Systems and Technology Group

Cell Software Model

Course Code: L2T1H1-11
Cell Ecosystem Solutions Enablement

Class Objectives – Things you will learn

- **Cell software considerations including**
  - Two levels of parallelism: SIMD and parallel task execution
  - Computational
    - Multicore
    - Multithreading
    - Multiple local store accesses
  - Commutational
    - DMA and bus bandwidth
    - Traffic control
    - Shared memory/message passing
    - Synchronization
- **Programming models that exploit cell features**

Class Agenda

- **Cell Software Design Considerations**
- **Cell Programming Model Overview**
- **PPE Programming Models**
- **SPE Programming Models**
- **Parallel Programming Models**
- **Multi-tasking SPEs**
- **Cell Software Development Flow**

References

- Michael Day, Ted Maeurer, and Alex Chow, Cell Software Overview

Trademarks

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 (multi-threaded)
  - 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
  - Shared memory / Message passing abstraction overhead
  - Synchronization
  - DMA latency handling

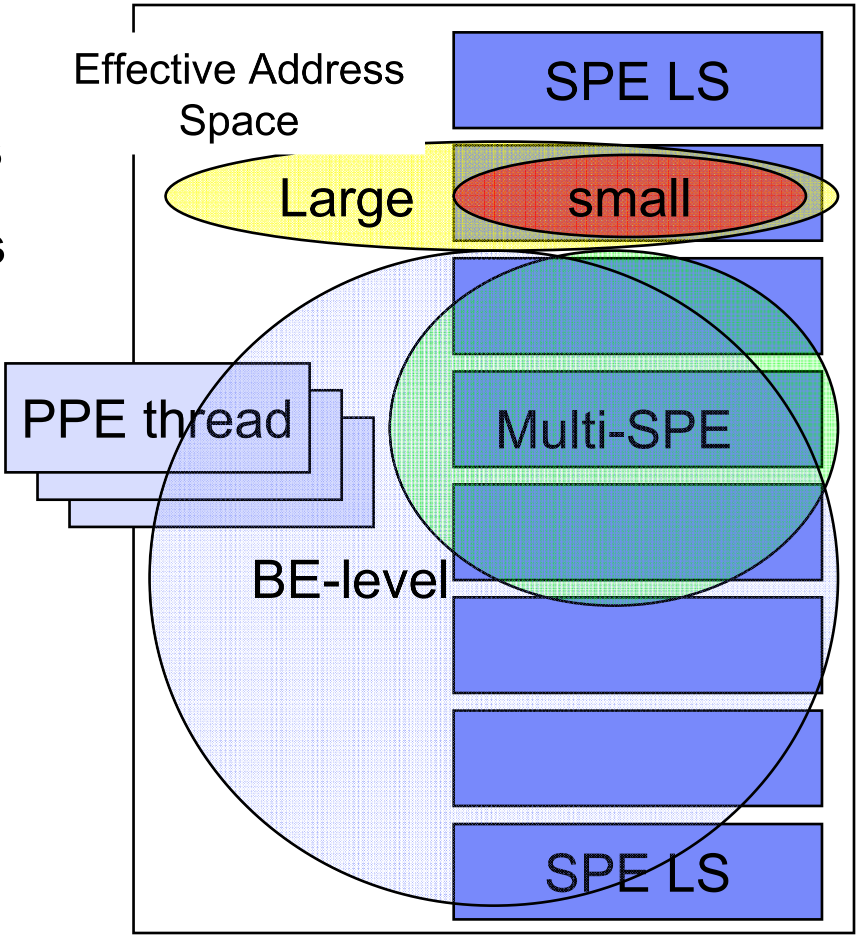
The role of Cell programming models

- **Cell provides a massive computational capacity.**
- **Cell provides a huge communicational bandwidth.**
- **The resources are distributed.**
- **A properly selected Cell programming model provides a programmer a systematic and cost-effective framework to apply Cell resources to a particular class of applications.**
- **A Cell programming model may be supported by language constructs, runtime, libraries, or object-oriented frameworks.**

Cell programming models

Single Cell environment:

- **PPE programming models**
- **SPE Programming models**
  - Small single-SPE models
  - Large single-SPE models
  - Multi-SPE parallel programming models
- **Cell Embedded SPE Object Format (CESOF)**



Cell programming models - continued

- **Multi-tasking SPEs**
  - Local Store resident multi-tasking
  - Self-managed multi-tasking
  - Kernel-managed SPE scheduling and virtualization
- **Application development flow**
- **Final programming model points**

PPE programming model (participation)

- **PPE is a 64-bit PowerPC core, hosting operating systems and hypervisor**
- **PPE program inherits traditional programming models**
- **Cell environment: a PPE program serves as a controller or facilitator**
  - CESOF support provides SPE image handles to the PPE runtime
  - PPE program establishes a runtime environment for SPE programs
    - e.g. memory mapping, exception handling, SPE run control
  - It allocates and manages Cell system resources
    - SPE scheduling, hypervisor CBEA resource management
  - It provides OS services to SPE programs and threads
    - e.g. printf, file I/O

Small single-SPE models

- **Single tasked environment**
- **Small enough to fit into a 256KB- local store**
- **Sufficient for many dedicated workloads**
- **Separated SPE and PPE address spaces – LS / EA**
- **Explicit input and output of the SPE program**
  - Program arguments and exit code per SPE ABI
  - DMA
  - Mailboxes
  - SPE side system calls
- **Foundation for a function offload model or a synchronous RPC model**
  - Facilitated by interface description language (IDL)

