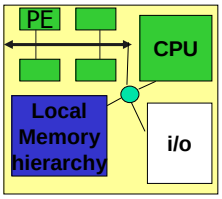
Luca Benini – DEIS Università di Bologna
lbenini@deis.unibo.it

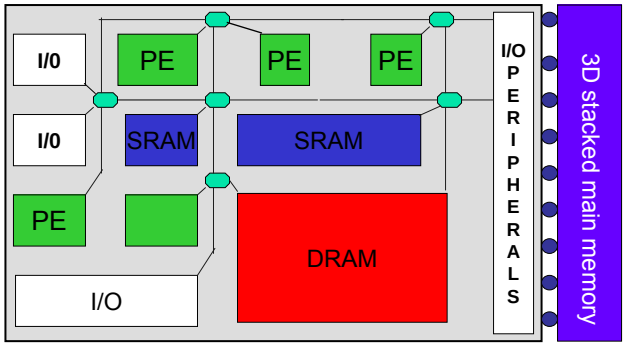
# Part III – MPSoC SW Platforms

- Software challenges-
- System software – middleware
- Industrial standardization intiatives
- Case studies



# Architecture Evolution



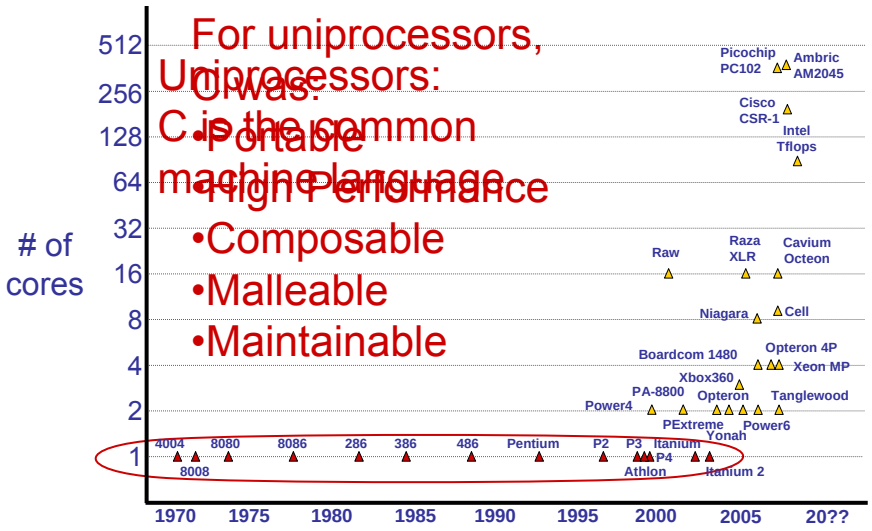




- Roadmap continues: 90→65→45 nm
- "Traditional" Bus based SoCs fit in one tile !!
- Communication demand is staggering, but unevenly distributed, because of architectural heterogeneity

# Multicores Are Here!

For uniprocessors, C was:
- Portable
- High Performance
- Composable
- Malleable
- Maintainable

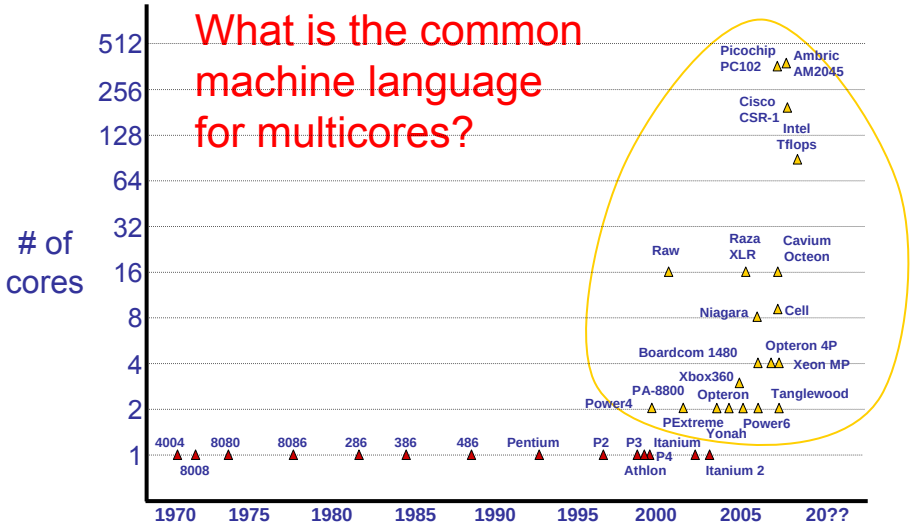
Uniprocessors:
C is the common machine language






# Multicores Are Here!

What is the common machine language for multicores?



# Common Machine Languages

Uniprocessors:

| Common Properties |
| --- |
| Single flow of control |
| Single memory image |

| Differences: |
| --- |
| Register File |
| ISA |
| Functional Units |

Register Allocation

Instruction Selection

Instruction Scheduling

von-Neumann languages represent the common properties and abstract away the differences

Multicores:

| Common Properties |
| --- |
| Multiple flows of control |
| Multiple local memories |

| Differences: |
| --- |
| Number and capabilities of cores |
| Communication Model |
| Synchronization Model |

Need common machine language(s) for multicores



# Embedded vs. General Purpose

## Embedded Applications

- *Asymmetric Multi-Processing*
  - Differentiated Processors
- Specific tasks known early
  - Mapped to dedicated processors
- Configurable and extensible processors: performance, power efficiency
- Communication
  - Coherent memory
  - Shared local memories
  - HW FIFOS, other direct connections
- Dataflow programming models
- Classical example – Smart mobile – RISC + DSP + Media processors

## Server Applications

- *Symmetric Multi-Processing*
  - Homogeneous cores
- General tasks known late
  - Tasks run on any core
- High-performance, high-speed microprocessors
- Communication
  - large coherent memory space on multi-core die or bus
- SMT programming models (Simultaneous Multi-Threading)
- Examples: large server chips (eg Sun Niagara 8x4 threads), scientific multi-processors

## Parallelism & Programming Models

- MP is difficult: Concurrency, and "Fear of Concurrency"
- No robust and general models to automatically extract concurrency in 20-30+ years of research
- Many programming models/libraries - SMT, SMP
  - OpenMP, MPI (message passing interface)
  - Users manually modify code
    - Concurrent tasks or threads
    - Communications
    - Synchronisation
- Today:
  - Coarse-grained (whole application/data-wise) concurrency – unmodified source + MP scheduler
  - API for communications and synchronisation

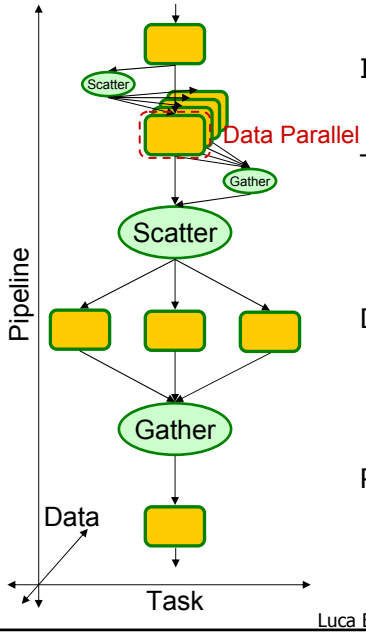


## Sequential execution model

- The most common
  - Supported by traditional (imperative) languages (C, C++, Fortran, etc.)
  - Huge bulk of legacy code
- The most well understood
  - We are trained to solve problems algorithmically (sequence of steps)
  - Microprocessors have been originally designed to run sequential code
- The easiest to debug
  - Tracing the state of the CPU
  - Step-by-step execution

**But… it HIDES parallelism!!**

# Types of Parallelism




Instruction Level Parallelism (ILP)
- Compilers & HW are mature

Task Parallelism
- Parallelism explicit in algorithm
- Between filters *without* producer/consumer relationship

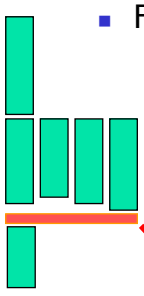
Data Parallelism
- Between iterations of a *stateless* filter
- Place within scatter/gather pair (*fission*)
- Can't parallelize filters with state

Pipeline Parallelism
- Between producers and consumers
- *Stateful* filters can be parallelized

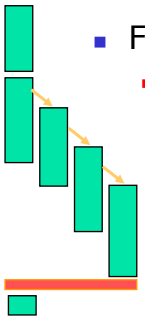


# Parallelizing Loops: a Key Problem

- 90% of execution time is in loops
- Partial success in automatic extraction
  - Mostly "well-behaved loops"
  - Challenges: dependency analysis & interaction with data placement
- Cooperative approaches are common
  - The programmers drives automatic parallelization (openMP)



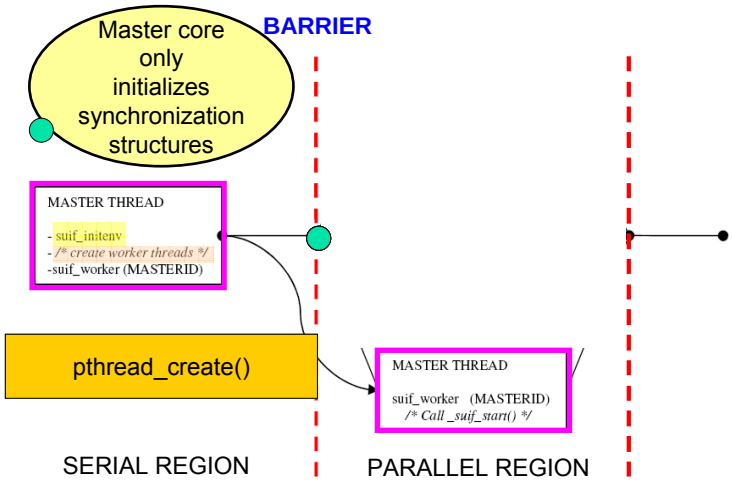
- FORALL
  - No "loop carried dependences"
  - Fully parallel



- FORACROSS
  - Some "loop carried dependences"

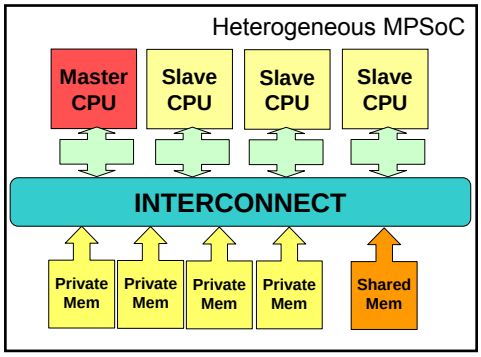
Parallelized loops rely on **Barrier Synchronization**

# Barrier with Pthreads






# Pthreads on Heterogeneous CPUs?



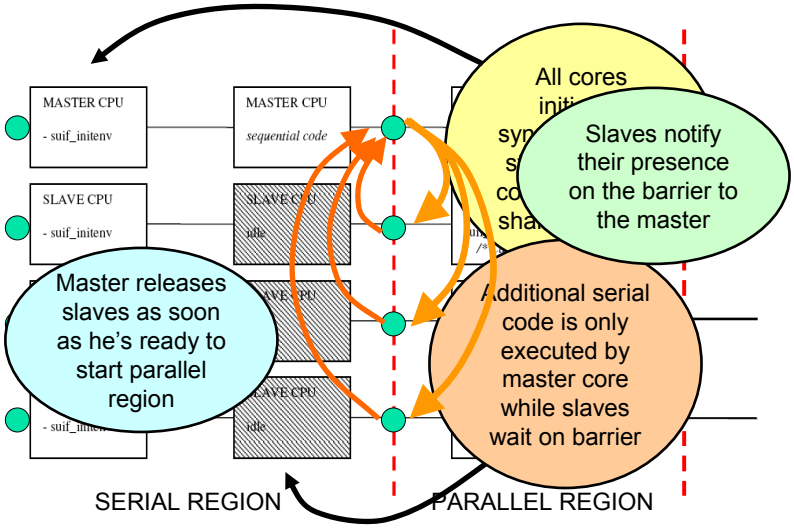

**ISSUES**

- There is an OS running on each core.
- No means for the master core to fork new threads on slave nodes.
- Use of pthreads is not a suitable solution.

