



Andreas Heinig¹, Jochen Strunk¹, Wolfgang Rehm¹, Heiko Schick²

¹ Chemnitz University of Technology, Germany ² IBM Deutschland Entwicklung GmbH, Germany

{heandr,sjoc,rehm}@informatik.tu-chemnitz.de, schickhj@de.ibm.com

Advance Heterogenous Computing Research

Case Study

- Tight Integration of Cell/B.E. (Cell) into AMD Opteron Ecosystem
- based on AMD's Open platform for system builders "Torrenza" as well as
- IBM Software Development Kit (SDK) for Multicore Acceleration V 3.0

Architectural Challenge

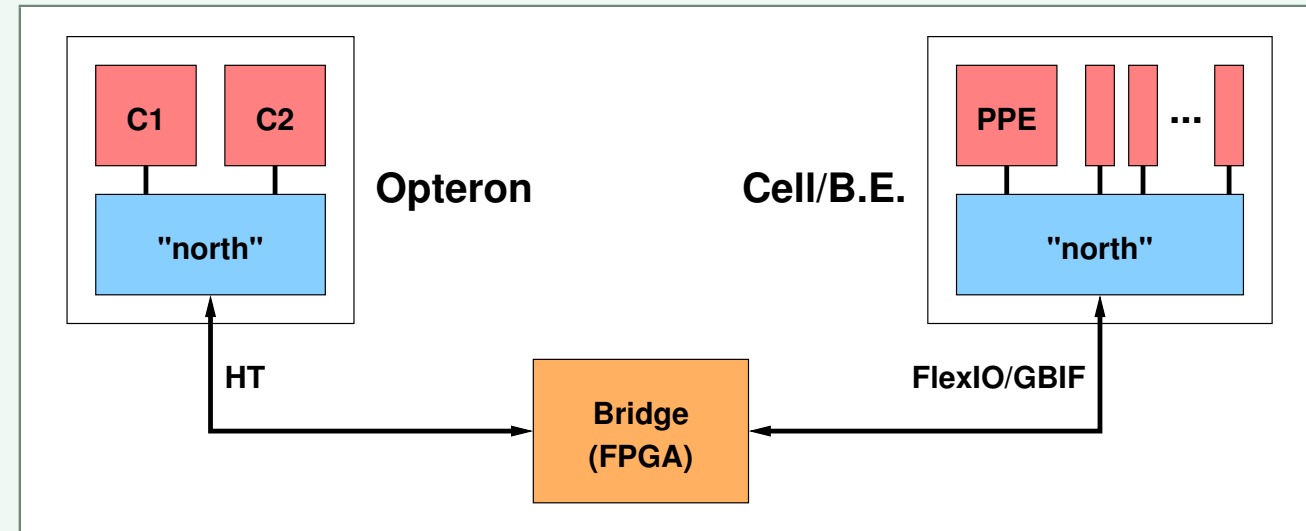
- Proof of concept that Cell/B.E. and Opteron can closely work together (even coherent)
- Accomplish direct coupling of AMD Hypertransport (HT) interface with Cell FlexIO interface via a bridge device
- Our approach: exporting Cell's memory-mapped-IO (MMIO) SPE registers as well as SPE's localstores to Opteron

System Software Challenge

- IBM SDK for Multicore Acceleration Version 3.0 is state of the art but
- the included Data Communication and Synchronization Library (DaCS) on Hybrid does not allow for directly accessing Cell SPE's from Opteron
- Our approach: Introduce "Remote SPUFS" to directly access SPE's