Small single-SPE models – tools and environment

- **SPE compiler/linker compiles and links an SPE executable**
- **The SPE executable image is embedded as reference-able RO data in the PPE executable (CESOF)**
- **A Cell programmer controls an SPE program via a PPE controlling process and its SPE management library**
  - i.e. loads, initializes, starts/stops an SPE program
- **The PPE controlling process, OS/PPE, and runtime/(PPE or SPE) together establish the SPE runtime environment, e.g. argument passing, memory mapping, system call service.**

Small single-SPE models – a sample

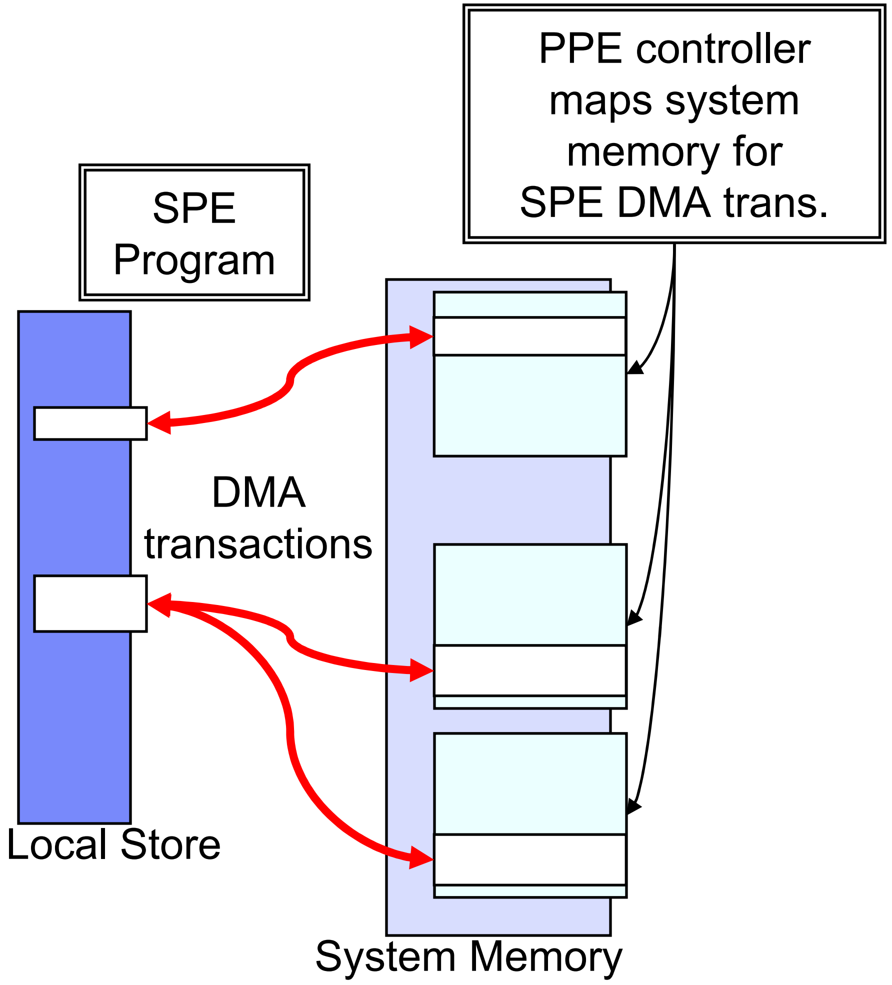
```
/* spe_foo.c:
 * A C program to be compiled into an executable called “spe_foo”
 */

int main( int speid, addr64 argp, addr64 envp)
{
char i;

            /* do something intelligent here */
            i = func_foo (argp);

            /* when the syscall is supported */
            printf( “Hello world! my result is %d \n”, i);

            return i;
}
```

Small single-SPE models – PPE controlling program

```
extern spe_program_handle spe_foo;  /* the spe image handle from
   CESOF */

   int main()
   {
   int rc, status;
   speid_t spe_id;

              /* load & start the spe_foo program on an allocated spe */
              spe_id = spe_create_thread (0, &spe_foo, 0, NULL, -1, 0);

              /* wait for spe prog. to complete and return final status */
              rc = spe_wait (spe_id, &status, 0);

              return status;
   }
```

