

# Evaluating the Portability of UPC to the Cell Broadband Engine

Dipl. Inform. Ruben Niederhagen

June 4, 2008  
JSC Cell Meeting



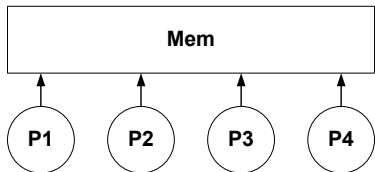
**CHAIR FOR OPERATING SYSTEMS**

*Univ.-Prof. Dr. habil. Thomas Bemerl*

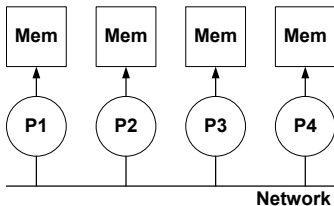


- Introduction
  - UPC
  - Cell
  
- UPC on Cell
  - Mapping
  - Compiler and Runtime System
  
- Optimization: Software Managed Cache
  
- Conclusion

*Unified Parallel C* (UPC) is a C-language derivate for parallel computing.



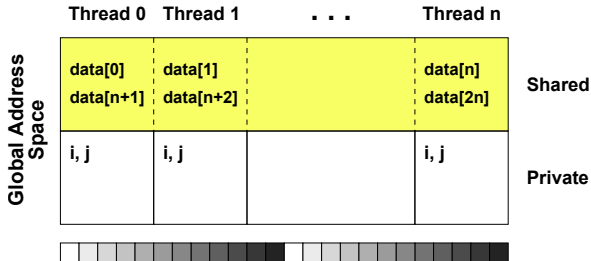
Shared Memory Architecture  
 ⇒ OpenMP



Distributed Memory Architecture  
 ⇒ MPI

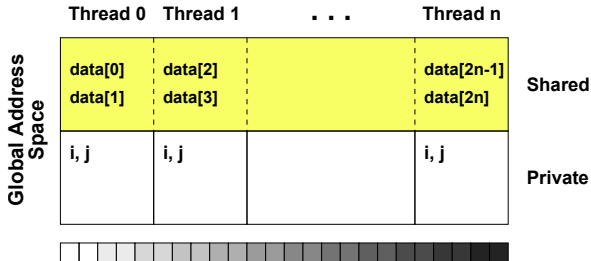
## Shared Data

```
int i, j;
shared int data[2*n]; // n number of threads
```



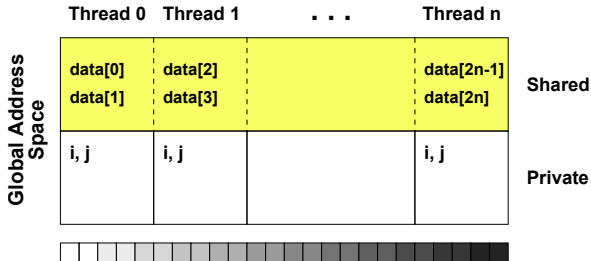
## Shared Data: Blocking

```
int i, j;
shared [2] int data[2*n]; // n number of threads
```



## Shared Data: Blocking

```
int i, j;
shared [2] int data[2*n]; // n number of threads
```



- Shared data can be directly accessed by all threads.
- UPC takes care for remote accesses.

## Work Sharing

the loop:

```
upc_forall (int i = 0; i < n; i++; i)
    data[i] = MYTHREAD;
```

translates to:

```
for (int i = 0; i < n; i++)
    if (i % THREADS == MYTHREAD)
        data[i] = MYTHREAD;
```

## Work Sharing

equally:

```
upc_forall (int i = 0; i < n; i++; &data[i])
    data[i] = MYTHREAD;
```

translates to:

```
for (int i = 0; i < n; i++)
    if (&data[i] is an affine address)
        data[i] = MYTHREAD;
```



## Work Sharing

equally:

```
upc_forall (int i = 0; i < n; i++; &data[i])
    data[i] = MYTHREAD;
```

translates to:

```
for (int i = 0; i < n; i++)
    if (&data[i] is an affine address)
        data[i] = MYTHREAD;
```

- Distribute work over several threads easily.
- Utilize data locality.