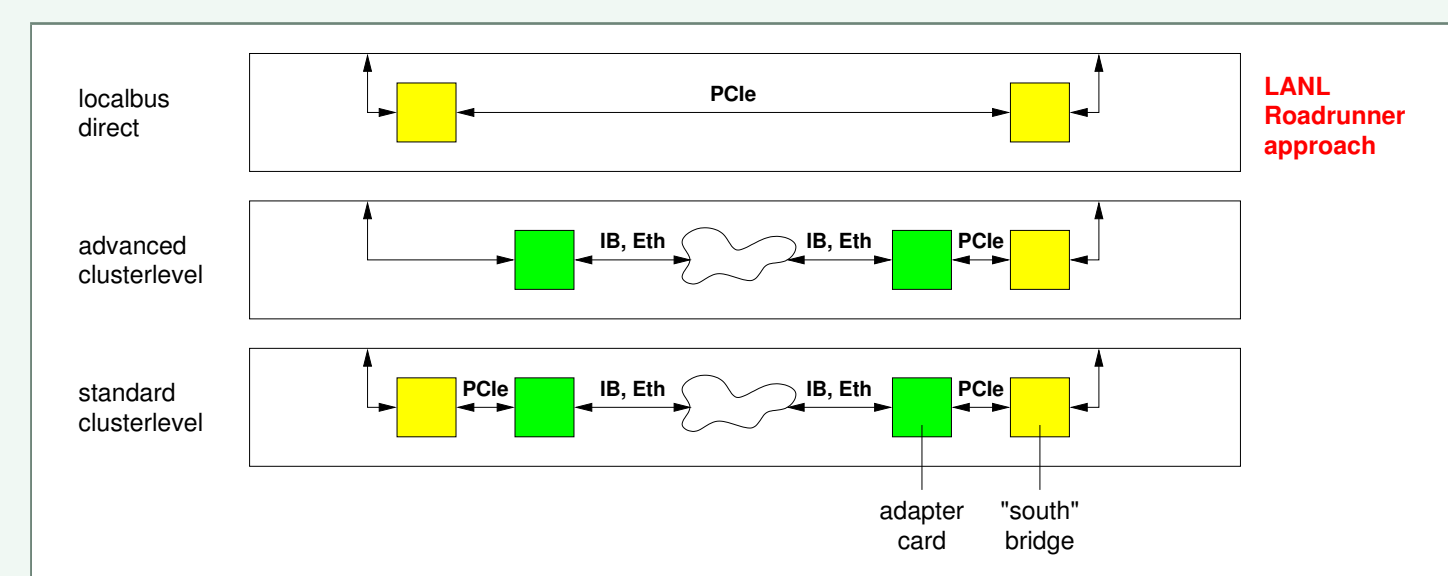
Interconnection Architecture

Direct coupling via bridge device

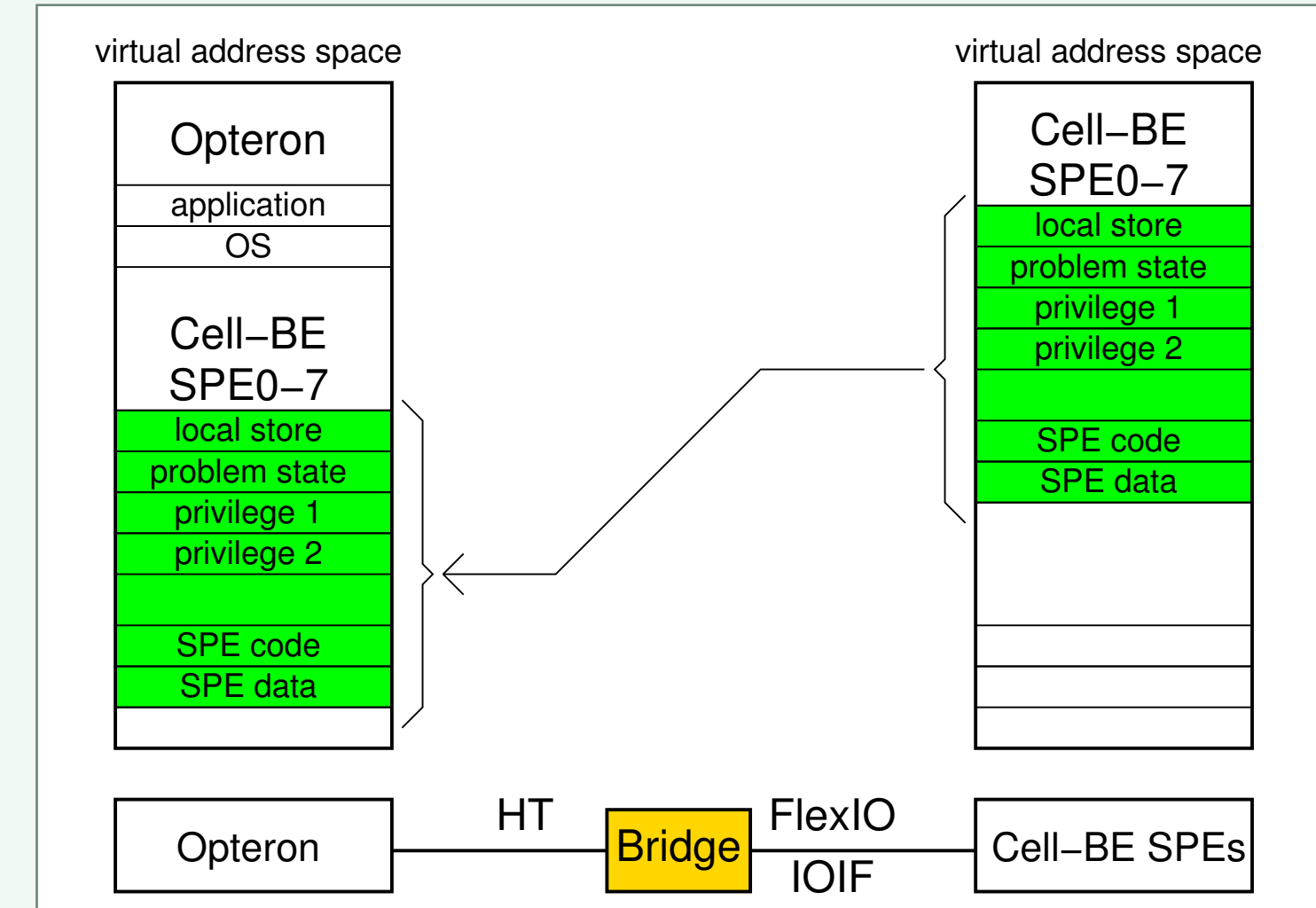


- coupling Opteron's Hypertransport interface (open standard) and Cell's FlexIO/IOIF using the "Global Bus Interface (GBIF) Architecture" (IBM proprietary)
- Proof of concept is accomplished by using a Xilinx FPGA that implements an appropriate Request Translation Engine

Related approaches



Memory mapping scheme



- exporting Cells memory-mapped-IO (MMIO) SPE registers as well as SPE's local stores to Opteron
- local store: 128x128bit registers for each SPE
- problem state: DMA setup and status, SPE run-control/status, SPE signal notification, SPE next program counter and mailbox register
- privilege area 1: MFC SR1 state, MFC ADR, MFC Interrupt status, MFC DAR, DSISR register
- privilege area 2: MFC MMU control, SPE channel control, MFC control, SPE debug enable registers