**SOLUTION**

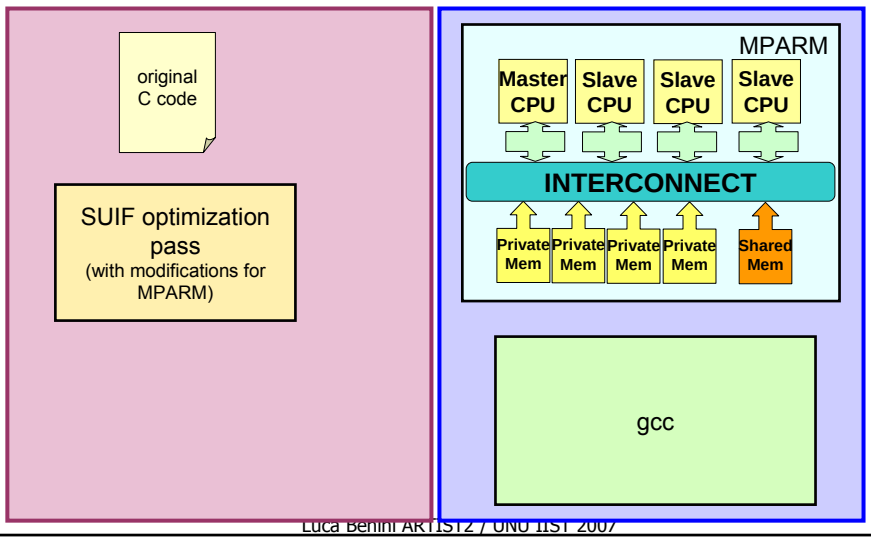
- Standalone implementation.
- Master/Slave cores instead of threads.
- Synchronization through shared memory.

# SPMD Barrier






# OptComp – Backend SDK bridge



# Runtime Library

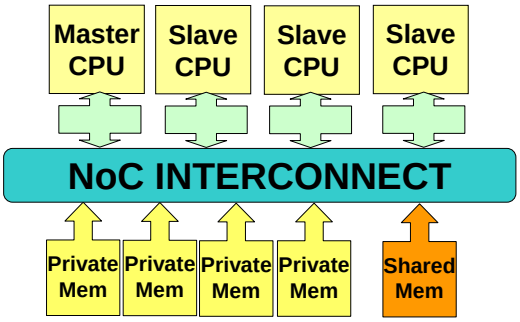
The Runtime Library is responsible for

- Initializing needed synchronization features, creating new worker threads (in the original implementation) and coordinating their parallel execution over multiple cores
- Providing implementation of synchronization facilities (locks, barriers)



# Code Execution

- Each CPU execute the same program. Basing upon the CPU id we separate portions of code to be executed by master and slaves.
- Master CPU executes serial code, initializes synchronization structures in shared memory, etc..
- Slave CPUs only execute the parallel regions of code, behaving like the typical slave threads




```
int main() {
  …
  if (MASTERID) {
    serial code
    synchronization
  }
  …
  if (SLAVEID) {
    parallel code
  }
  …
}
```

Synchronization structures

(barriers, locks)

# Synchronization

- Parallel programming through shared memory requires global and point-to-point synchronization
- Original implementation uses *pthreads* library synchronization facilities, MPARM library uses hw semaphores

```
void lock(pthread_mutex_t *lock)
{
   pthread_mutex_lock(lock);
}

void unlock(pthread_mutex_t *lock)
{
   pthread_mutex_unlock(lock);
}
```

```
void lock(int *lock)
{
   while(*lock);
}

void unlock(pthread_mutex_t *lock)
{
   *lock = 0;
}
```



# Typical Barrier implementation

```
LOCK(bar->lock);
bar->entry_count++;
if (bar->entry_count < nproc) {
        UNLOCK(bar->lock);
        while(bar->entry_count != nproc);
        LOCK(bar->lock);
        bar->exit_count++;
        if (bar->exit_count == nproc)
                bar->entry_count = 0x0;
        UNLOCK(bar->lock);
} else {
        bar->exit_count = 0x1;
        if (bar->exit_count == nproc)
                bar->entry_count = 0x0;
        UNLOCK(bar->lock);
}
while(bar->exit_count != nproc);
```

```
struct barrier {
        lock_type lock;
        int entry_count;
        int exit_count;
} *bar;
```

Shared counters protected by locks

# Barrier Implementation Issues

**ISSUES**

- This approach is not very scalable
- Every processor notifies its arrival to the barrier increasing the value of a common shared variable
- As the number of cores increases contention for the shared resource may increase significantly

**POSSIBLE SOLUTION**

- A vector of flags, one per each core, instead of a single shared counter



# New Barrier Implementation

```
typedef struct Barrier {
  int entered[NSLAVES];
  int usecount;
} Barrier;
```

```
void Master_Wait (
  int i;
```

procs ) {

No busy-wa
due to conter
of a shared
counter. Each
slave updates its
own flag

Only the slaves
spin wait on a
shared counter
that is updated by
the master

Only the master
spin waits on
each slave's flag
to detect their
presence on the
barrier

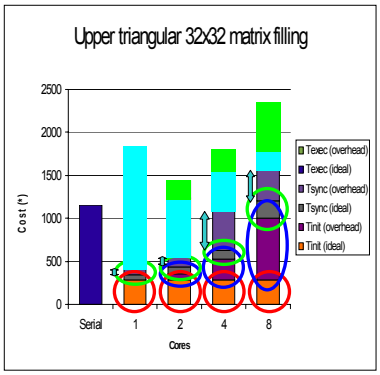
```
void Master_Release(Barrier b) {
  b->usecount++;
}
```

# Compiler aware of synchronization cost?

- A lightweight implementation of the synchronization structures allows a parallelized code with a big number of barrier instruction to still perform better than the serial version
- It would be useful to let the compiler know about the cost of synchronization. This would allow it not only to select the parallelizable loops, but also to estabilish if the parallelization is worthwhile
- For well distributed workloads across the cores the proposed barrier performs well, but for a high degree of load imbalance an interrupt-based implementation may be best suited. The compiler may choose which barrier instruction to insert depending on the amount of busy waiting



# Performance analysis

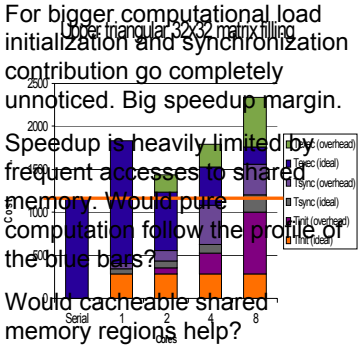



- Ideal parallel execution time was estimated for the single core configuration, taking the best core single execution time
- As expected, the overhead with the single processor case is the sum of work and idle cycles.
- Simultaneous accesses to the shared memory generate bus traffic and the overhead increases with the number of cores.
- Overall execution time decreases for all the configurations, but the speed-up is far from ideal because of the synchronization structures in shared memory.

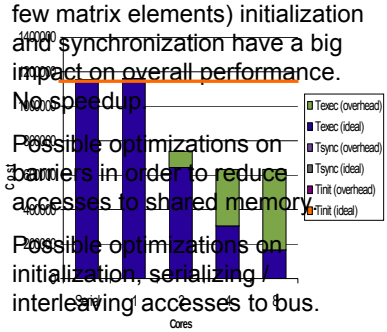
(*) Overall number of cycles normalized by the number of cycles spent for an ideal bus transaction (1 read + 1 write)

# Performance analysis 2

- For bigger computational load initialization and synchronization contribution go completely unnoticed. Big speedup margin.
- Speedup is heavily limited by frequent accesses to shared memory. Would pure computation follow the profile of the blue bars?
- Would cacheable shared memory regions help?




- For small computational load (i.e. few matrix elements) initialization and synchronization have a big impact on overall performance. No speedup.
- Possible optimizations on barriers in order to reduce accesses to shared memory.
- Possible optimizations on initialization, serializing / interleaving accesses to bus.



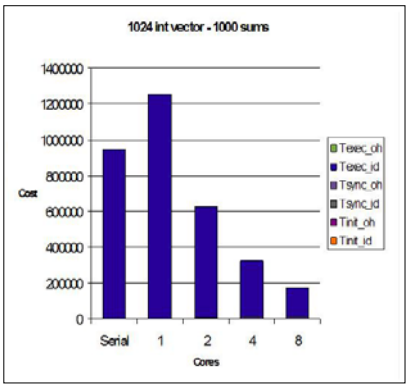



# Performance analysis 3

Enabling caching of shared (read only) data

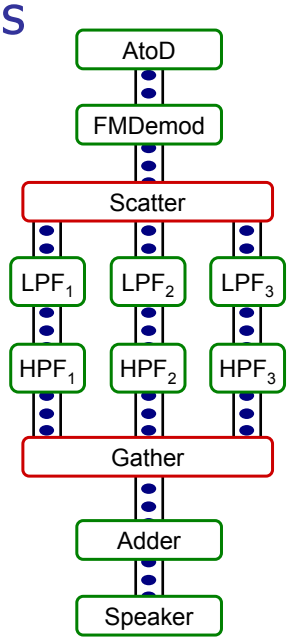
```
for (i=0; i<ITER; i++)
    for (j=SIZE*ID/nprocs;
         j<SIZE*(ID+1)/nprocs; j++)
        tmp += A[j];
```

- Allowing shared data to be cached improves performance but…
- Cache coherency has to be granted by means of software flushes after a barrier instruction



# Dataflow/Streams

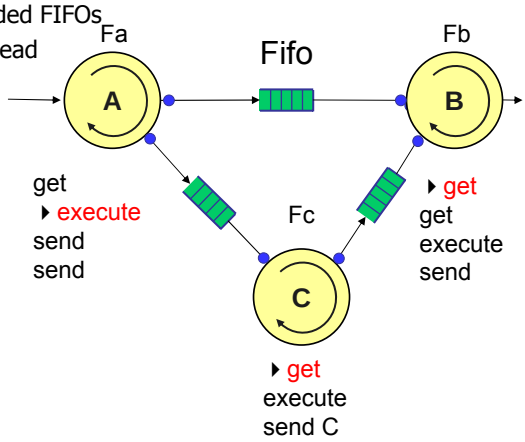
- Different model of computation
  - Regular and repeating computation
- Independent processes (filters) with explicit communication
  - Segregated address spaces and multiple program counters
- Natural expression of Parallelism:
  - Producer / Consumer dependencies
  - Enables powerful, whole program transformations
- Expressing control flow is difficult






# Kahn Process Network (KPN)

- Formal model for DF [Kahn 1974][Parks&Lee 95]
  - Processes run autonomously
  - Communicate via unbounded FIFOs
  - Synchronize via blocking read
- Process is either
  - **executing (execute)**
  - **communicating (send/get)**
- Characteristics
  - Deterministic
  - Distributed Control
    - No Global Scheduler is needed
  - Distributed Memory
    - No memory contention




**NATURAL MATCH for NoC Architecture**

# Example: MP-Queue library

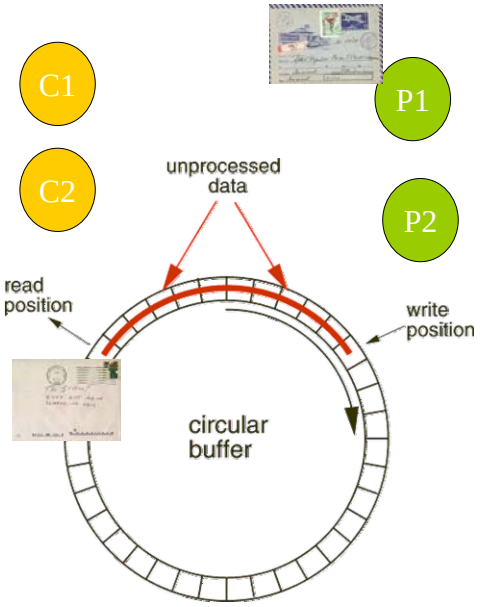
- MP Queue is a library intended for message passing among different cores in a MPSoc environment.
- Highly optimized C implementation:
  - Low level exploitation of data structures and semaphores:
    - low overhead;
    - data transfer optimized for performance:
      - analyses of disassembled code;
    - synch operations optimized for minimal interconnect utilization
- Producer onsumer paradigm, different topologies:
  - 1-N
  - N-1
  - N-N



# Communication library API

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).

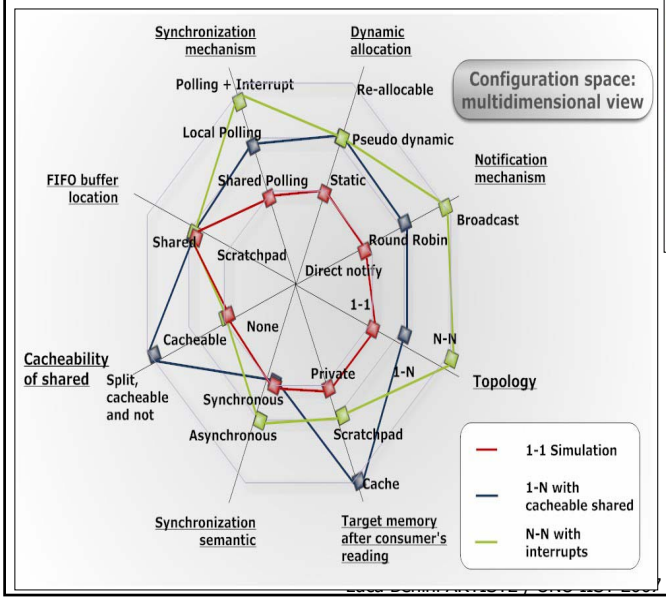
# Communication semantics

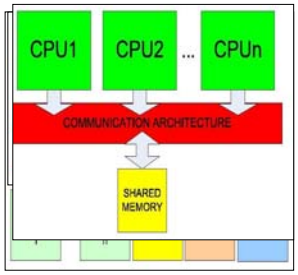



- Notification mechanisms available:
  - Round robin.
  - Notify all.
  - Target core specifying.
- The i-th producer:
  - Gets the write position index.
  - Puts data onto the buffer.
  - Signals either one consumer (round-robin / fixed) or all consumers (notify all).
- The i-th consumer:
  - Gets the read position index.
  - Gets data from the buffer.
  - Signals either one producer (round-robin / fixed) or all producers (notify all).