## Example: Matrix Multiplication

```

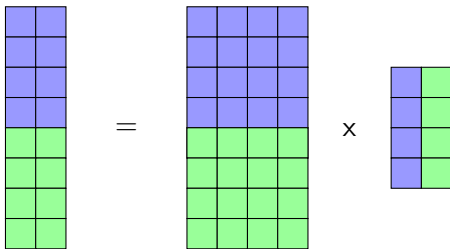
shared [N*P /THREADS] int a[N][P];
shared [N*M /THREADS] int c[N][M];
// a and c are row-wise blocked shared matrices

shared[M/THREADS] int b[P][M];
//column-wise blocking

void main (void) {
    int i, j , l; // private variables
    upc_forall(i = 0 ; i<N ; i++; &c[i][0])
    {
        for (j=0 ; j<M ;j++)
        {
            c[i][j] = 0;
            for (l= 0 ; l<P ; l++)
                c[i][j] += a[i][l]*b[l][j];
        }
    }
}

```

## Example: Matrix Multiplication

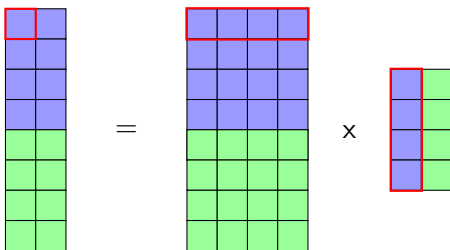


```

upc_forall(i = 0 ; i<N ; i++; &c[i][0])
{
    for (j=0 ; j<M ; j++)
    {
        c[i][j] = 0;
        for (l= 0 ; l<P ; l++)
            c[i][j] += a[i][l]*b[l][j];
    }
}

```

## Example: Matrix Multiplication

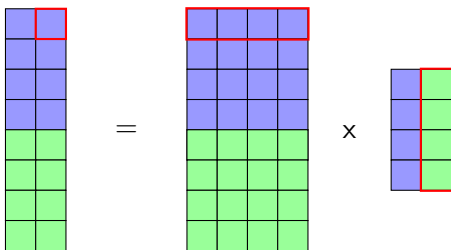


```

upc_forall(i = 0 ; i<N ; i++; &c[i][0])
{
    for (j=0 ; j<M ; j++)
    {
        c[i][j] = 0;
        for (l= 0 ; l<P ; l++)
            c[i][j] += a[i][l]*b[l][j];
    }
}

```

## Example: Matrix Multiplication

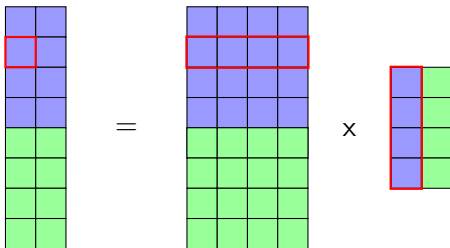


```

upc_forall(i = 0 ; i<N ; i++; &c[i][0])
{
    for (j=0 ; j<M ; j++)
    {
        c[i][j] = 0;
        for (l= 0 ; l<P ; l++)
            c[i][j] += a[i][l]*b[l][j];
    }
}

```

## Example: Matrix Multiplication

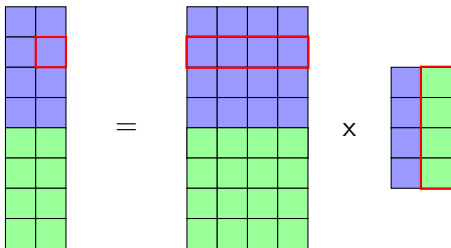


```

upc_forall(i = 0 ; i<N ; i++; &c[i][0])
{
    for (j=0 ; j<M ; j++)
    {
        c[i][j] = 0;
        for (l= 0 ; l<P ; l++)
            c[i][j] += a[i][l]*b[l][j];
    }
}