RSPUFS: System Software

Target

- integration of the Cell/B.E. processor into an AMD Opteron system

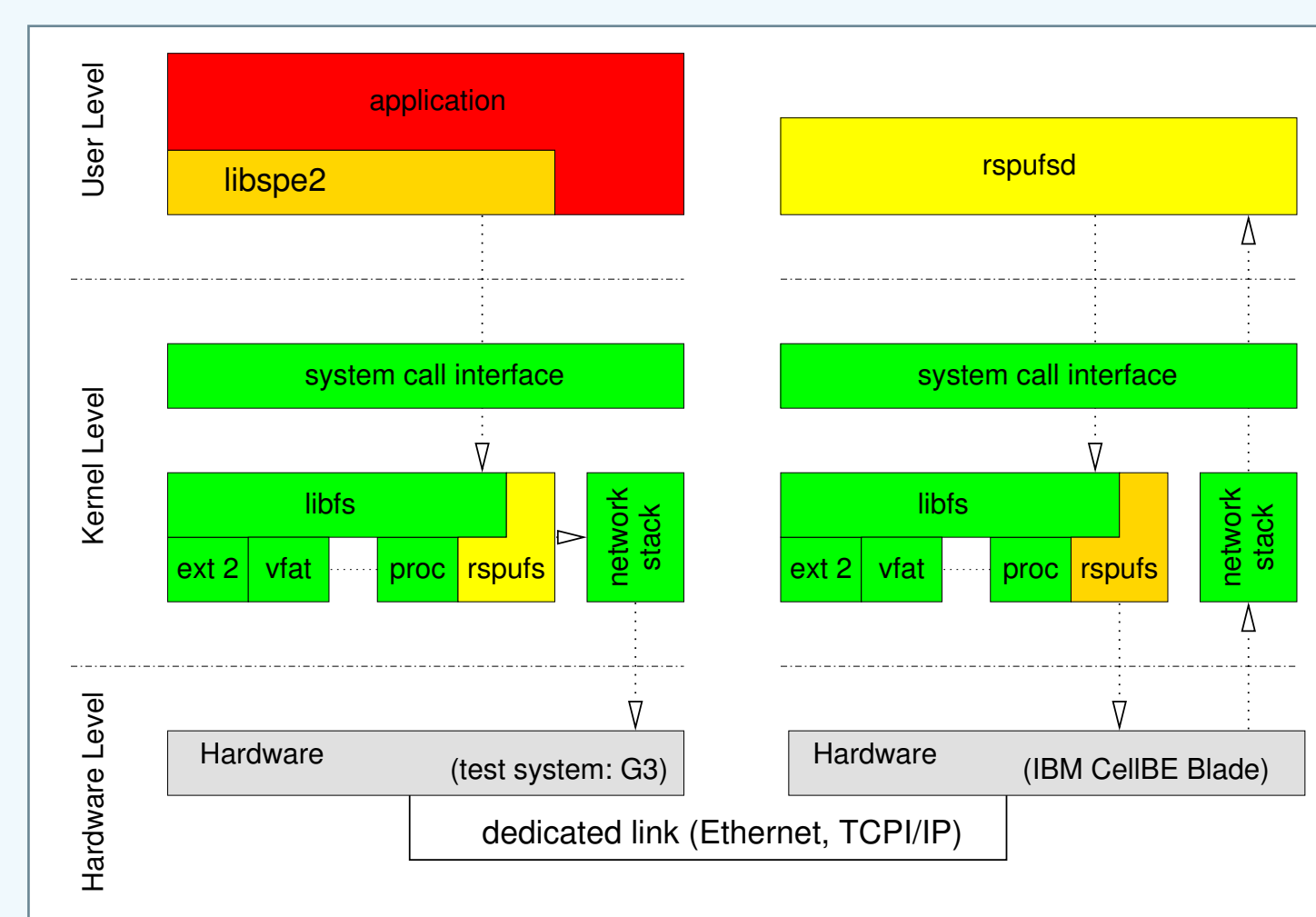
Tasks

- focus on an acceleration model where SPE code is provided in application or middle ware libraries
- analyzing the SPUFS
- integrate SPUFS into Opteron Linux kernel with TCP/IP
- using closer coupling via InfiniBand or PCIe
- using tight coupling via a system bus RDMA interface (optional)

RSPUFS: concept

- virtual file system mounted on /spu by convention
 - mapping of hardware resources
 - every directory in spu fs refers to a logical SPU context
 - only one operation on the root directory is allowed: *spu.create*
 - each context contains a fixed set of files: (some are)
 - **mem**
local store memory of the SPE
 - **mbox**
ibox
wbox
user space side of the mailbox
 - ...
 - the SPE gets working when calling *spu.run* suspending the calling thread
- ⇒ implements the interface of spu fs ("the Linux programming model for Cell" introduced from IBM 2005) without direct hardware access