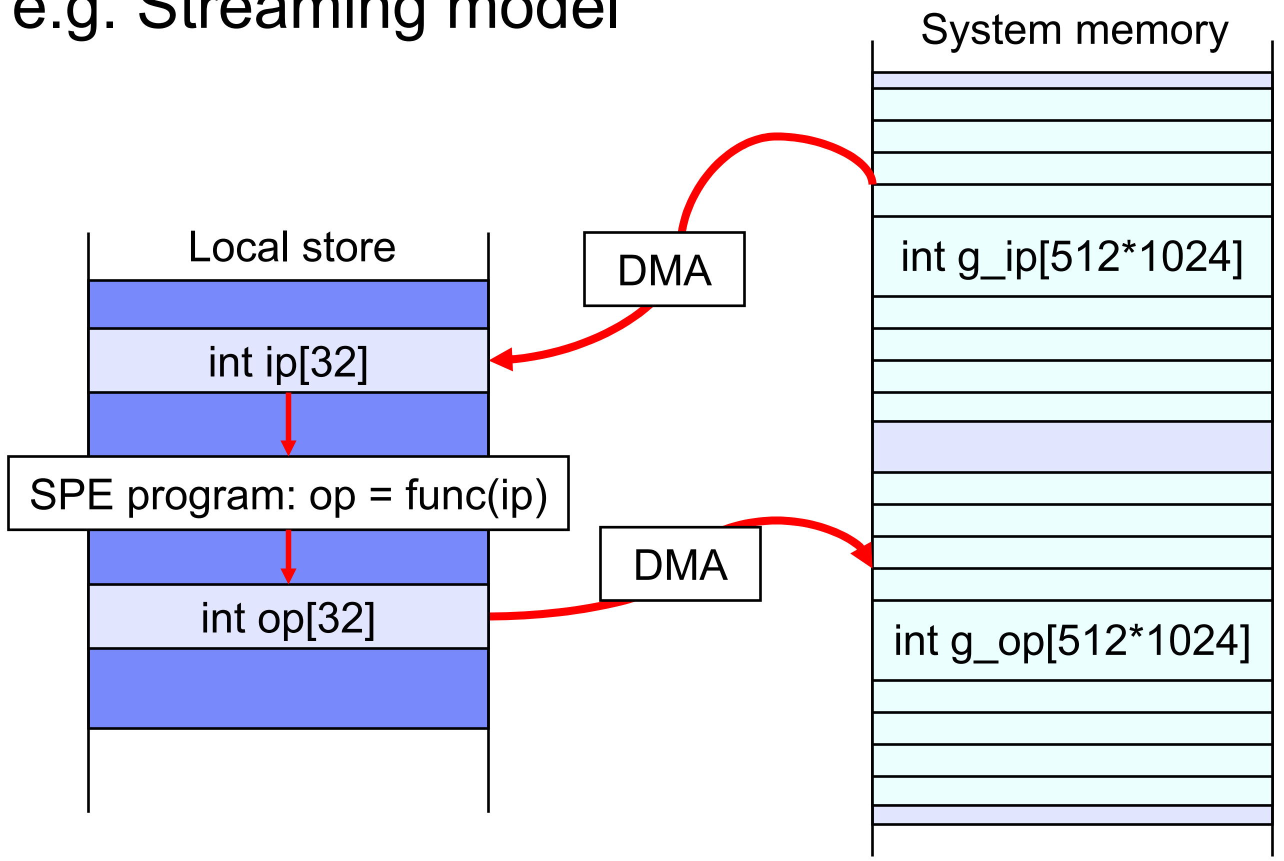
Large single-SPE programming models

- **Data or code working set cannot fit completely into a local store**
- **The PPE controlling process, kernel, and libspe runtime set up the system memory mapping as SPE's secondary memory store**
- **The SPE program accesses the secondary memory store via its software-controlled SPE DMA engine - Memory Flow Controller (MFC)**



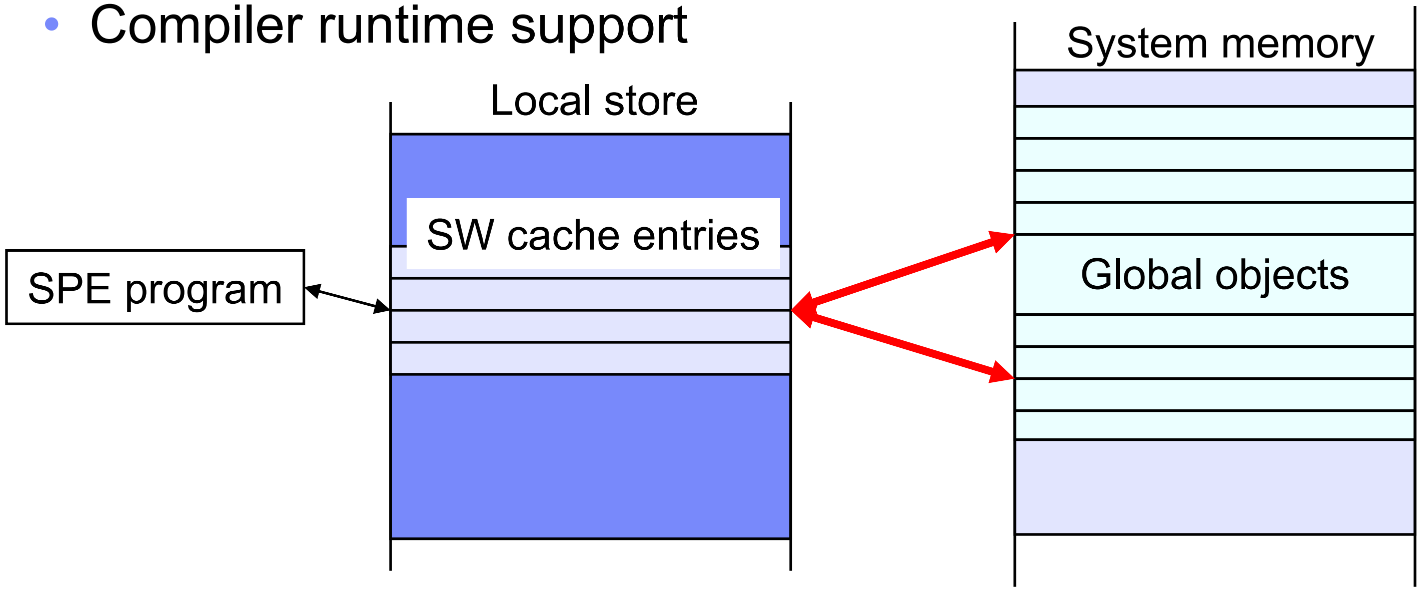
Large single-SPE programming models – I/O data