```

## Example: Matrix Multiplication

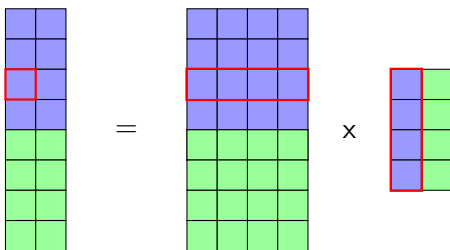


```

upc_forall(i = 0 ; i<N ; i++; &c[i][0])
{
    for (j=0 ; j<M ; j++)
    {
        c[i][j] = 0;
        for (l= 0 ; l<P ; l++)
            c[i][j] += a[i][l]*b[l][j];
    }
}

```

## Example: Matrix Multiplication



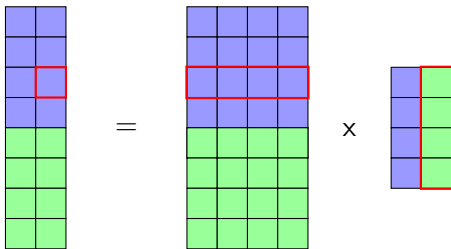
```

upc_forall(i = 0 ; i<N ; i++; &c[i][0])
{
    for (j=0 ; j<M ; j++)
    {
        c[i][j] = 0;
        for (l= 0 ; l<P ; l++)
            c[i][j] += a[i][l]*b[l][j];
    }
}

```



## Example: Matrix Multiplication

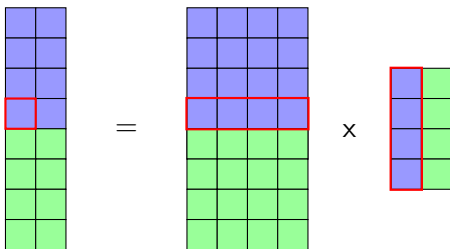


```

upc_forall(i = 0 ; i<N ; i++; &c[i][0])
{
    for (j=0 ; j<M ; j++)
    {
        c[i][j] = 0;
        for (l= 0 ; l<P ; l++)
            c[i][j] += a[i][l]*b[l][j];
    }
}

```

## Example: Matrix Multiplication

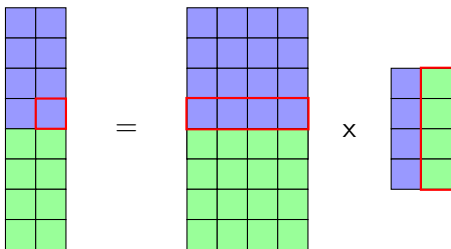


```

upc_forall(i = 0 ; i<N ; i++; &c[i][0])
{
    for (j=0 ; j<M ; j++)
    {
        c[i][j] = 0;
        for (l= 0 ; l<P ; l++)
            c[i][j] += a[i][l]*b[l][j];
    }
}

```

## Example: Matrix Multiplication



```

upc_forall(i = 0 ; i<N ; i++; &c[i][0])
{
    for (j=0 ; j<M ; j++)
    {
        c[i][j] = 0;
        for (l= 0 ; l<P ; l++)
            c[i][j] += a[i][l]*b[l][j];
    }
}

```

## And Some More Stuff

- Dynamic Shared Memory Allocation
- Synchronization
  - ▶ Barriers
  - ▶ Split-Phase Barriers
  - ▶ Locks
- UPC Libraries
  - ▶ Collective Library
  - ▶ I/O Library
- ...

Memory Consistency

```
shared int flag = 0;  
shared int data[2];
```

**Thread 0:**

```
data[0] = 42;  
data[1] = 23;  
flag = 1;
```

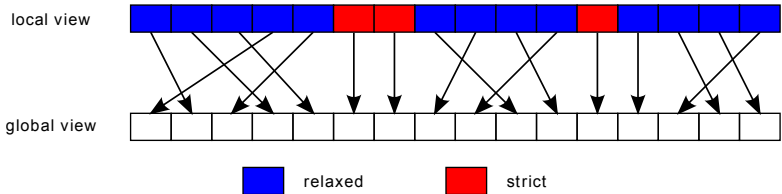
**Thread 1:**

```
while(flag == 0);  
data[0] += data[1];
```