# Architectural Flexibility

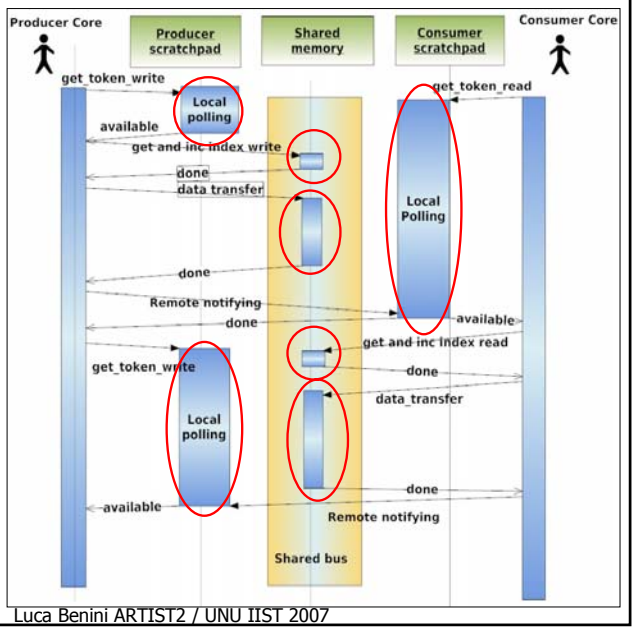



1. Multi core architectures with distributed memory.
2. Purely shared memory based architectures.
3. Hybrid platforms

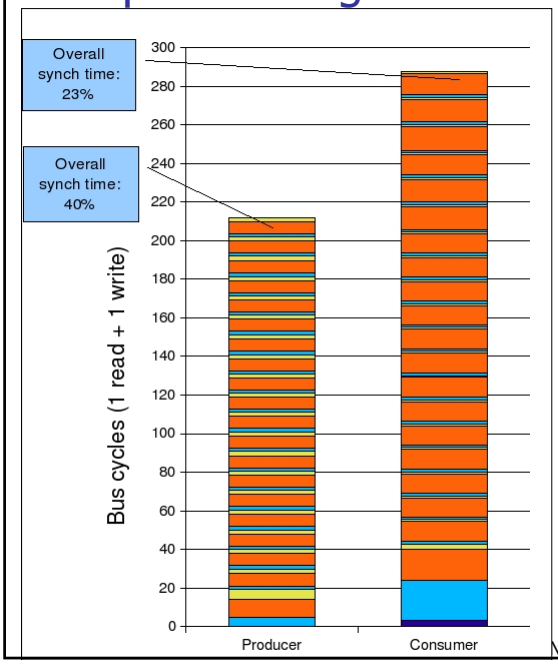
# Transaction Chart

- 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.





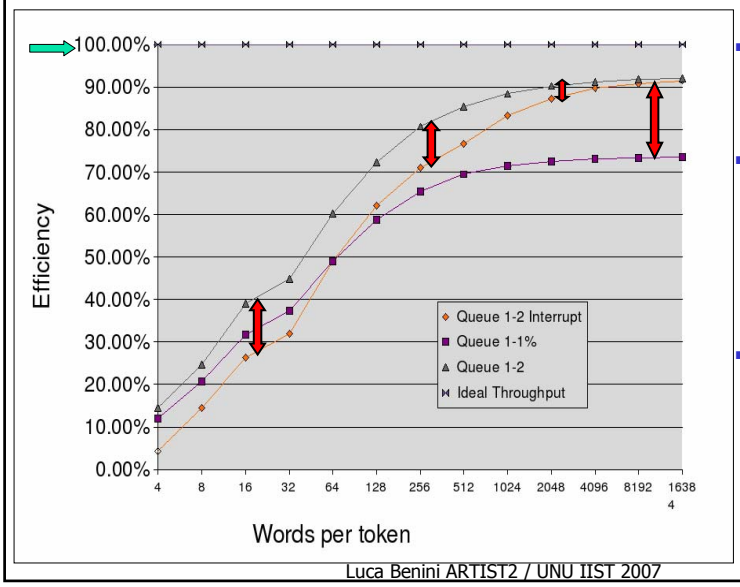

# Sequence diagrams




- 1 producer and 1 consumer (parallel activity).
- Synch time vs pure data 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

# Communication efficiency




- Comparison against ideal point-to-point communication.
- 1-N queues leverages bus pipelining:
  - bigger asymptotic efficiency.
- Interrupt based notification allows more than one task per core.
  - significant overhead (up to 15%).



# Low-level optimizations are critical!

16 words per token

```
00000000 <main>:
   0:   e1a0c00d    mov ip, sp
   4:   e92dd830    stmdb   sp!, {r4, r5, fp, ip,
   8:   e3a0547f    mov r5, #2130706432 ; 0x7f000
   c:   e3a04001    mov r4, #1  ; 0x1
  10:   e24cb004    sub fp, ip, #4  ; 0x4
  14:   e24dd080    sub sp, sp, #128    ; 0x80
  18:   e24be094    sub lr, fp, #148    ; 0x94
  1c:   e5c54000    strb    r4, [r5]
  20:   e5854070    str r4, [r5, #112]
  24:   e8be000f    ldmia   lr!, {r0, r1, r2, r3}
  28:   e24bc054    sub ip, fp, #84 ; 0x54
  2c:   e8ac000f    stmia   ip!, {r0, r1, r2, r3}
  30:   e8be000f    ldmia   lr!, {r0, r1, r2, r3}
  34:   e8ac000f    stmia   ip!, {r0, r1, r2, r3}
  38:   e8be000f    ldmia   lr!, {r0, r1, r2, r3}
  3c:   e8ac000f    stmia   ip!, {r0, r1, r2, r3}
  40:   e89e000f    ldmia   lr, {r0, r1, r2, r3}
  44:   e88c000f    stmia   ip, {r0, r1, r2, r3}
  48:   e5854074    str r4, [r5, #116]
  4c:   e5c54004    strb    r4, [r5, #4]
  50:   ebfffffe    bl  0 <main>
  54:   e51b0054    ldr r0, [fp, -#84]
  58:   e91ba830    ldmdb   fp, {r4, r5, fp, sp, pc}
```

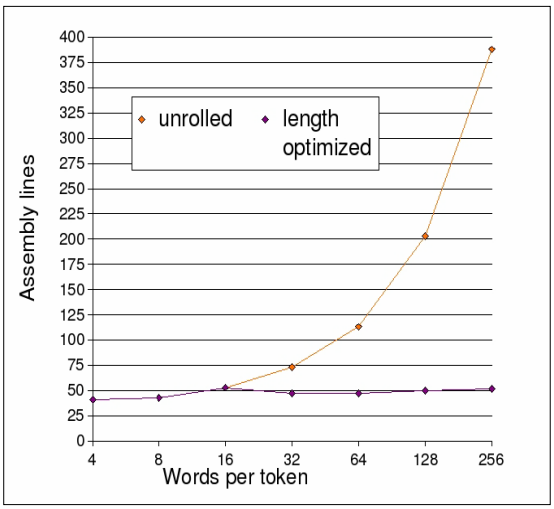
32 words per token

```
   0:   e1a0c00d    mov ip, sp
   4:   e92dd830    stmdb   sp!, {r4, r5, fp, ip, lr, pc}
   8:   e24cb004    sub fp, ip, #4  ; 0x4
   c:   e3a0547f    mov r5, #2130706432 ; 0x7f000000
  10:   e3a04001    mov r4, #1  ; 0x1
  14:   e24dd001    sub sp, sp, #256    ; 0x100
  18:   e24b1f45    sub r1, fp, #276    ; 0x114
  1c:   e3a02080    mov r2, #128    ; 0x80
  20:   e5c54000    strb    r4, [r5]
  24:   e24b0094    sub r0, fp, #148    ; 0x94
  28:   e5854070    str r4, [r5, #112]
```

Not produced any more by compiler!!!

```
  2c:   ebfffffe    bl  0 <main>
  30:   e5854074    str r4, [r5, #116]
  34:   e5c54004    strb    r4, [r5, #4]
  38:   ebfffffe    bl  0 <main>
  3c:   e51b0094    ldr r0, [fp, -#148]
  40:   e91ba830    ldmdb   fp, {r4, r5, fp, sp, pc}
```

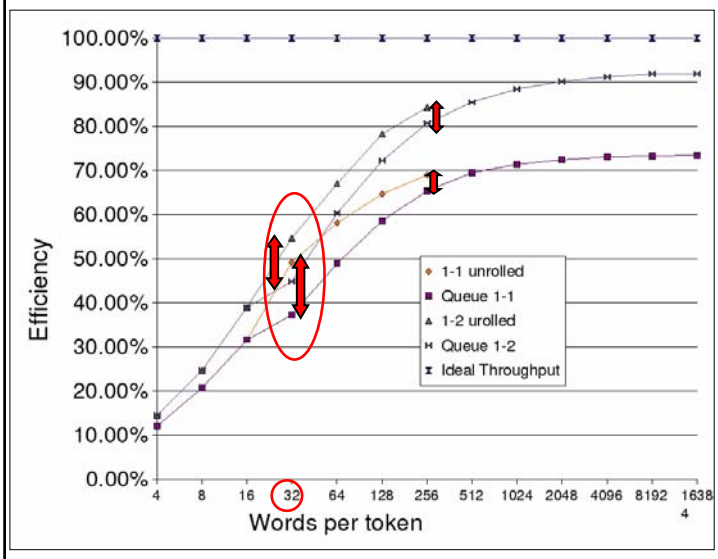
# Growth of assembly length for copy sections




- Gcc compiler avoids to insert the multiple load /multiple store loop from 32 words on.
- Code size would be exponentially rising.
- Where high throughput is required, a less compact but more optimized representation is desired.

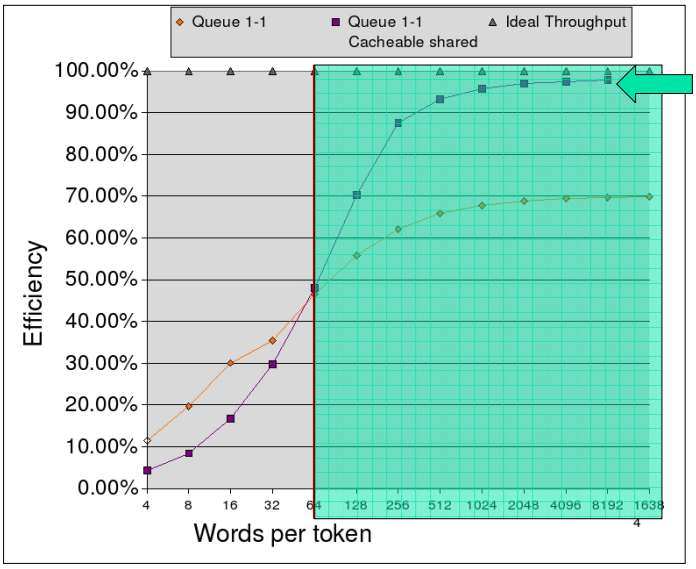


# Compiler-aware optimization benefits




- Compiler may be forced to unroll data transfer loops.
- About 15% improvement with 32 word sized messages.
- A Typical JPEG 8x8 block is encoded in a 32 word struct.
  - 8x8x16 bit data.

## Shared cacheable memory management with flush




- Flushing is needed to avoid "thrashing".
- With small messages, flush compromises performance.
- 64 words is the break-even size for cacheable-shared based communication.
- Efficiency is asymptotically rising to 98% !