- **System memory for large size input / output data**
  - e.g. Streaming model



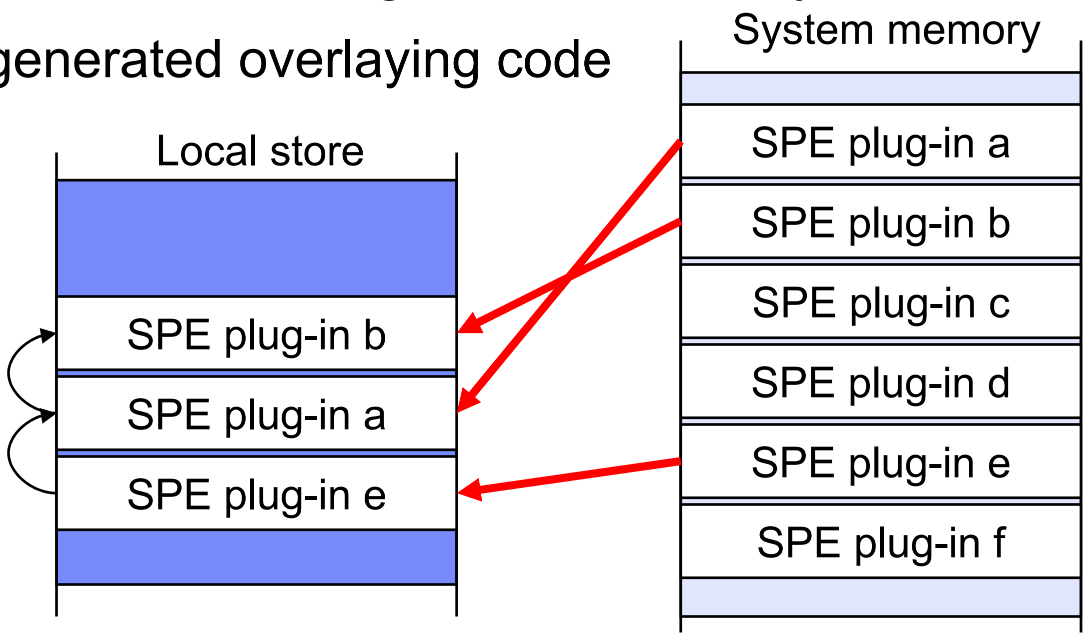
Large single-SPE programming models

- **System memory as secondary memory store**
  - Manual management of data buffers
  - Automatic software-managed data cache
    - Software cache framework libraries
    - Compiler runtime support



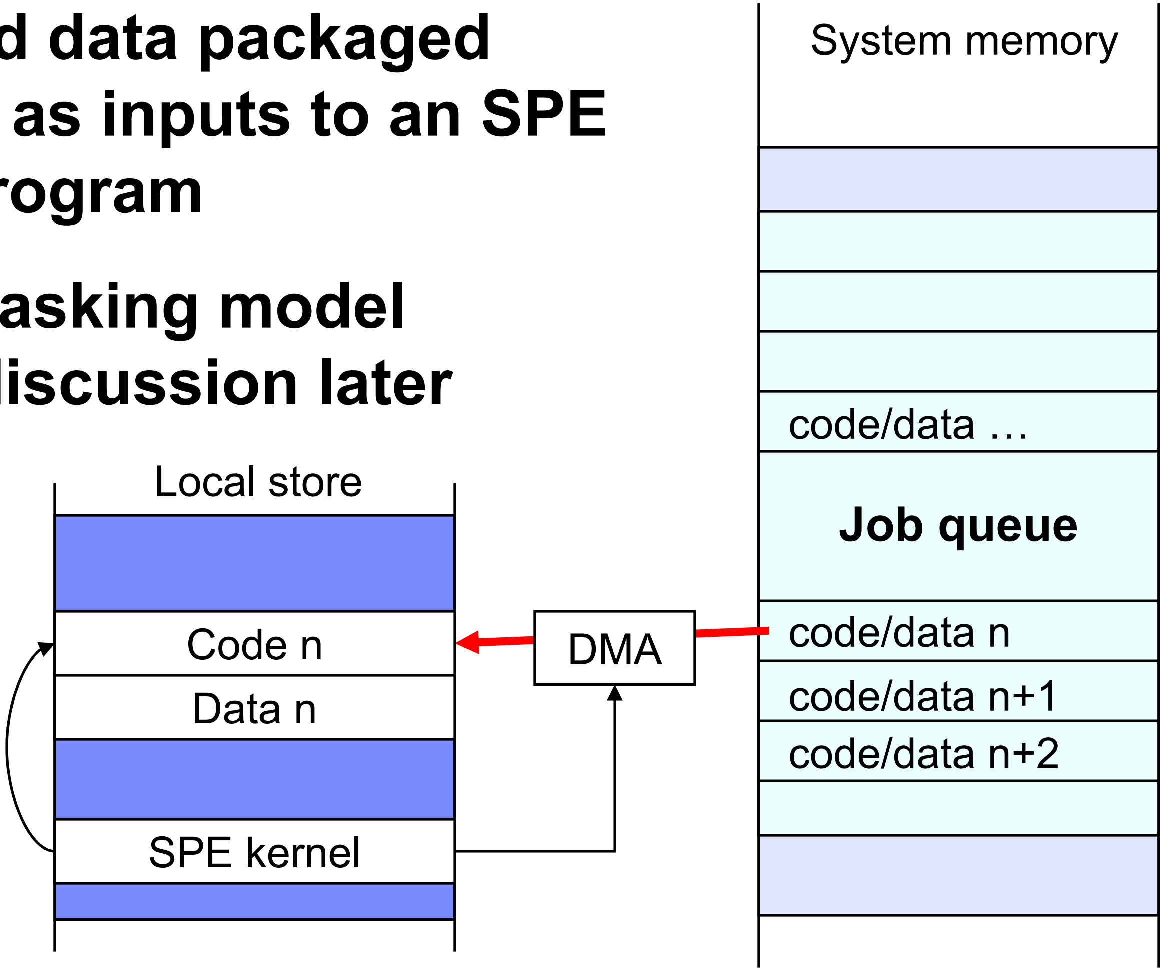
Large single-SPE programming models

- **System memory as secondary memory store**
  - Manual loading of plug-in into code buffer
    - Plug-in framework libraries
  - Automatic software-managed code overlay
    - Compiler generated overlaying code