## Memory Consistency

The memory consistency model of UPC defines two consistency types:

- **strict**: enforces completion of any memory access preceding a strict access, delays any subsequent access until the strict access has been finished
- **relaxed**: a sequence of memory accesses which are relaxed is allowed to be reordered as long as interdependencies are respected



Memory Consistency

```
strict shared int flag = 0;  
relaxed shared int data[2];
```

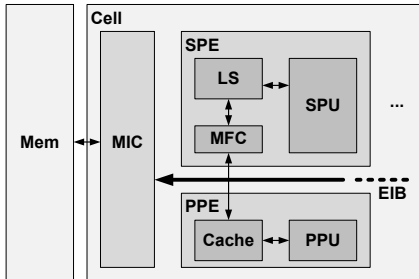
**Thread 0:**

```
data[0] = 42;  
data[1] = 23;  
flag = 1;
```

**Thread 1:**

```
while(flag == 0);  
data[0] += data[1];
```

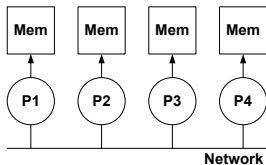
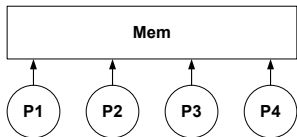
The Cell processor is a hybrid multicore processor:



- Power Processing Element
  - ▶ Cache
  - ▶ Power Processing Unit
- Synergistic Processing Element
  - ▶ Memory Flow Controller
  - ▶ Local Store
  - ▶ Synergistic Processing Unit
- Element Interconnect Bus
- Memory Interface Controller

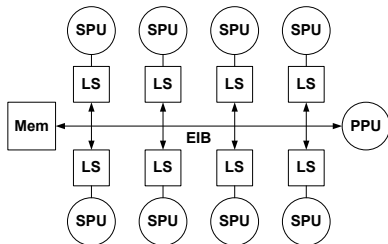


- The PPU runs the operating system and administrative tasks.
- The SPUs handle the parallel workload.
- The PPU has full transparent access to main memory.
- The SPUs access their LS.
- Data have to be transferred from main memory to LS explicitly.



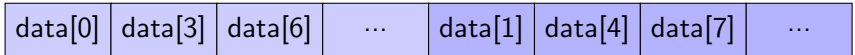
?

- no direct access to main memory  
→ no classical Shared Memory
- LS too small for distributed data  
→ no classical Distributed Memory



## Solution

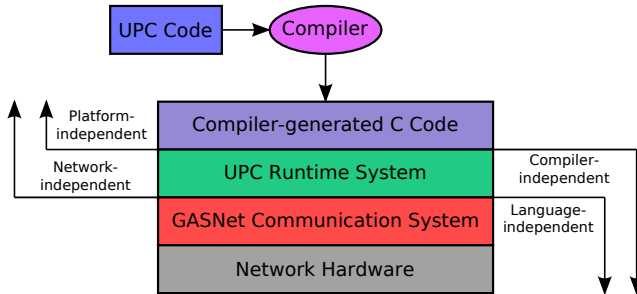
- All shared data resides in main memory.
- Data is transferred between main memory and LS on demand.
- Data partitioning has to be maintained to achieve data affinity:



## Solution

- relaxed:
  - ▶ on read access: transfer data from main memory to LS  
→ blocking transfer
  - ▶ on write access: transfer data from LS to main memory  
→ non-blocking transfer
- strict:
  - ▶ as before but:  
usage of DMA barrier and synchronization mechanisms to  
maintain consistency
 → more expensive

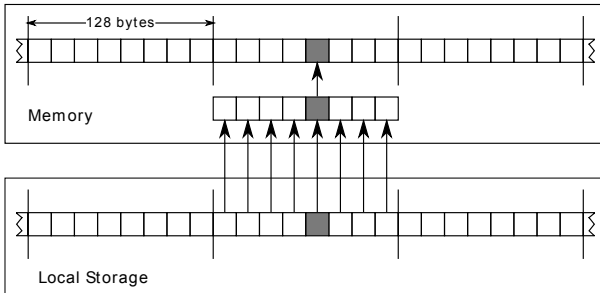
## Berkeley UPC



- use the Berkeley UPC-to-C compiler
- implement the UPC runtime layer for the Cell processor

