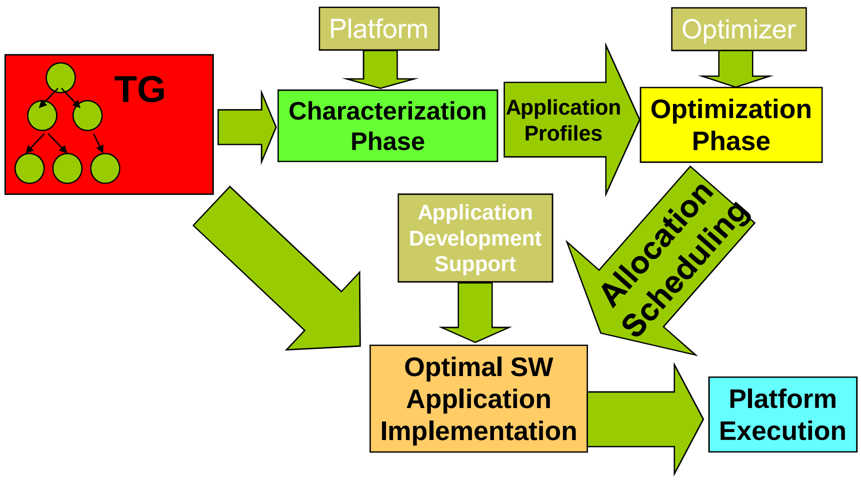


- No-OS on SPEs:
  - Local scheduling support
- Local Memory Limitation:
  - Code overlays

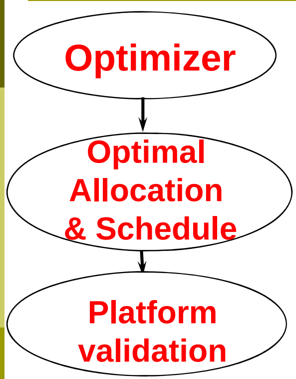


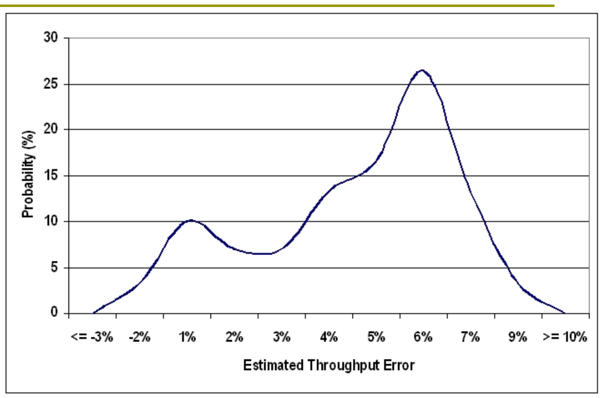
- Automatic Application Configuration:
  - Mailbox system
  - Task Generation and Configuration
  - Buffer allocations

Methodology



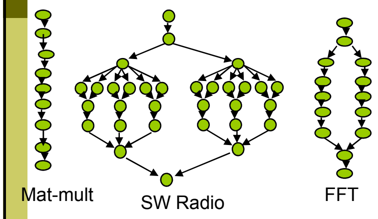
Validation of optimizer solutions

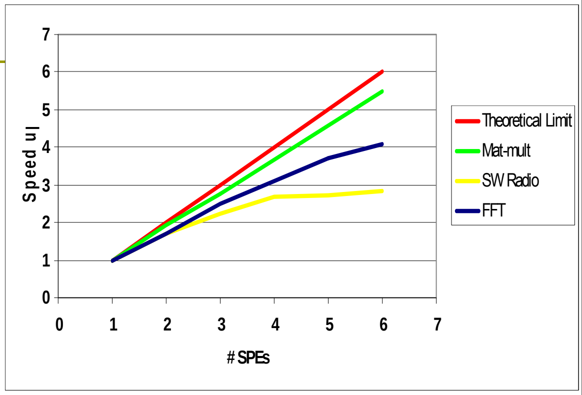




- Throughput comparison between the predicted by the optimizer and the real one;
- MAX error lower than 10%;
- AVG error equal to 4.8%, with standard deviation of 2.41;

Cellflow: Experimental results





- The Mat-mult benchmark scales almost perfectly:
  - efficiency of the runtime environment
  - almost negligible overhead.
- Good speed-ups also for the FFT ;
- The software radio benchmark shows good speedup until only three SPEs:
  - A critical path limits the performance boost
    - Functional pipeline case.

Ongoing work

- Dynamic resource management
- Performance tuning of the middleware
- Interaction with high level tools:
  - OpenMP support
  - Full dataflow (i.e. streaming) support
  - Model-based environments
    - Matlab
- Integration with advanced architectures
  - NoC support for dataflow;
  - Transactional Memories;
  - Etc.

Progettino: Cellsim

Progettino: plug-in eclipse

Progettino: Parallelizzare 3D graphic kernels

Thank you!

Contacts:

martino.ruggiero@unibo.it

luca.benini@unibo.it