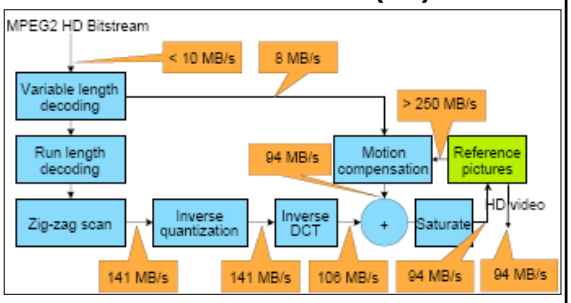


# Data bandwidth bottleneck

[Philips07]

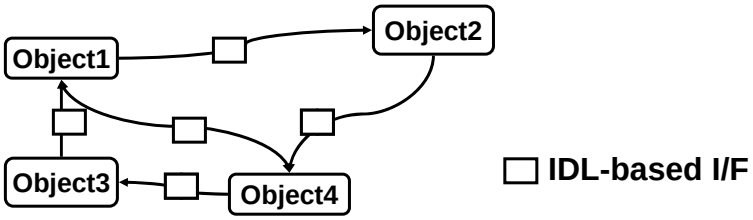
- Frame memory size = width x height x bits/pixel
  - WVGA aRGB overlay: 848 x 480 x 4 ~= 1.62 MB/frame
  - SDTV YUV4:2:0 8bit: 720 x 576 x 1.5 ~= 0.62 MB/frame
  - HDTV YUV4:2:0 8bit: 1920 x 1080 x 1.5 ~= 3.1 MB/frame
- Trends
  - Resolution 1920x1080p standard for TV
  - Longer term 2500*1600 and quad HD
  - Refresh rate 120Hz
  - Color resolution YUV4:4:4
  - Increasing bit depth 10,12,16
- Huge frame size → Ext DRAM
- DRAM port becomes the architectural bottleneck!!

**MPEG2 Decode (HD)**



# Distributed objects

[Paulin TVLSI06]



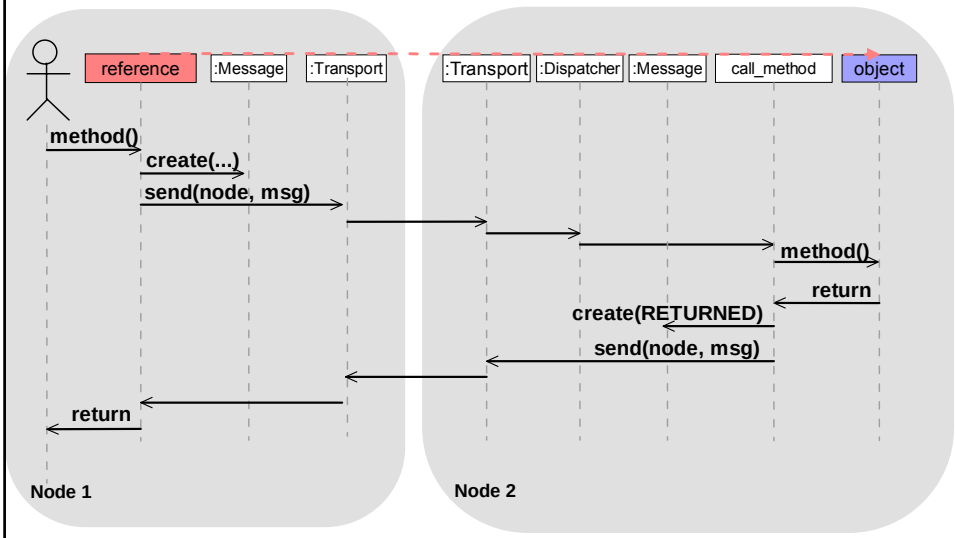

- Distributed Object Modeling: e.g. CORBA, DCOM
- Objects represent application functionality
  - Granularity of simple function to complete sub-system
- Inter-object communication via standard interface
  - Use of lightweight IDL (Interface Description Language)
  - Strong communication-computation decoupling: more flexibile than DF!
- No assumptions on subsequent assignment of processing engines (H/W or S/W), interconnect, O/S



# What are distributed objects?

- Natural extension of OOP to distributed system
  - Call methods on remote objects as well as local
  - Remote call mediated by message passing
  - Marshal method id, object id, parameters into request
  - Update parameters with response
- Message passing layer generated automatically from interface specification
  - Maps between messages and method calls
- Generalized object reference
  - includes location as well as address
  - Can be copied between nodes

# Remote Method Call






# Example: Redplain's redFOX

- Flexibility & convenience of distributed objects
- Suitable for Board/Chip level Distributed System
- Compatible with
  - Low Latency Messaging
  - Slow processors with limited memory
- Real Time
  - Maintain thread priority, Bounded Processing Times
- Heterogeneous
  - Programs, Processors (32/64 bit, mixed endian), OSes, message transports

[Multicore Expo06]

# What is a *dref*?

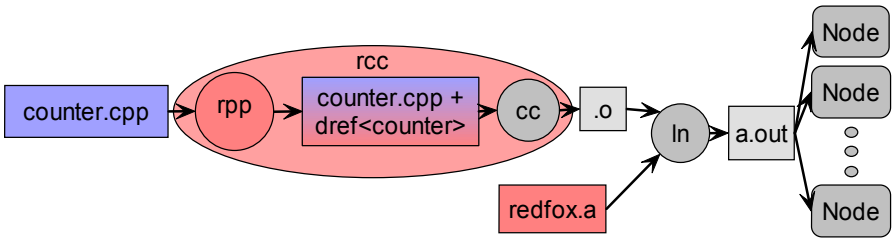
- **D**istributed **Ref**erence
- Conceptually like C++ reference
  *dref<T> ~ T&*
  *dref<const T> ~ const T&*
  *dptr<T> ~ T**
  *dptr<const T> ~ const T**
- Knows object location and methods
  - direct call if local
  - message passing if needed
- Fixed size node number and object address
  - Small, fixed size, fast to use, copy and transmit, valid on any node
  - Control application by passing around drefs



# Using *drefs*

- Automatically generated template class specialization
  *class A { ... };*
  *#pragma redplain dref<A> interface implementation*
- Supports method calls locally and remotely
  *dref<A> myA(…);*
  *myA.f();*
- Constructors support remote object creation
  - Control nodes by creating, using and destroying remote objects on them.
  - No name service needed
- Can copy and pass as method parameters

# Tool Chain

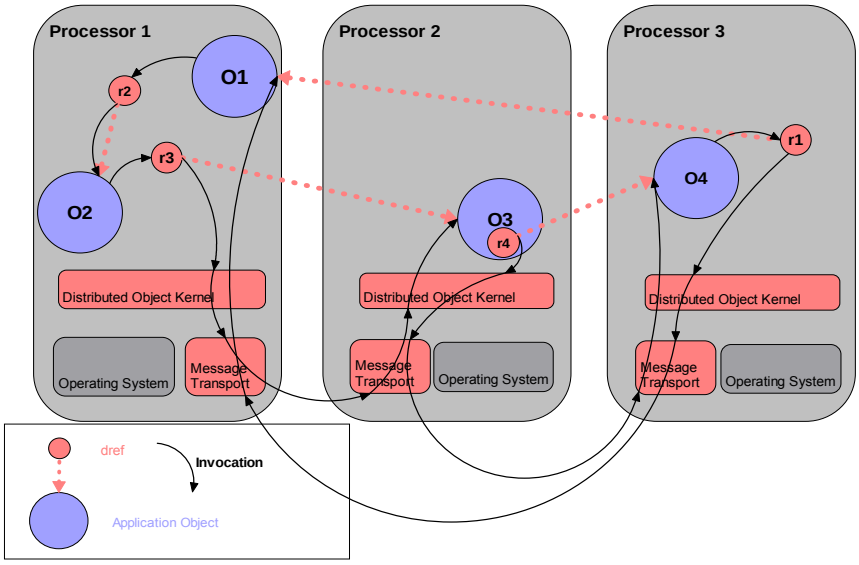



redFOX injects middleware into C++ source

- No intermediate files to manage



# Distributed Method Calls

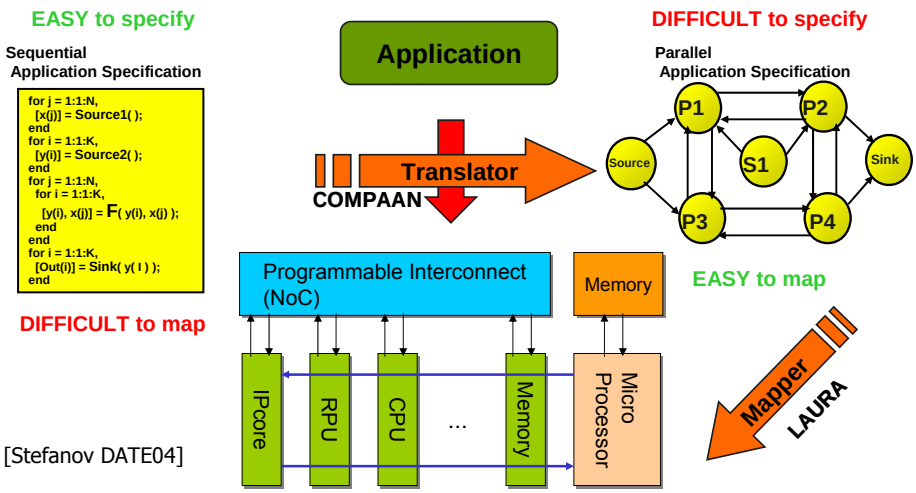


# MoCs and Productivity

- Distributed computing MoCs match NoC architecture very well
- Difficult to convince programmers to use them
  - Learning curve
  - What's the productivity at steady state?
- Productivity enhancement approaches
  - Automatic transformations (academia)
  - New languages (academia)
  - Standardized Component based design (industry)



# Automatic Transformations




[Stefanov DATE04]

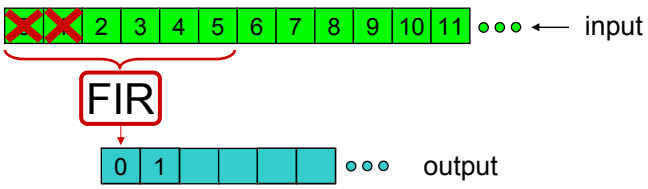
# New Languages

- StreamIt: a language for streaming applications
  - Provides high-level stream abstraction
    - Exposes Pipeline Parallelism
  - Improves programmer productivity
- Breaks the von Neumann language barrier
  - Each filter has its own control-flow
  - Each filter has its own address space
  - No global time
  - Explicit data movement between filters
  - Compiler is free to reorganize the computation

[Amarasinghe MIT]



# Example StreamIt Filter




*Stateless*

```
float→float filter FIR  (int N, float[N] weights) {

    work push 1 pop 1 peek N {
        float result = 0;

        for (int i = 0; i < N; i++) {
            result += weights[i] * peek(i);
        }
        pop();
        push(result);
    }
}
```

*Stateful*

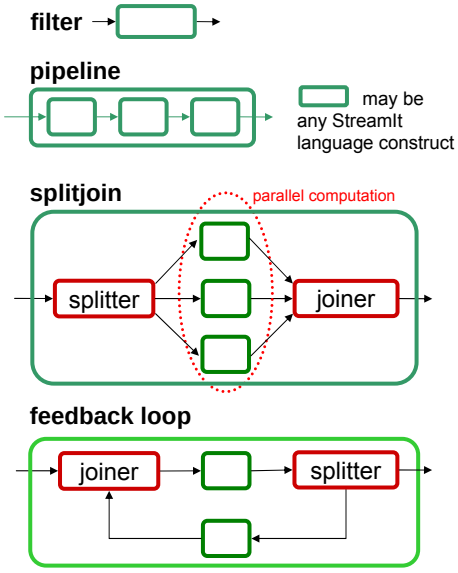
```
float→float filter FIR  (int N) {

    work push 1 pop 1 peek N {
        float result = 0;

        for (int i = 0; i < N; i++) {
            result += weights[i] * peek(i);
        }
        pop();
        push(result);
    }
}
```

# StreamIt Language Overview

- Modular and composable
  - Simple structures composed to creates complex graphs
- Malleable
  - Change program behavior with small modifications
- Compiler support






# Industrial Approaches

- TIPC, Chronos OpenMax, …
- Strongly emphasize standardization
  - Platform independent
  - Architecture agnostic
  - Very generic
- Domain specific
  - Large market segments (e.g. multimedia, automotive, communication)
- A lot of market hype: adoption is the ultimate measure of success

# TIPC: Transparent Interprocess Communication

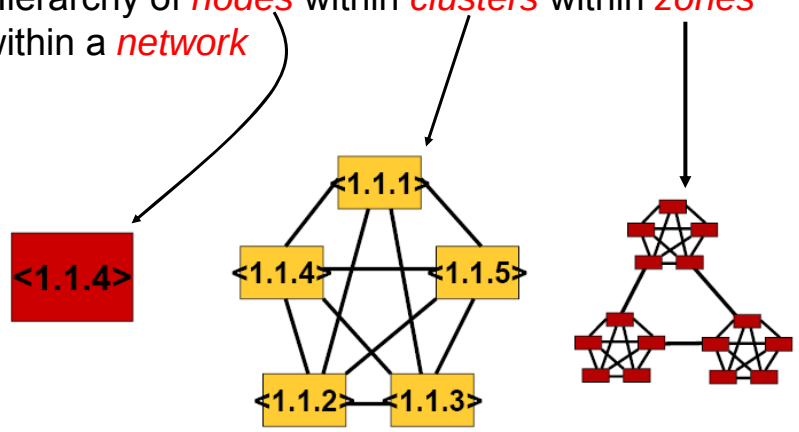
- Socket-based messaging for cluster environments
  - Footprint: smaller than IP
  - Performance: greater than IP
  - Scalability: From a single-node to a large cluster
  - Portability: OS and hardware-independent, standard socket API
  - Reliability: Link failover, reliable delivery, system monitoring
  - Flexibility: Dynamic discovery, location-transparent addressing
- Adapted to NoC characteristics:
  - Most messages are delivered in one hop (intracluster)
  - Transfer time is short for most messages
  - Packet loss is low; retransmission is infrequent
  - Available bandwidth and memory are high
  - Packet integrity is checked by underlying hardware
  - Security is not a major issue for intracluster messages
  - Network topology is relatively stable