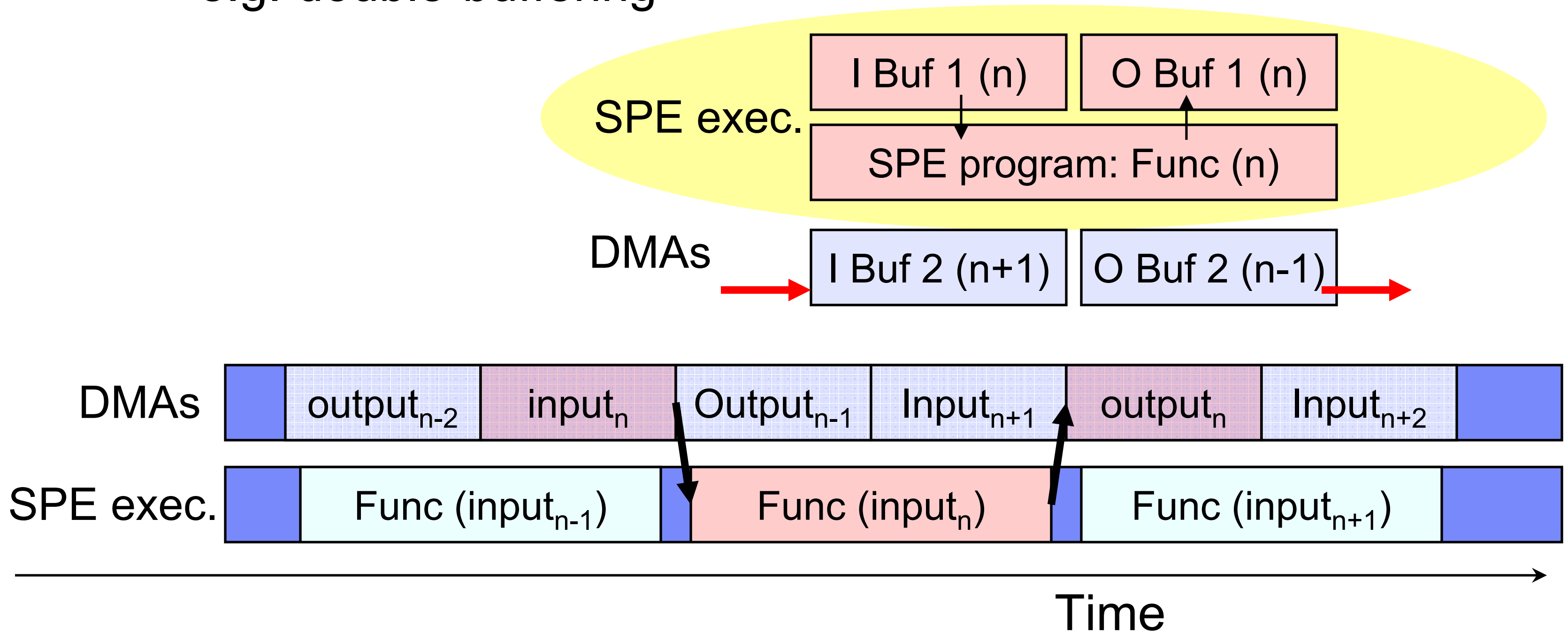
Large single-SPE prog. models – Job Queue

- **Code and data packaged together as inputs to an SPE kernel program**
- **A multi-tasking model – more discussion later**



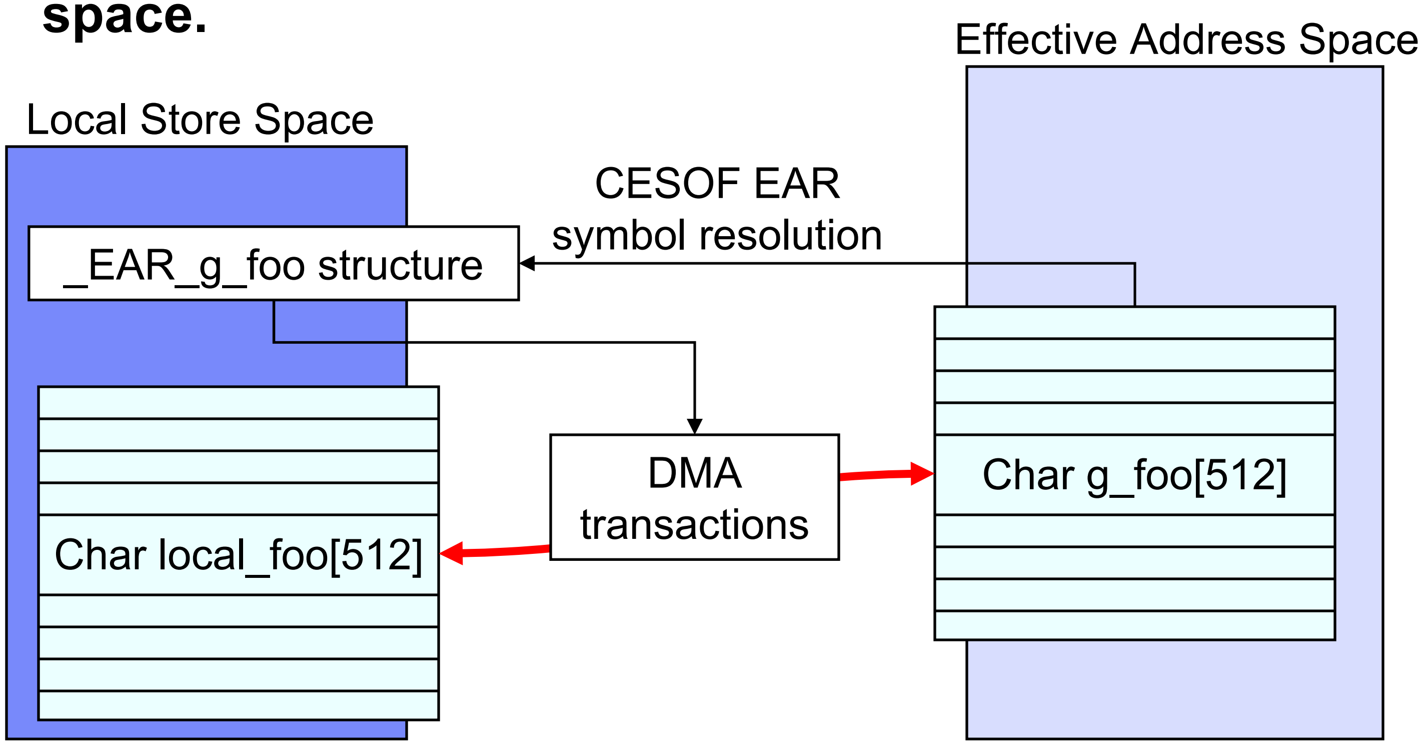
Large single-SPE programming models - DMA