RSPUFS: Software Stack current work, first step



- the dotted arrow on the figure shows the communication to the SPU for an example mailbox write
- libspe2 is a wrapper library (provided by IBM) for easier communication with the SPU's
- for the first step our test hardware is a PowerBook G3, so we can avoid porting libspe2 to the Opteron
- for better debugging, all the communication stuff is handled by an user space program called **rspufsd**
- the **rspufsd** uses the unmodified spu file system, so that all errors are probably based on our own modules
- with the usage of **MMIO** we may be able to replace the operation system on the Cell side with a small interrupt handler

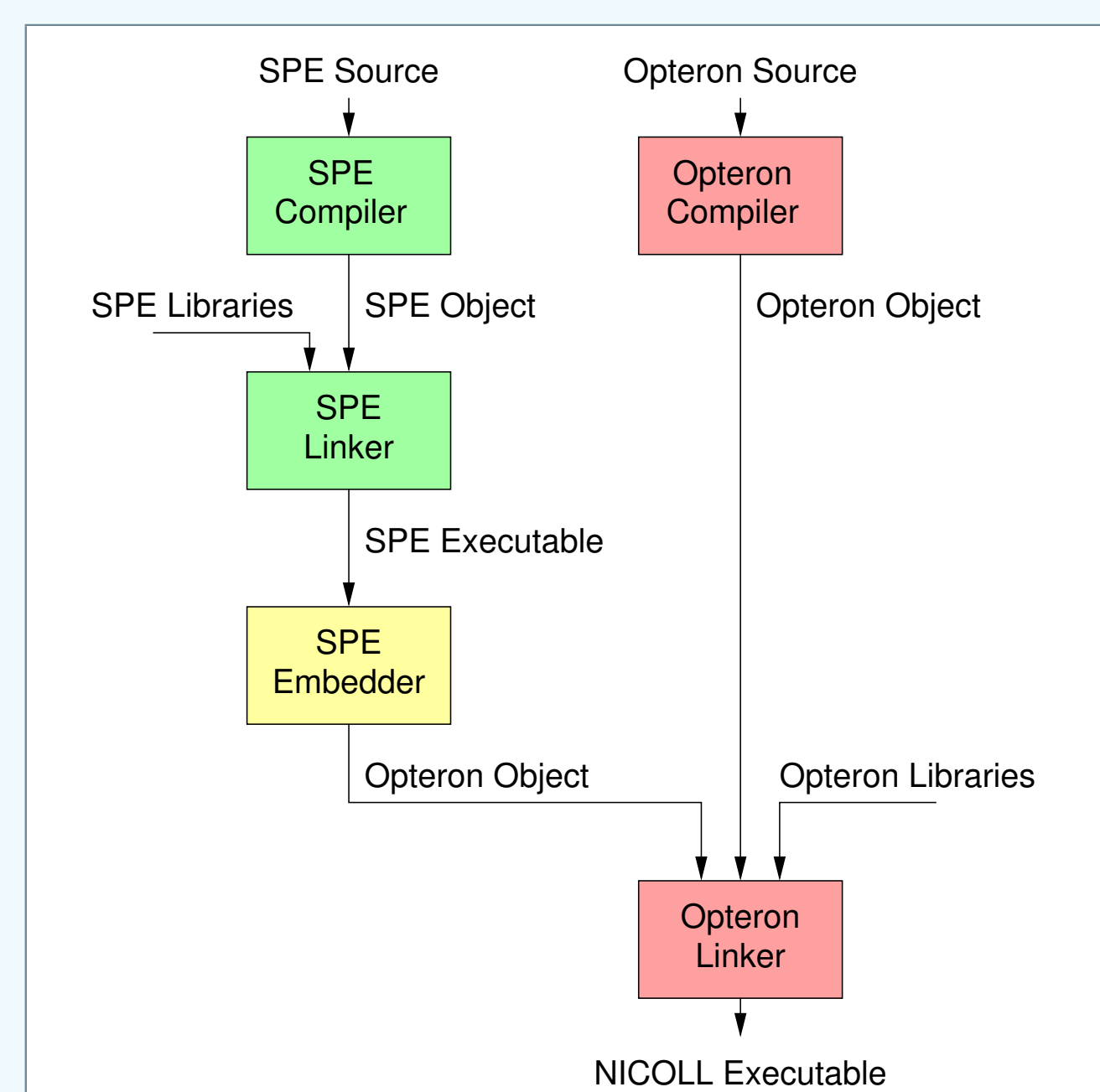
RSPUFS: Working Sequence

	Opteron (current G3)	Cell
1 mount rspufs on /spu	issue mount command.	-
2 create context <i>spu.create</i>	establish a new connection to the Cell and submit context parameters (name, flags) return the context file descriptor to the application	create a spu fs context an submit the content to the Opteron
3 open files <i>open</i>	send open file request and map the returned descriptor to the local	return the file descriptor from the Cell side
4 do something <i>read, write, ...</i>	not implemented	not implemented
5 start executing <i>spu.run, ...</i>	not implemented	not implemented
6 close files <i>close</i>	send close file request with Opteron fd	close opened file
7 destroy context	implicit when closing the context fd (program termination or explicit close) disconnect form Cell, removing context from rspufs	close all (eventually) open files and call close on context fd

⇒ live demo on the PowerBook G3

RSPUFS: Tool Chain

- to support applications linking against libspe2 it's necessary to adjust the original tool chain (introduced with spu fs)
- the tool chain embeds the SPE code in the Opteron binary creating one executable file



- using original tool chain for SPE code (green)
- replace the PPE tool chain by a standard Opteron tool chain (red)
- patching the embedder (yellow) to produce an Opteron object file

RSPUFS: Implementation Status

Current

1. setup the basics of a new file system (super block, inodes)