[Windriver 06]



# TIPC Networking

Hierarchy of *nodes* within *clusters* within *zones* within a *network*



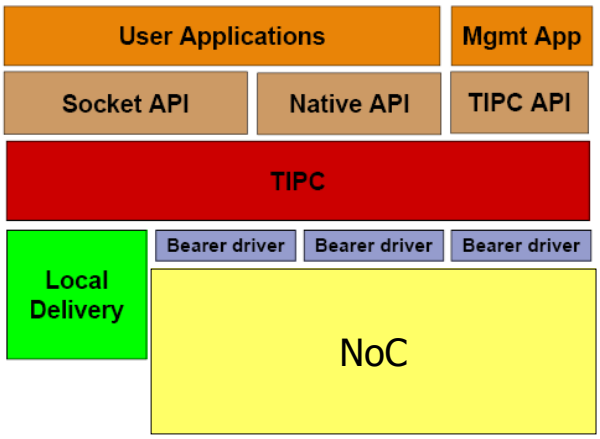
# TIPC Messaging

- All communication involves an exchange of data between *ports* using *sockets* (AF_TIPC)
- A *message* is a byte array from 1 to 66KB long
- Connectionless and connection-oriented messaging
- *Services* are identified by port name


**Provides a consistent message passing method that is OS-agnostic**



# TIPC architecture

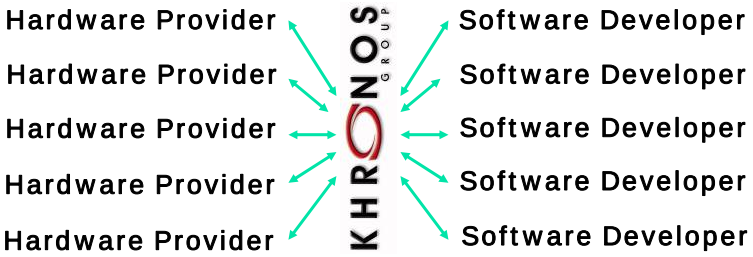



- Location transparency for services
  - Allows redundant services to gracefully (even seamlessly) handle failures
- Easy migration from a multiprocessor to a multicore system
- TIPC is open-source

# Kronos Media APIs

- An API is a contract between hardware and software worlds
- SW developers see reduced variability across multiple platforms
  - More software can reach market faster at a better level of functionality and quality
- Hardware vendors can accelerate many applications
  - Adding value to their platform

[Freescale 05]




**Khronos develops "Foundation-Level" APIs**
As close-to-the-HW as possible while providing portable access to hardware acceleration. High performance.
Good foundation for higher-level engines and middleware



# Khronos' Business Model

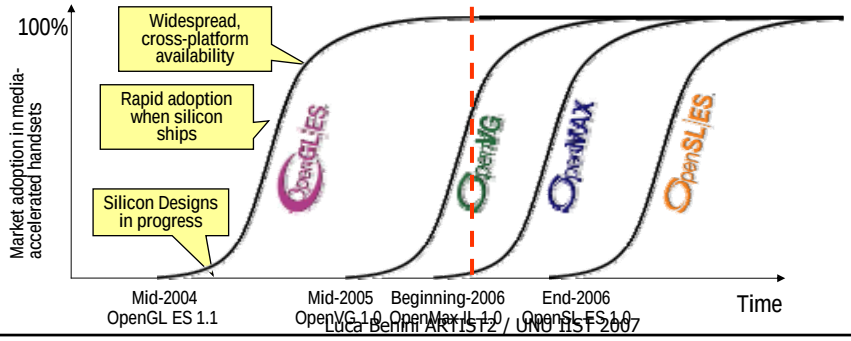
**Open Membership**
Any company can become member
Funded by membership dues - $5K / year

**Open Standards**
Publicly available on web-site
Royalty-free

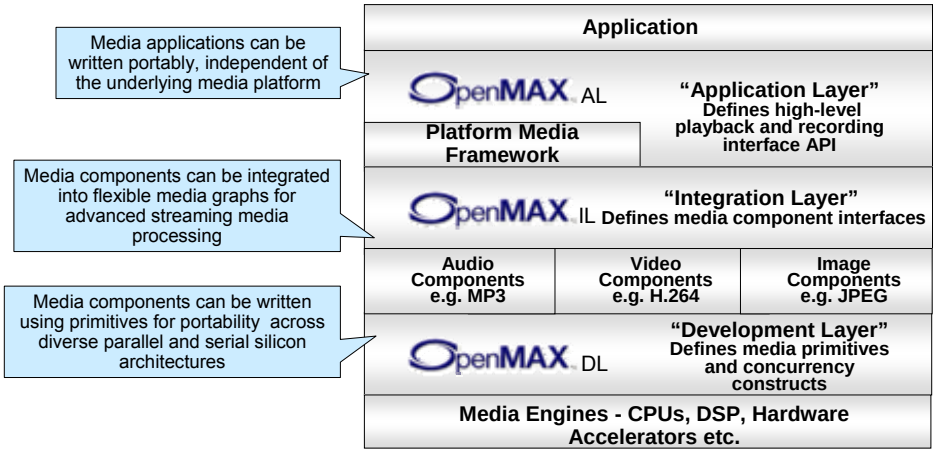
KHRONOS GROUP
Open Standard Platform for Embedded Rich Media Acceleration

**Cross Platform**
Enabling diverse handheld and embedded markets

**Promoting Ecosystem**
Conformance tests, tools, developer materials and outreach



# OpenMAX-based Media Stack

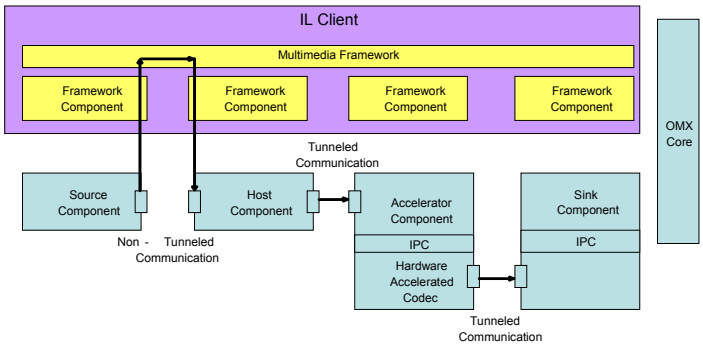



OpenMAX layers can be implemented together or independently from the other layers



# System Architecture

- OMX components behave deterministically
  - Whether they are loosely or tightly accelerated
- Interop profile components are guaranteed to interoperate
  - Via a specified interoperability protocol



# OpenMAX AL - Application Level

- Enabling application developers to easily leverage OpenMax acceleration
  - A simple high-level interface for common multimedia playback and capture use cases
- Typical applications are found in:
  - Mobile Phones
  - Mobile Music/Video Players
  - PDAs
  - Digital still cameras
  - Digital Media Adapters
  - STBs, PCs, etc…

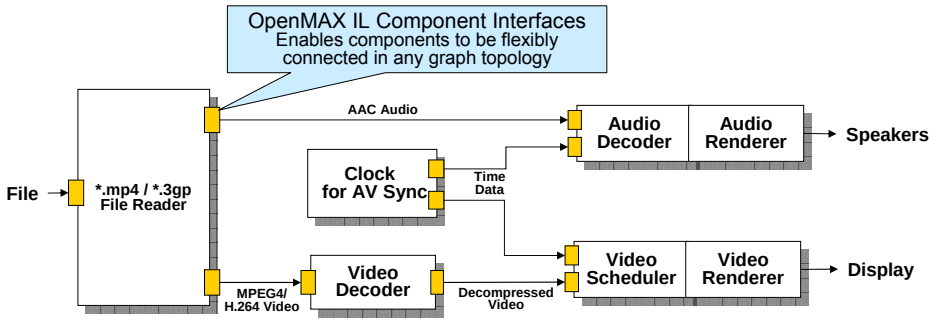


# OpenMAX IL - Integration Level

- Defines component interfaces to construct portable media graphs
  - OpenMAX IL graphs are consistent across systems
- Abstracts hardware architecture
  - Processor specific code is encapsulated within components
  - Intelligently built components maximize system utilization
- Reusable integration with major media frameworks
  - Provides a uniform interface for integration across many architectures
  - Designed to sit below major frameworks - e.g. Symbian MDF, GStreamer,
  - Defines a low level initialization and communication protocol
- Extensible
  - Extensions to expose non standard features with only minor tweaks
- Enables Performance Comparisons and Optimization
  - Common API allows benchmarking of different architectures, implementations and frameworks

# OpenMAX IL Example Graph

- Standardized component interfaces enable flexible media graphs
- Includes multi-stream synchronization

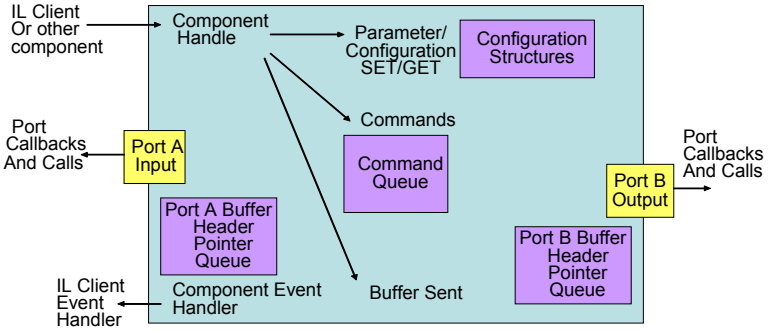



**Example: MPEG-4 video synchronized with AAC audio decode**



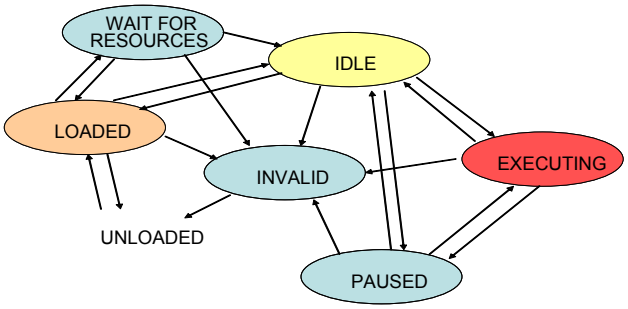
# Component Architecture

- Component Registration is specified for low level operating systems
  - May not have a defined mechanism for discovery
  - Statically linked components use OpenMAX mechanisms
  - Dynamically linked components may be registered and loaded with system mechanisms
- Standard configuration structures are assigned to individual ports
  - Ensuring component interoperability and may be expanded with proprietary configurations



# Component States

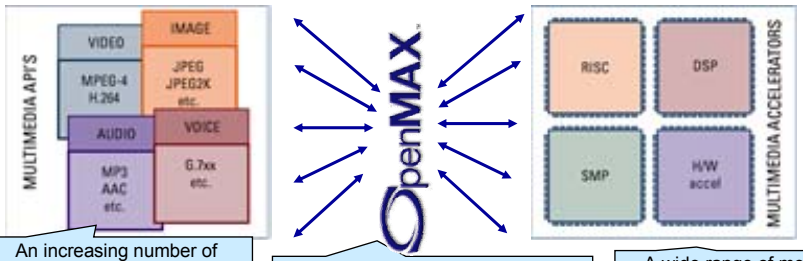
- Resource allocation takes place between Loaded and Idle
  - With possible resource management pending in Wait For Resources
- The invalid state is entered only upon internal detection of a problem






# OpenMAX DL – Low-Level Media API

- OpenMAX DL is a library of key static primitive functions
  - Designed to cover 80% of the processing required in a multimedia codec
- Abstracts the ISA from the multimedia codec
  - Enables faster codec development time and faster porting of existing codecs
- Enables third party codec vendors to sell processor-agnostic codecs
  - Multi-cores architectures gain greater code reuse between cores




An increasing number of multimedia API codecs for video, audio, graphics and images

Silicon vendors supply optimized OpenMAX DL library for rapid porting of codecs across multiple accelerators

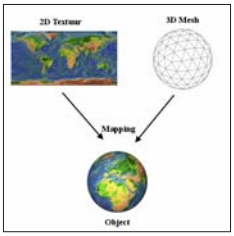
A wide range of media acceleration silicon using many diverse architectures

# Example: OpenMAX DL Video Domain

- Computationally intensive “hotspots” for video applications
  - Basic video processing building blocks
- Typical devices
  - Digital still cameras, PDAs, Mobile Phones, Portable Media Players, Set-top-boxes, PCs, etc.
- Example video primitive functions in OpenMAX DL 1.0
  - 8x8 Add, Sub and 16X16 Add, Sub
  - 8x8 DCT+Q+Scan and 8x8 IDCT+Q+InvScan
  - MPEG-4 Variable Length Decode
- Merged functions, improved performance on some architectures
  - Motion Estimation, Motion Compensation, Deblocking
- Video codecs covered by OpenMAX DL 1.0
  - MPEG-4 SP/H.263 BL (encode & decode)
  - H.264 (encode and decode)



# Advanced SoC Applications Scenario




Scalable video rendering

3D Virtual reality games

Wireless protocols

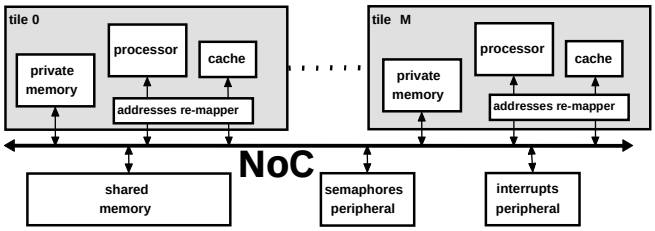
- Complex object-oriented high-level design (C++, Java)
- Very dynamic (variable use of resources for each input)
- High demand for processing (energy cost?)