- **DMA latency handling is critical to overall performance for SPE programs moving large data or code**
- **Data pre-fetching is a key technique to hide DMA latency**
  - e.g. double-buffering



Large single-SPE programming models - CESOF

- **Cell Embedded SPE Object Format (CESOF) and PPE/SPE toolchains support the resolution of SPE references to the global system memory objects in the effective-address space.**



Parallel programming models

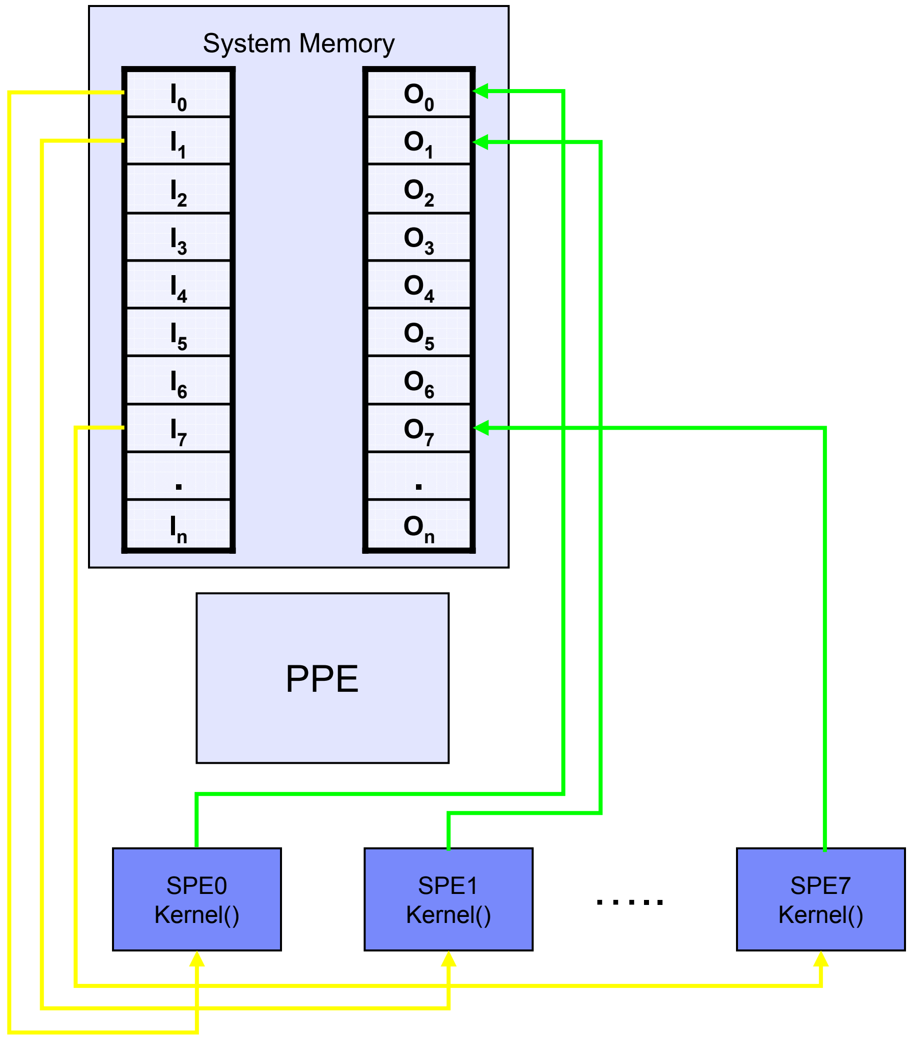
- **Traditional parallel programming models applicable**
- **Based on interacting single-SPE programs**
- **Parallel SPE program synchronization mechanism**
  - Cache line-based MFC atomic update commands similar to the PowerPC lwarx, ldarx, stwcx, and stdcx instructions
  - SPE input and output mailboxes with PPE
  - SPE signal notification / register
  - SPE events and interrupts
  - SPE busy poll of shared memory location

Parallel programming models – Shared Memory

- **Access data by address**
  - Random access in nature
- **CESOF support for shared effective-address variables**
- **With proper locking mechanism, large SPE programs may access shared memory objects located in the effective-address space**
- **Compiler OpenMP support**