2. spu fs specific system calls
 - *spu.create* ✓
 - *spu.run* ✗
3. file system calls
 - *open / close* ✓
 - *read / write* ✗
 - *mmap* ✗

Joined Steps

- after finishing those system calls we are able running the first test applications
- it should be possible to link the application against libspe2 and execute it on the PowerBook Cell/B.E. Hybrid

Further Work

- in the next steps we want to implement shared memory support
- therefore we need RDMA features for acceptable speed
- linking both systems with PCIe (like Road Runner) should be sufficient
- but with tightly coupling we are able to get the most speedup
- also this can support **cache coherency** if we need it later

Memory mapping advantages for RSPUFS

- direct access to SPE avoids message passing
- with mapping of the XDR we can avoid to introduce a special function to gain access to the Opteron memory from the Cell side:
 - the application can store any calculation relevant data to XDR and loads the results from XDR
 - for those actions we may need cache coherency which is only possible with our hardware approach

RSPUFS vs. DaCSH

DaCSH

- Data Communication and Synchronisation on Hybrid library
- state of the art in SDK 3 (LANL Road Runner)
- provides a set of services for developing heterogeneous applications
- DaCSH services are implemented as a set of APIs
- all components are executed in user space

Comparing DaCSH with RSPUFS

	DaCSH	RSPUFS
known environment	fully known	particularly known
access to SPE	no access	full access
OS modification Cell	no	no
OS modification Opteron	no	yes
interface	new interface	same interface
needed modification on existing applications	many	possible less modifications

Conclusion

- DaCSH is a completely other basic approach for heterogeneous programming as *rspufs* or *spufs*.
- the advantage of DaCSH is the usability not only for x86-Cell, but also for a wide range of possible heterogeneous systems
- but we want **direct hardware access** to avoid communication overheads and kernel entries worsening performance
- thus that we are coupling the system buses we could achieve better latency and throughput