# Dynamic Resource Management: Why?

- The need for dynamic resource management in MPSoCs
  - Strongly impacts performance, energy consumption and reliability
  - Required by flexible (programmable/configurable) platforms
  - Strongly impacted by algorithm/application architecture
- DRM objectives
  - Performance
  - **Energy**
  - Reliability
  - Variability
- DRM should accomplish these task without impacting programmability
  - However, cooperative design-time/offline run-time/online approaches are needed

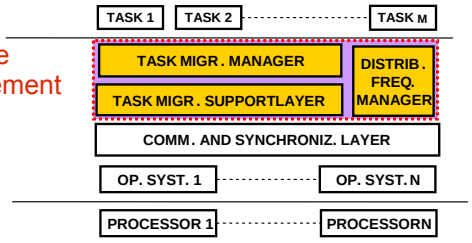


# System Model




Distributed MPSoC architecture

resource management layer




**Software architecture**

# 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

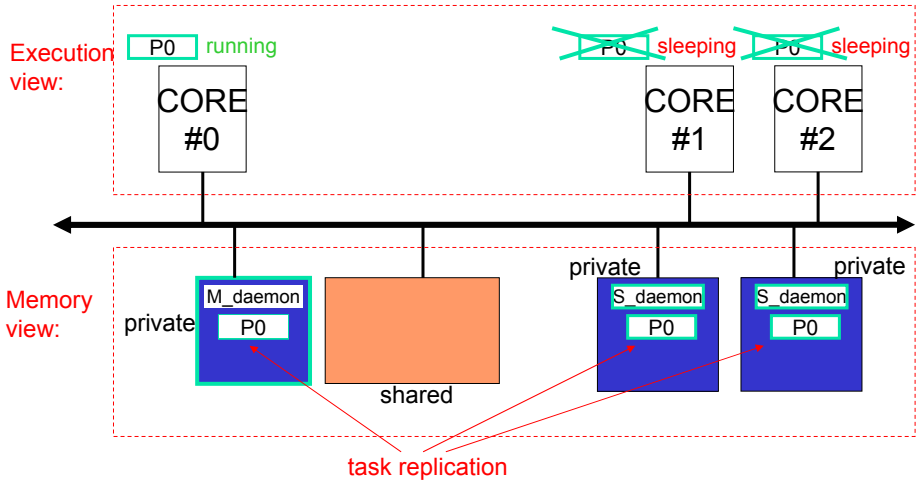
[Acquaviva DATE06]



# Kernel Daemons Infrastructure

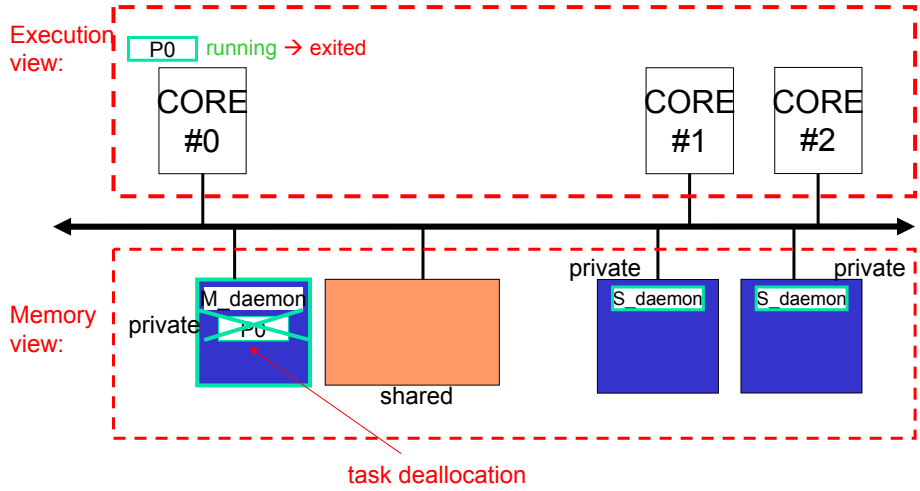
- The migration process is managed using two kinds of kernel daemons:
  - a master daemon on a single (master) processor,
  - slave daemons on each (worker) processor.
- The master daemon is directly interfaced to the decision engine providing the selected policy for run time task allocation.
- Master 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

# Task replication

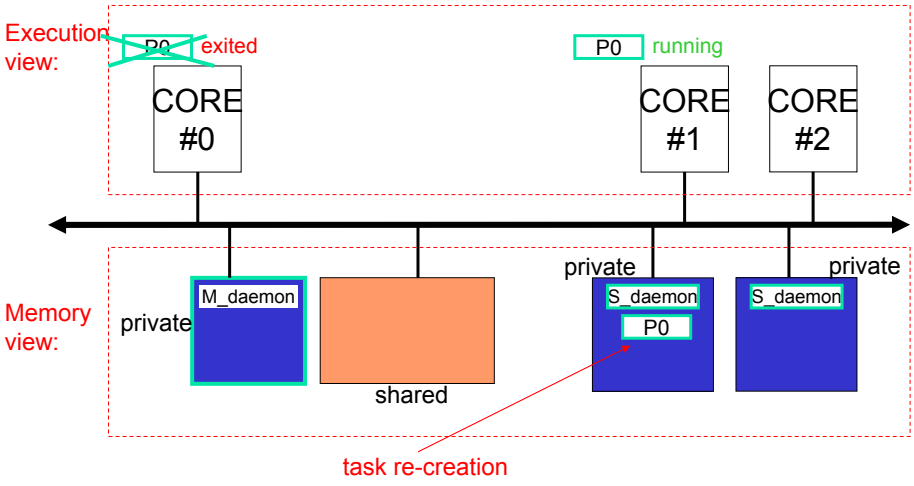





# Task re-creation

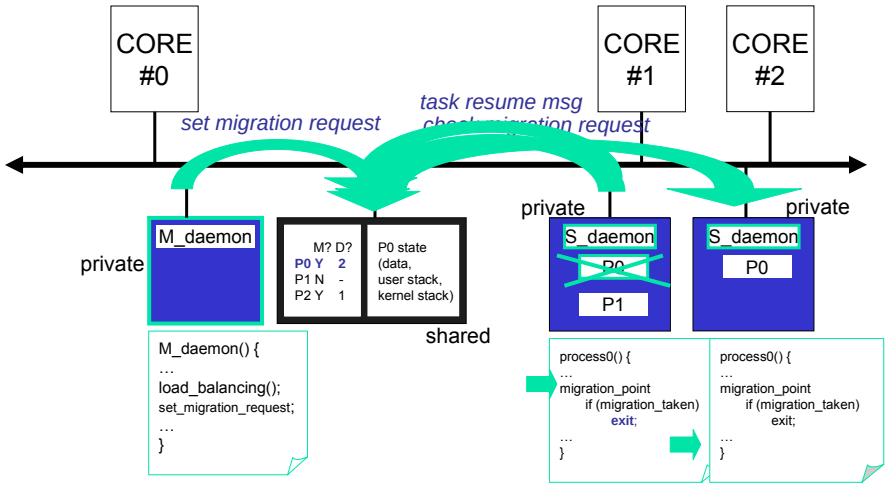


# Task re-creation

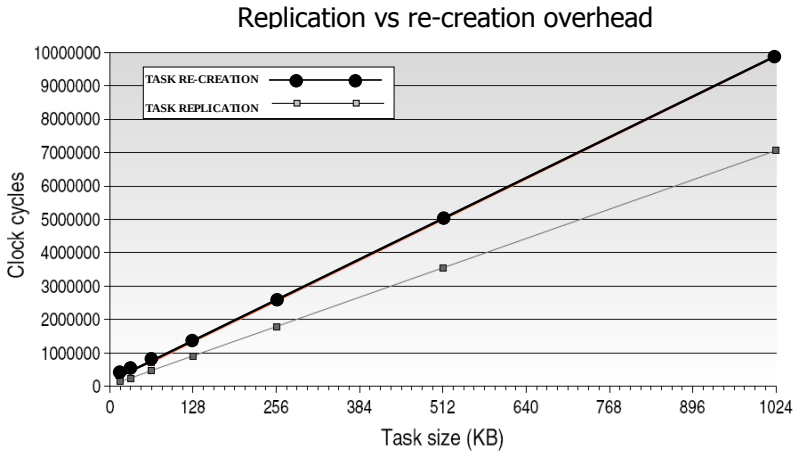





# Migration mechanism



# Replication vs. Re-creation

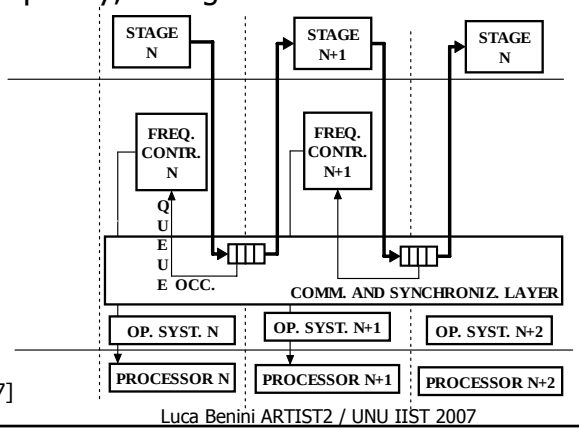



**Cost of replication: increased memory usage**



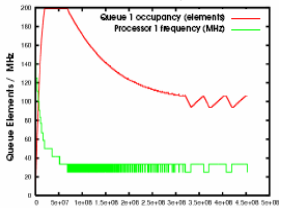
# Frequency, Voltage Setting Support

- Migration+dynamic f,Vdd setting critical for energy management
  - DVFS for multi-stage producer-consumer streaming exploits info on occupancy of the synchronization queues
  - @equilibrium, average output rate should match input rate in each queue → occupancy level monitored to adjust PE speed and Vdd
- Local PE frequency, voltage controller

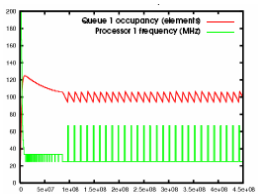



[Carta TECS07]

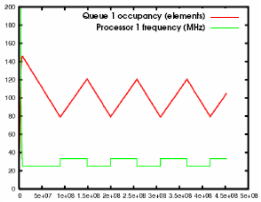
# Controller Comparison



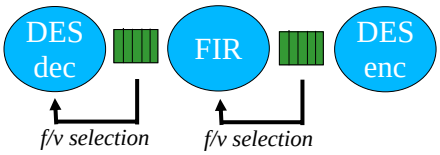

*PI controller setup 1:*
Low reactivity
Small steady-state oscillations




*PI controller setup 2:*
High reactivity
High steady-state oscillations




*Non Linear controller:*
High reactivity
Low steady-state oscillations

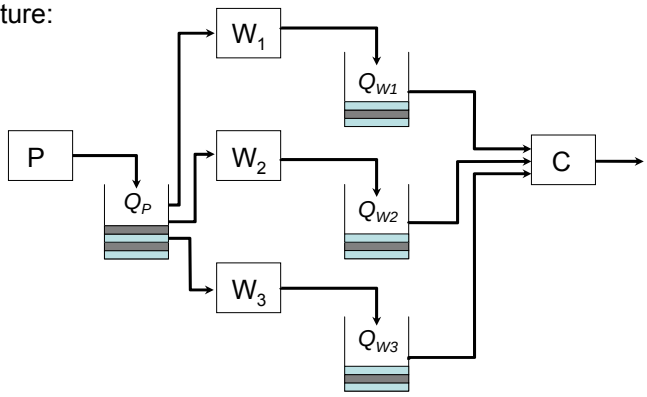



**3 stage pipelined streaming application**



# Pipeline/Parallel Streaming

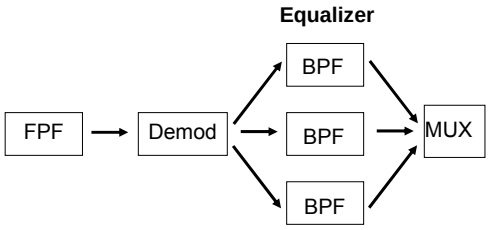
- Generalization by considering stages composed by multiple independent processing elements *working in parallel*.
- 3-stage architecture including a producer processor *P*, three workers $W_1$, $W_2$, $W_3$ working in parallel, and a consumer processor *C*
- *Split-join* structure:




[Alimonda ISIES06]

# MISO Controller

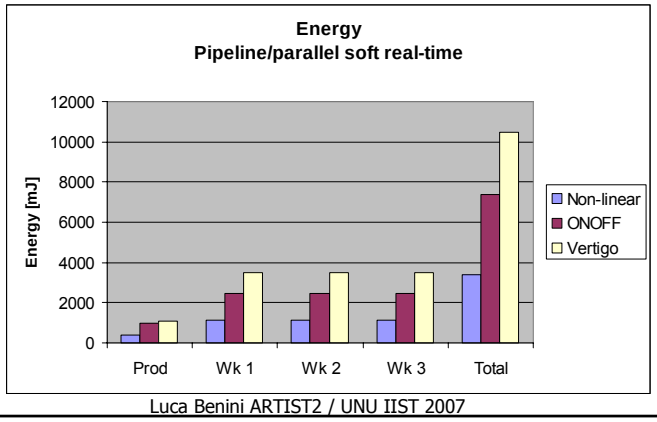
- The producer must look at three output queues:
  - MISO (Multiple Input-Single Output) controller
  - The control rule is the same as for simple pipeline
- BUT: what queue we have to look now?
  - Average (AVG)
  - Minimum (MIN)
  - Maximum difference (MAX-DIFF)
- They are currently under test using a software FMRadio benchmark





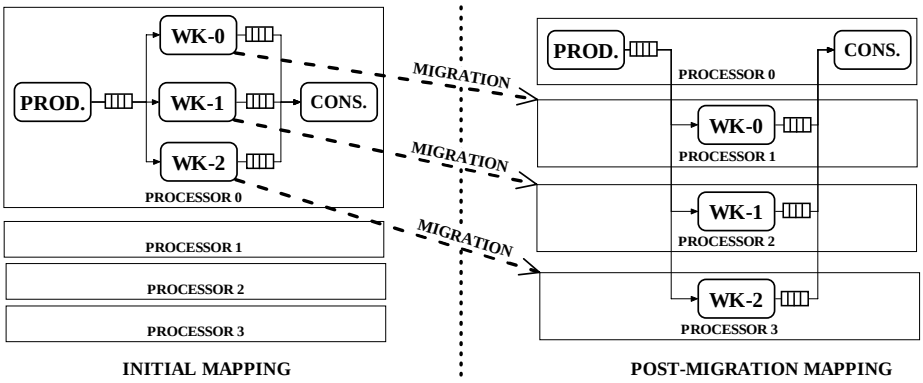

# Energy Efficiency Comparison

- Non-linear control strategy requires 50% of energy consumption wrt the ON-OFF controller
- 30% of energy consumption wrt open-loop (Vertigo)



# DVFS + Task migration

**Parallel DES load balancing on 4 processors configuration**

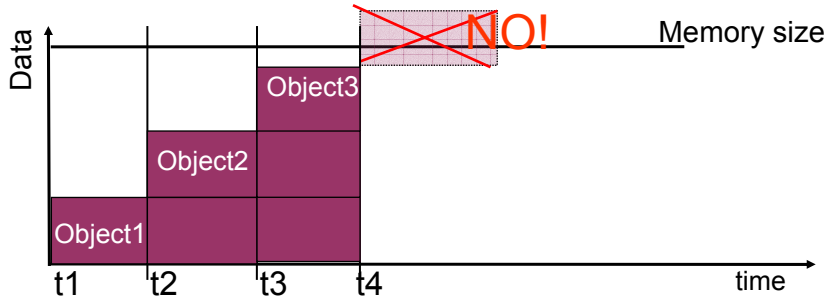




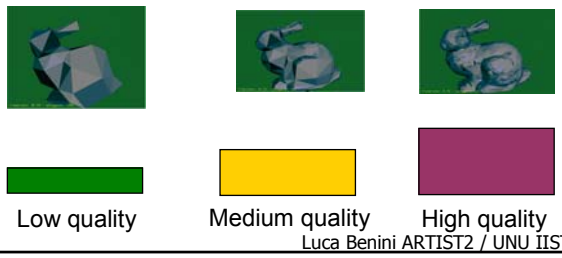

# Static memory vs Dynamic Memory (DM):

## Compile-time (worst case)

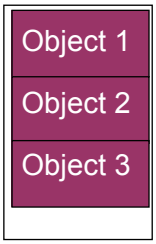
[Atienza DATE05]



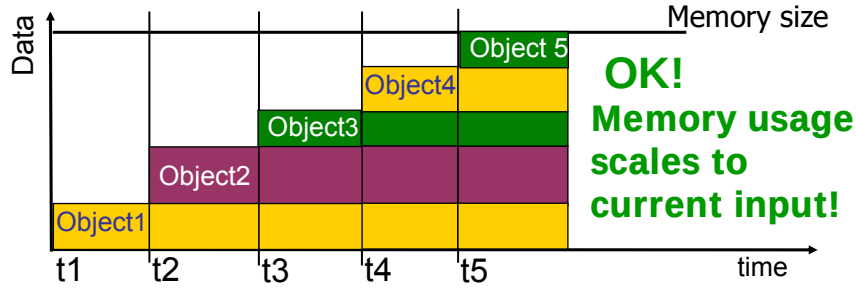

Scalable 3D decoding (per object):




Memory:

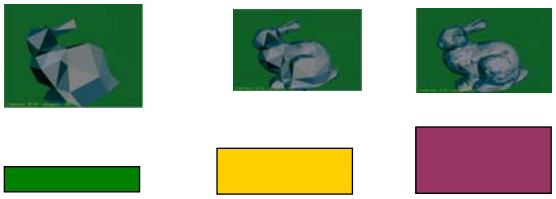


# Static memory vs Dynamic Memory:
## Run-time (or DM Management)



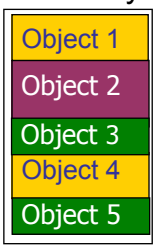

**OK!**
**Memory usage scales to current input!**

Scalable 3D decoding (per object):



Low quality | Medium quality | High quality

Memory:



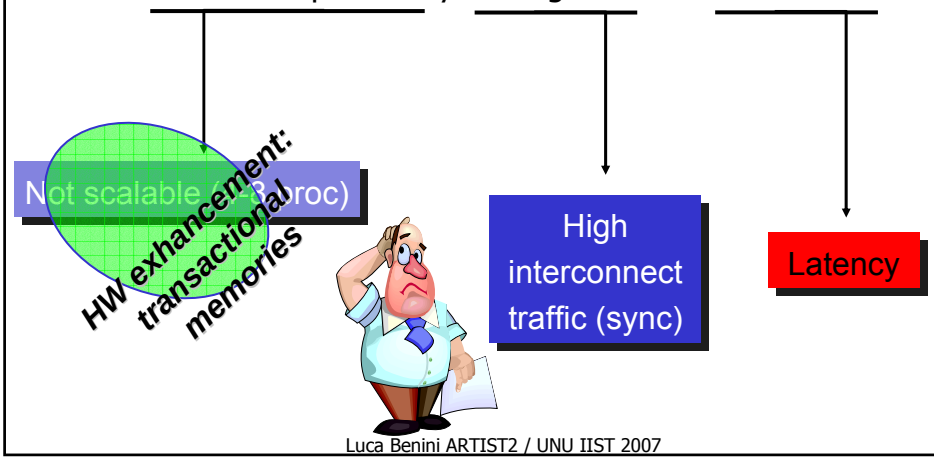



# New issues in DMMs for MPSoCs

- Low Latency
  - Malloc/new() & Free()/delete() must perform like in monoprocessor systems
- Scalability
  - Performance should scale linearly with number of processors (independent of number of memories)
- Memory efficiency (*blowup* problem)
  - Fragmentation competitive with monoprocessors
- Orthogonality
  - Independent of threading models (supporting large number of threads in preemptive multitasking environment).
- Minimal interconnection traffic
  - No cache coherency protocol synchronization
  - Software-based local & main heaps synchronization

# State-of-the-art DMMs MPSoCs (1)

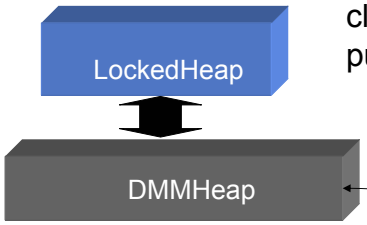
- Single shared heap (e.g., Solaris libc, McRT Malloc)
  - 1 global mutex
  - Simple design (1 extra layer to monoproc DMM)
  - No heap memory wastage






# State-of-the-art DMMs MPSoCs (1)

- Protecting the upper global heap:




```
class LockedMallocHeap:
public LockedHeap<DMMHeap> {};
```

```
void * malloc (sz) {
  acquire lock;
  void * p =   SuperHeap::malloc (sz);
  release lock;
  return p;
}
```

# State-of-the-art DMMs MPSoCs (2)

- Private heap per thread (e.g., CilK, STL)
  - Memory footprint wastage
  - Almost no synchronization per thread (local heaps)

Heap memory size

False sharing (blowup)

Special HW support
Locked/store conditional

Shared synchro



# State-of-the-art DMMs MPSoCs (3)

- Multiple threads per local heap (e.g., Hoard, LFMalloc, Vee-Hsu)
  - Hierarchical structure of shared blocks
  - Limited false sharing (no blowup)

Extra heaps to avoid contention

Emptiness threshold

Threads have to become 'mature'

Tuning per app.

## Still to come in DMMs for MPSoCs

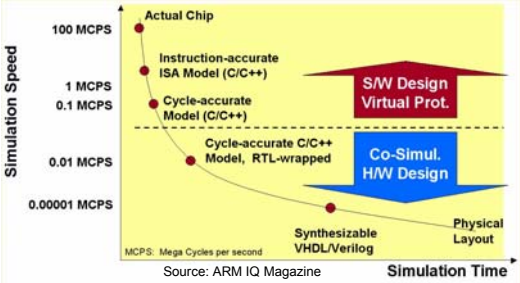
- Tuning of DMMs for MPSoCs
  - Profiling mechanisms not available, manual work
- Exploitation of hardware
  - Scratchpad memories not supported
  - Management of heaps for HW-controlled caches
- Complex design
  - Custom synchronization methods
  - No standard & flexible implementation methods
  - Support for hardware synchronization in final architecture
- Non-orthogonal
  - Dependencies on number of threads
- Influence of interconnect
  - Monitoring interconnect status (e.g, congestion?)