Parallel programming models – Streaming

- **Large array of data fed through a group of SPE programs**
- **A special case of job queue with regular data**
- **Each SPE program locks on the shared job queue to obtain next job**
- **For uneven jobs, workloads are self-balanced among available SPEs**

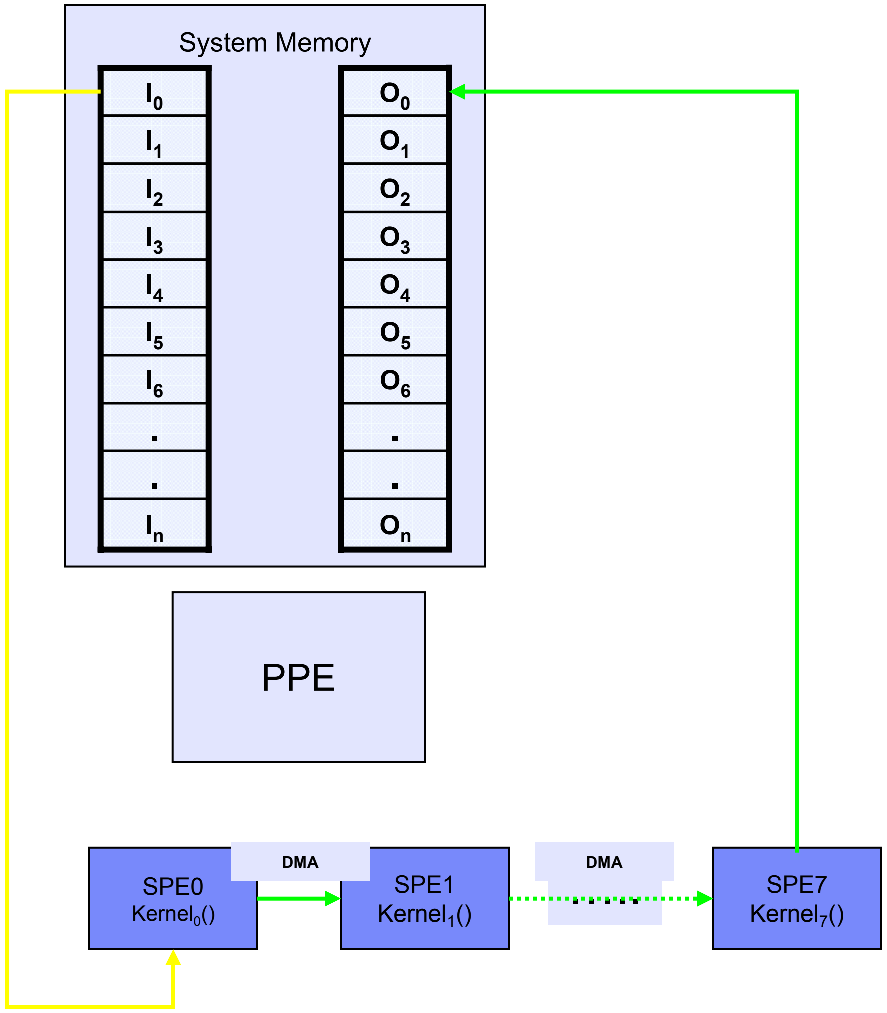


Parallel programming models – Message Passing

- **Access data by connection**
  – Sequential in nature
- **Applicable to SPE programs where addressable data space only spans over local store**
- **The message connection is still built on top of the shared memory model**
- **Compared with software-cache shared memory model**
  – More efficient runtime is possible, no address info handling overhead once connected
  – LS to LS DMA optimized for data streaming through pipeline model

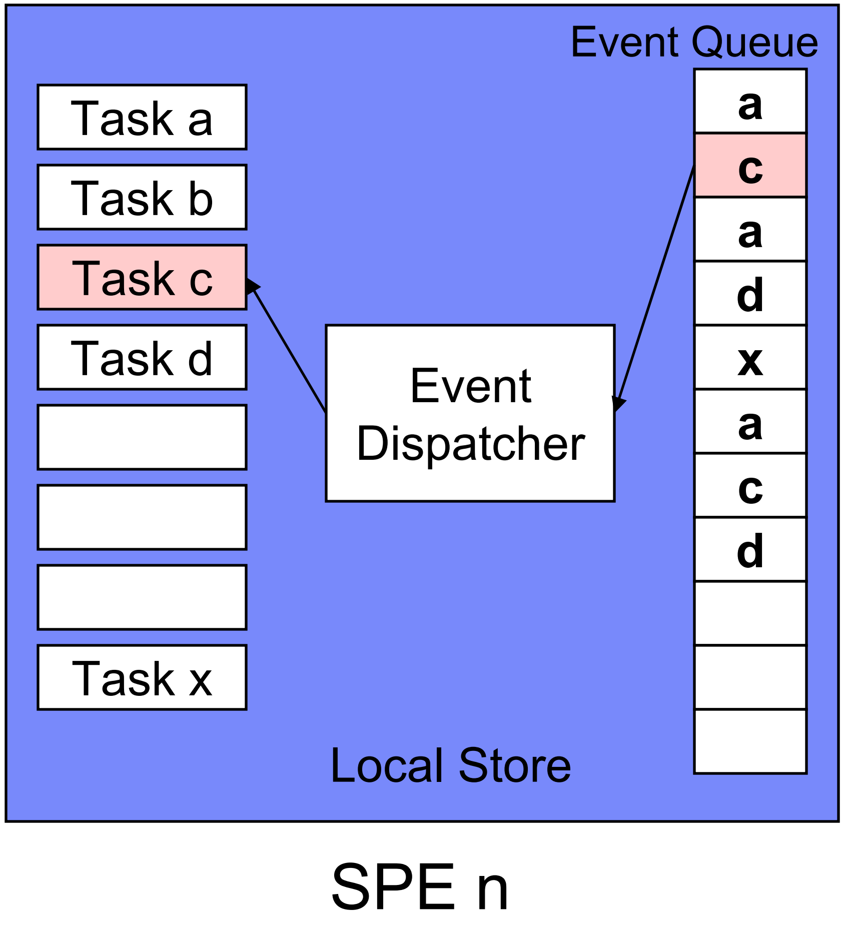
Parallel programming models – Pipeline

- **Use LS to LS DMA bandwidth, not system memory bandwidth**
- **Flexibility in connecting pipeline functions**
- **Larger collective code size per pipeline**
- **Load-balance is harder**