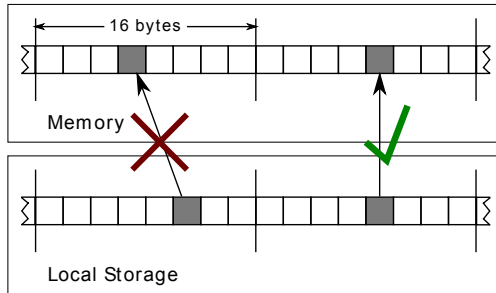
## Disadvantages of the Simple Implementation

- every DMA transfer is parted into aligned 128 byte blocks  
 → inefficient bus utilization, retransmission of sequential data



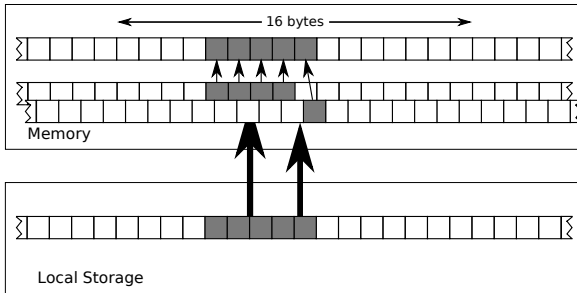
## Disadvantages of the Simple Implementation

- every DMA transfer is parted into aligned 128 byte blocks  
→ inefficient bus utilization, retransmission of sequential data
- data must be aligned equally in a 16byte grid  
→ buffering and copying in LS necessary



## Disadvantages of the Simple Implementation

- every DMA transfer is parted into aligned 128 byte blocks  
→ inefficient bus utilization, retransmission of sequential data
- data must be aligned equally in a 16byte grid  
→ buffering and copying in LS necessary
- data size restricted to 1,2,4,8, or  $n \cdot 16$  bytes  
→ transfers of invalid size must be split





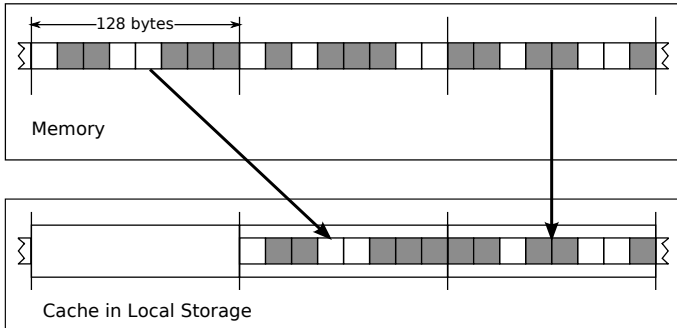
## Disadvantages of the Simple Implementation

- every DMA transfer is parted into aligned 128 byte blocks  
→ inefficient bus utilization, retransmission of sequential data
- data must be aligned equally in a 16byte grid  
→ buffering and copying in LS necessary
- data size restricted to 1,2,4,8, or  $n \cdot 16$  bytes  
→ transfers of invalid size must be split

So - why not use a cache with 128 byte cache lines?

## Cache Line and Cache Frame

Load **Cache Line** to a **Cache Frame** in LS on data access.