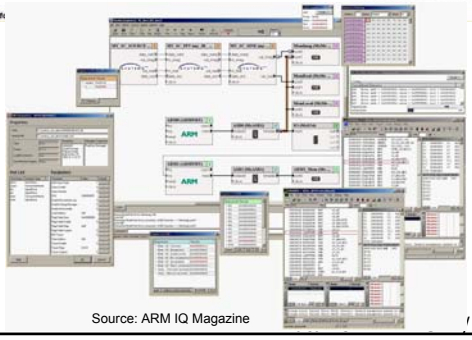
## Debugging

- When real code is running on a real MP system/models, “bad” things happen
- Debugging challenges
  - Tracing bugs across multiple models of computation
    - Loosely coupled composition of subsystems
    - Several different ISA processors
    - Multiple languages and communications models
  - How to sanely move from paradigm to paradigm?
- A single debugging “driver seat”
  - Preferably not with a different debugger per processor
  - Provide true MP Debug with a unified system view

# Debug & Simulation




Source: ARM IQ Magazine



Source: ARM IQ Magazine

- Today, simulation speed is limiting
- Need faster simulation
  - Enabling trade offs
  - Flexibility: appropriate accuracy at appropriate speed
- Today, single core debugging approaches don't scale to MPSoC
- Need true multi processor debug
  - Focused on threads
  - Scaling to 10+ processors



# Performance tuning/ Benchmarking

- Real-life Application complexity is huge
- Current embedded benchmarks are focused on single processors
- Need to assess scalability (future proof)
- Real applications vs. microbenchmarks
  - Finding common denominator
  - Generalize to assess future scalability
  - Compose to assess mixed workload

# µbenchmarks – UCB's 14 dwarfs

| | | HPC | Embed | SPEC | ML | Games | DB |
|---|---|---|---|---|---|---|---|
| 1 | Dense Matrix | | | | | | |
| 2 | Sparse Matrix | | | | | | |
| 3 | Spectral (FFT) | | | | | | |
| 4 | N-Body | | | | | | |
| 5 | Structured Grid | | | | | | |
| 6 | Unstructured | | | | | | |
| 7 | MapReduce | | | | | | |
| 8 | Combinational | | | | | | |
| 9 | Nearest Neighbor | | | | | | |
| 10 | Graph Traversal | | | | | | |
| 11 | Dynamic Prog | | | | | | |
| 12 | Backtrack/ B&B | | | | | | |
| 13 | Graphical Models | | | | | | |
| 14 | FSM | | | | | | |

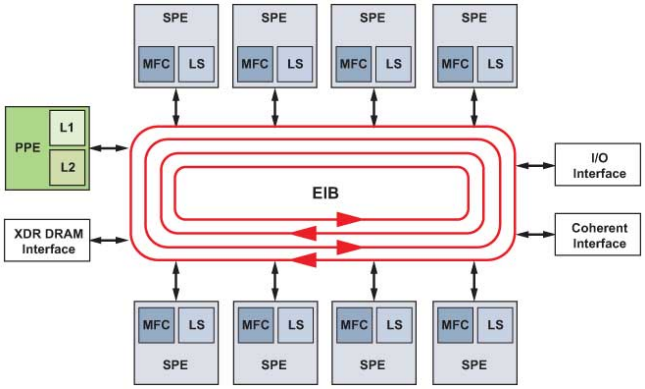


# Case study:
# Mercury's Multicore Plus SDK for CELL

A mature software environment
for a high-performance MPSoC

[Mercury IEEE AESS06]

# Cell BE Processor Architecture




Heterogeneous, distributed memory multiprocessor with explicit DMA over a ring-NoC



# Mercury Approach to Cell SDK

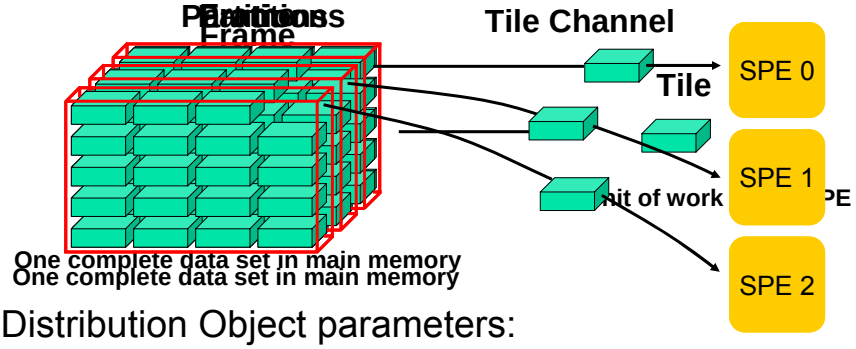
- Pragmatic
  - Can't wait for tools to mature
  - Develop in-house tools when needed
- Emphasis on explicitly programming the architecture rather than trying to hide it
  - When the tools are immature, this allows us to get maximum performance
- Achieve usability, portability through function offload model
  - Run legacy code on PPE
  - Offload compute intensive workload to SPEs
- An API for programming *heterogeneous multicores* with *explicit non-cached memory hierarchies*
- Provides an abstract view of the hardware oriented toward computation of *multidimensional data sets*

# MultiCore Framework Abstractions

- Function offload model
  - Worker Teams: Allocate tasks to SPEs
  - Plug-ins: Dynamically load and unload functions from within worker programs
- Data movement
  - Distribution Objects: Defining how n-dimensional data is organized in memory
  - Tile Channels: Move data between SPEs and main memory
  - Re-org Channels: Move data among SPEs
  - Multibuffering: Overlap data movement and computation
- Miscellaneous
  - Barrier and semaphore synchronization
  - DMA-friendly memory allocator
  - DMA convenience functions
  - Performance profiling

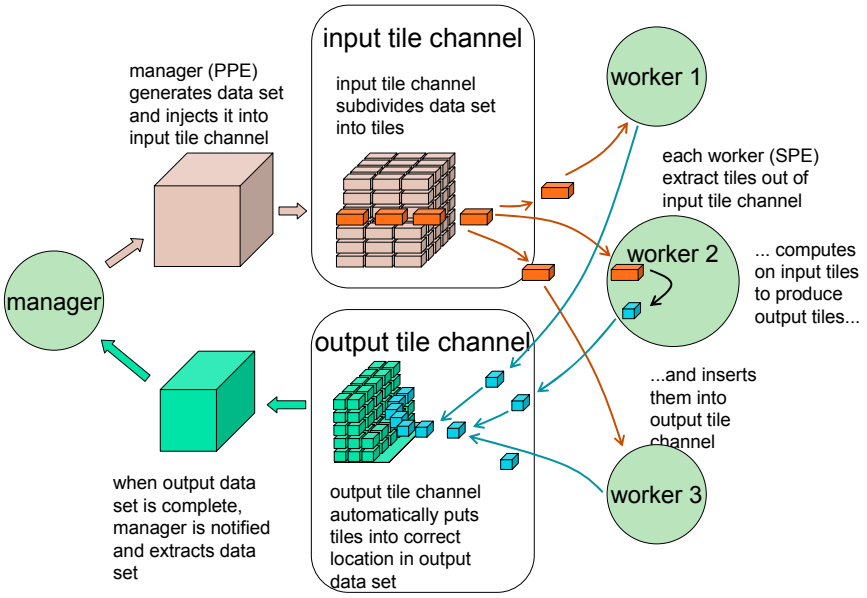


# MCF Distribution Objects




- Distribution Object parameters:
  - Number of dimensions
  - Frame size
  - Tile size and tile overlap
  - Array indexing order
  - Compound data type organization (e.g. split / interleaved)
  - Partitioning policy across workers, partition overlap

# MCF Tile Channels






# MCF Manager Program

```
main(int argc, char **argv) {
   mcf_m_net_create();
   mcf_m_net_initialize();

   mcf_m_net_add_task();
   mcf_m_team_run_task();

   mcf_m_tile_distribution_create_3d("in");
   mcf_m_tile_distribution_set_partition_overlap("in");
   mcf_m_tile_distribution_create_3d("out");

   mcf_m_tile_channel_create("in");
   mcf_m_tile_channel_create("out");
   mcf_m_tile_channel_connect("in");
   mcf_m_tile_channel_connect("out");

   mcf_m_tile_channel_get_buffer("in");

   // fill input data here

   mcf_m_tile_channel_put_buffer("in");
   mcf_m_tile_channel_get_buffer("out");

   // process output data here
}
```

- Add worker tasks
- Specify data organization
- Create and connect to tile channels
- Get empty source buffer
- Fill it with data
- Send it to workers
- Wait for results from workers

# MCF Worker Program

```
mcf_w_main (int n_bytes, void * p_arg_ls) {
   mcf_w_tile_channel_create("in");
   mcf_w_tile_channel_create("out");
   mcf_w_tile_channel_connect("in");
   mcf_w_tile_channel_connect("out");

   while (! mcf_w_tile_channel_is_end_of_channel("in")
   {
       mcf_w_tile_channel_get_buffer("in");

       mcf_w_tile_channel_get_buffer("out");

       // Do math here

       mcf_w_tile_channel_put_buffer("in");

       mcf_w_tile_channel_put_buffer("out");
   }
}
```

Create and connect to tile channels

Get full source buffer

Get empty destination buffer

Do math and fill destination buffer

Put back empty source buffer

Put back full destination buffer



# MCF Implementation

- Consists of
  - PPE library
  - SPE library and tiny executive (12 KB)
- Utilizes Cell Linux "libspe" support
  - But amortizes expensive system calls
  - Reduces overhead from milliseconds to microseconds
  - Provides faster and smaller footprint memory allocation library
- Based on Data Reorg standard
  - http://www.data-re.org
- Derived from existing Mercury technologies
  - Other Mercury RDMA-based middleware
  - DSP product experience with small footprint, non-cached architectures

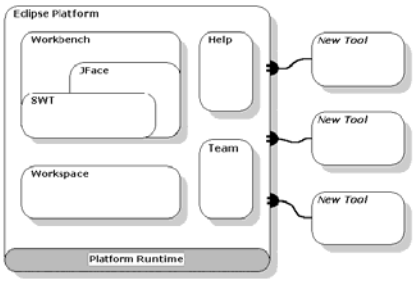
# Scientific Algorithm Library

- SAL is a collection of optimized functions
  - Baseline
    - Arithmetic, data type conversions, data moves
  - DSP
    - FFTs, convolutions, correlation, filters, etc.
  - Linear Algebra
    - Linear systems, matrix decomposition, etc.
  - Parallel Algorithms (future)
    - High level algorithms on multiple cores
    - Invoked from application running on PPE
    - Automatically use one or more SPEs
    - Initial work done for 1D and 2D FFTs and fast convolutions
- PIXL – Image Processing Library
  - Edge detection, fixed point operations and analysis, filtering, manipulation, erosion, dilation, histogram, lookup tables, etc.
  - Work in this area depend on customer demand.
- PPE SAL based on Altivec optimizations for G4 and G4A2
  - SAL C source code version also available
- SPE SAL is new implementation optimized for SPE architecture
  - Backwards compatibility with existing SAL API except in very rare cases
  - Some new APIs needed in order to extract best performance from SPE
  - Static and plug-in component versions for each function

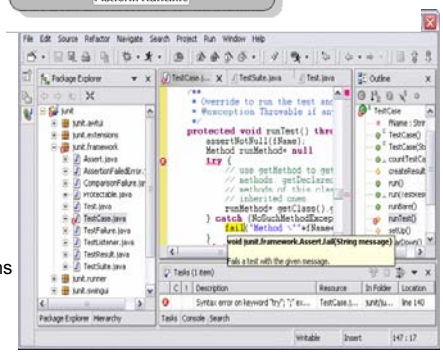


# Eclipse Framework

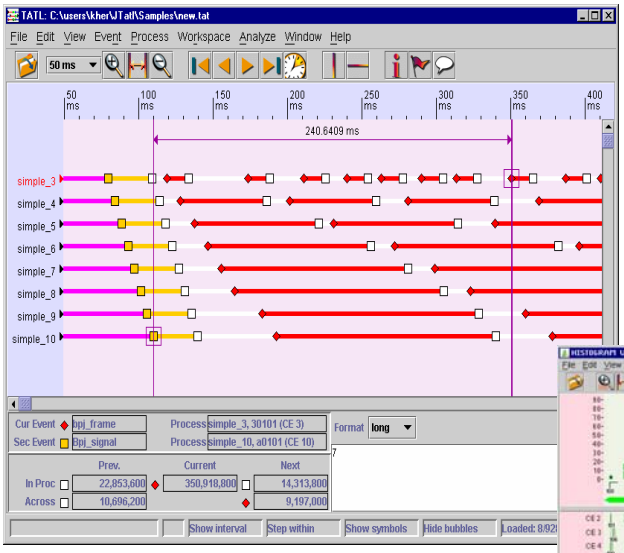
- Provides an open platform for creating an Integrated Development Environment (IDE)
- Eclipse Consortium manages continuous development of the tool
- Eclipse plug-ins extend the functionality of the framework
- Written in Java
- Compilers, debuggers, TATL, helpfiles, etc. are all be Eclipse plug-ins.




- PPE and SPE cross build support for
  - Gcc/gcc++
  - XLC/C++
- Eclipse CDT (C/C++ Development Toolkit)
  - Syntax highlighting
  - Code completion
  - Content assistance
  - Makefile generation
  - Remote debugging of PPE and SPE applications
  - TATL plug-in



# TATL™ Trace Analysis Tool




- Log events from PPE & SPE threads across multiple Cell chips
- Synchronized global timestamps
- Minimally intrusive in space and time
- Timeline trace and histogram viewers
- Structured log file for use in other tools



# SPE Assembly Development Kit

- The SPE architecture encourages “bare metal programmers”
  - Very deterministic architecture
  - Performance benefits from hand tuning the pipelines
- SPE-ADK dramatically improves bare metal productivity
- SPE-ADK consists of
  - Assembler preprocessor, optimizer and macro library
- Using SPE-ADK is similar to programming with SPE C extensions
  - But with more deterministic control of instruction scheduling and hardware resources
- SPE-ADK is a productized version of the internal development tool used by all Mercury SAL developers

# Summing up

- Harnessing the potential of NoC architectures from the SW "driver seat" is tough
- Programmers need all the help
- The landscape is shaping up rapidly
  - Support for MoCs with exposed parallelism
- Frontier of research
  - Efficient implementation
  - Parallelism extraction
  - Dynamic resource management