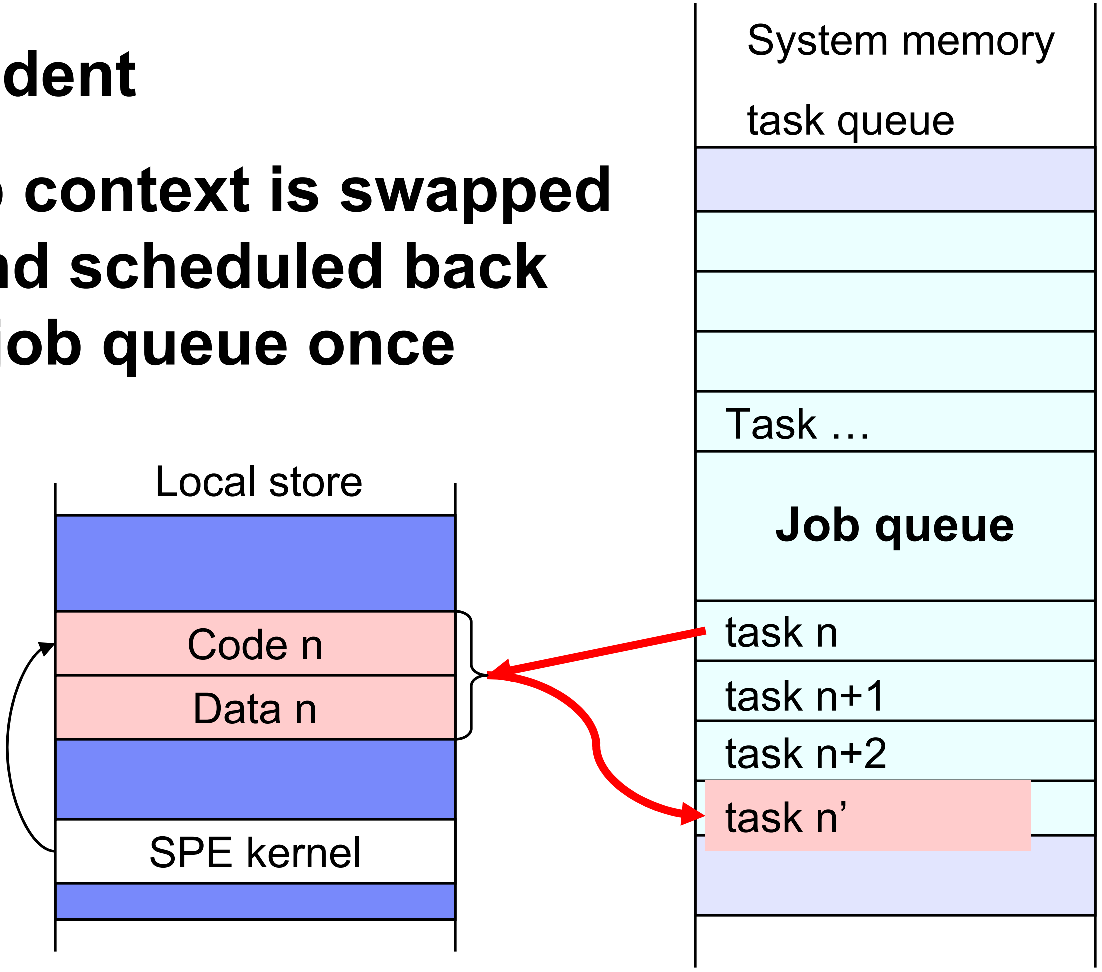
Multi-tasking SPEs – LS resident multi-tasking

- **Simplest multi-tasking programming model**
- **No memory protection among tasks**
- **Co-operative, Non-preemptive, event-driven scheduling**



Multi-tasking SPEs – Self-managed multi-tasking

- **Non-LS resident**
- **Blocked job context is swapped out of LS and scheduled back later to the job queue once unblocked**



Multi-tasking SPEs – Kernel managed

- **Kernel-level SPE management model**
  - SPE as a device resource
  - SPE as a heterogeneous processor
  - SPE resource represented as a file system
- **SPE scheduling and virtualization**
  - Maps running threads over a physical SPE or a group of SPEs
  - More concurrent logical SPE tasks than the number of physical SPEs
  - High context save/restore overhead
    - favors run-to-completion scheduling policy
  - Supports pre-emptive scheduling when needed
  - Supports memory protection

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

Programming Model Final Points

- A proper programming model reduces development cost while achieving higher performance
- Programming frameworks and abstractions help with productivity
- Mixing programming models are common practice
- New models may be developed for particular applications.
- With the vast computational capacity, it is not hard to achieve a performance gain from an existing legacy base
- Top performance is harder
- Tools are critical in improving programmer productivity

IBM Microelectronics Division
1580 Route 52, Bldg. 504
Hopewell Junction, NY 12533-6351

The IBM home page is http://www.ibm.com
The IBM Microelectronics Division home page is
http://www.chips.ibm.com