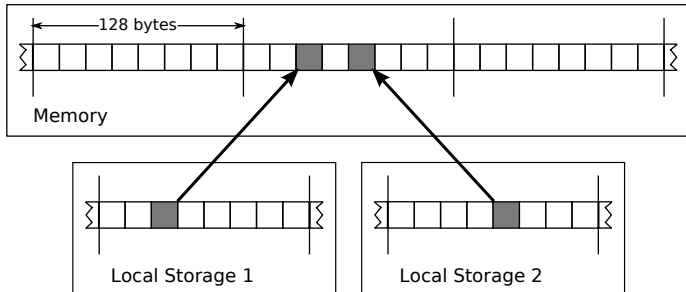


## Cache and Consistency

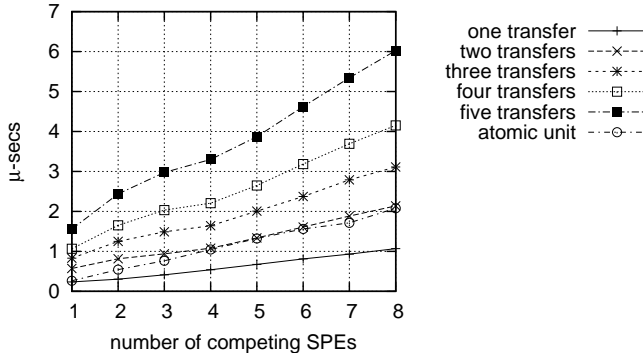
- relaxed access: can be reordered and thus be cached without synchronization
  - strict access: no caching to achieve consistency
- only relaxed accesses to shared data are allowed to be cached  
 ⇒ this avoids further inter-node communication

## Cache Flush

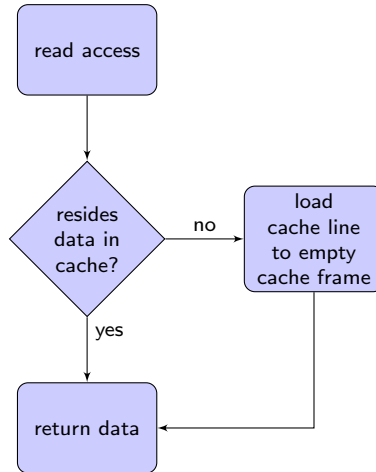
- Cache must be flushed on synchronization points: strict access, UPC barrier, UPC locks
- In contrast to classic HW-caches, only dirty bytes are allowed to be transferred to main memory:



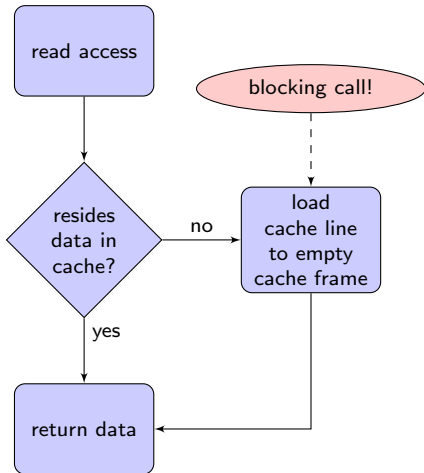
## Cache Flush



## Cache Strategy for Reads

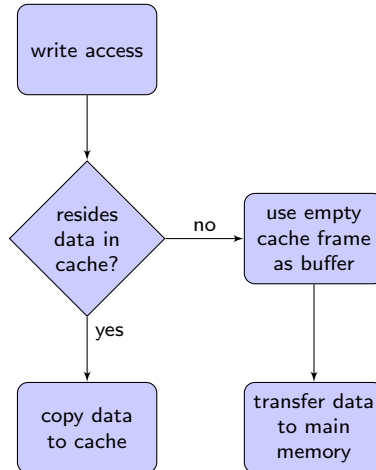


## Cache Strategy for Reads



## Cache Strategies for Writes

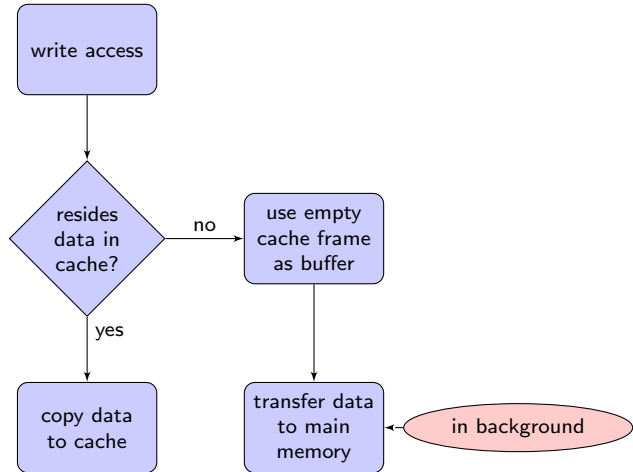
### Direct Write Through





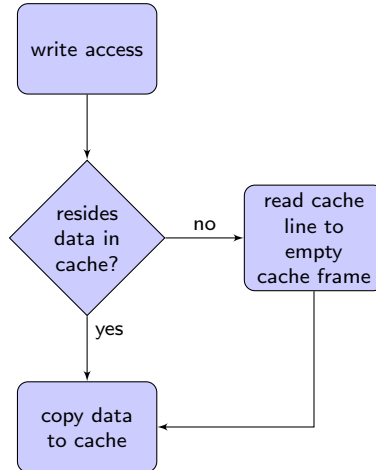
## Cache Strategies for Writes

### Direct Write Through



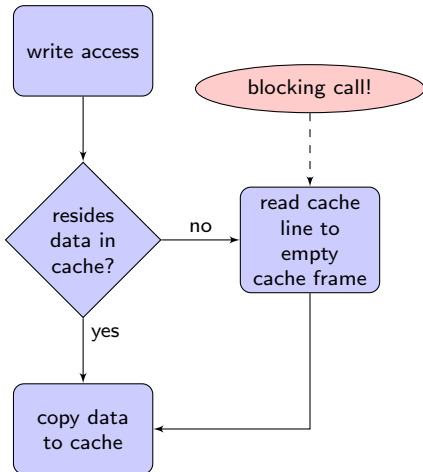
## Cache Strategies for Writes

Bundled Writes With Cache Line Read



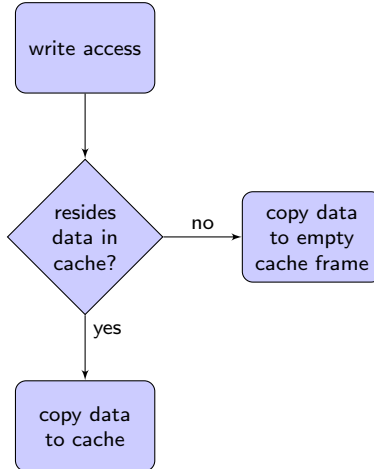
## Cache Strategies for Writes

### Bundled Writes With Cache Line Read



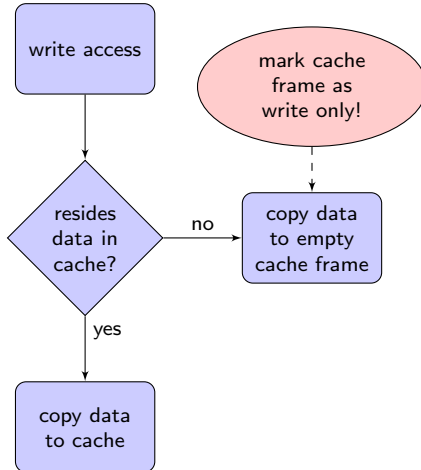
## Cache Strategies for Writes

Bundled Writes With Cache Line Read On Demand



## Cache Strategies for Writes

Bundled Writes With Cache Line Read On Demand



## Cache Strategies for Writes

- Direct Write Through:
  - ▶ advantage: small administrative overhead
  - ▶ disadvantage: bus congestion
- Bundled Writes With Cache Line Read:
  - ▶ advantage: bus unloading
  - ▶ disadvantage: latency on first write
- Bundled Writes With Cache Line Read On Demand:
  - ▶ advantage: fast write on shared data
  - ▶ disadvantage: bus congestion on synchronized cache flush

What did we already do?

→ We thought about all that very carefully.

What will we do next?

→ We are going to

- implement the proposed architecture.
- implement several caching strategies.
- compare and evaluate these strategies.
- extend our approach to clusters of Cell blades.
- integrate code partitioning into the system.

Questions?