CellFS: Taking "DMA" Out Of Cell Programming

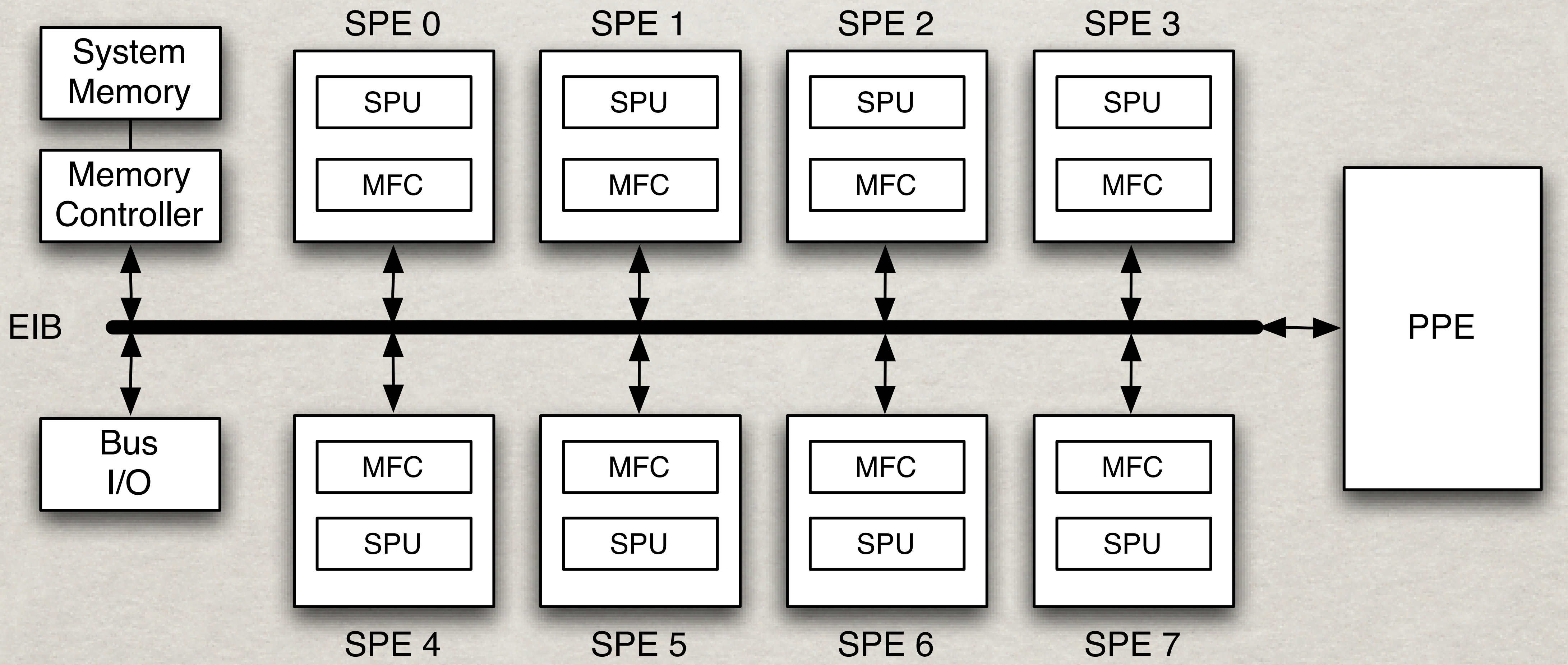
Latchesar Ionkov
Aki Nyrhinen
Andrey Mirtchovski

LANL Publication: LA-UR-07-0016

Overview

- Easy-to-use programming model
- Provides overlapping of communication and computation
- Performance is within 10% of hand-optimized implementations using triple-buffering
- Hides the details of the SPU communications
- Allows prototyping of SPU apps on non-SPU machines

Cell CBE Architecture

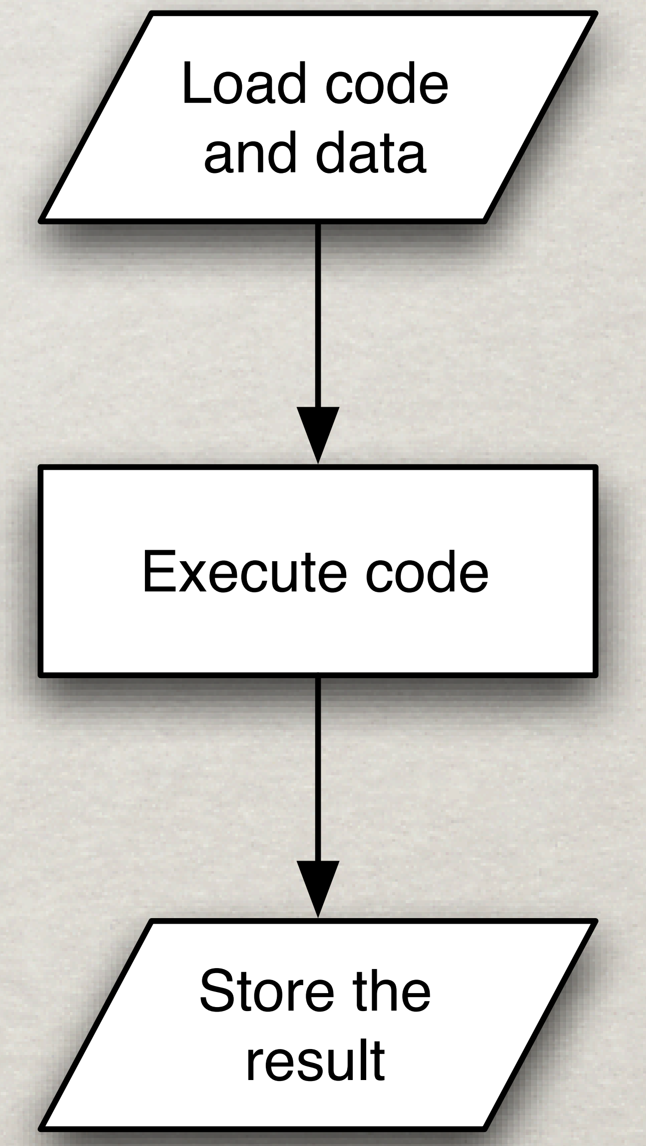


SPE Communication Primitives

- DMA transfers
  - up to 16 simultaneous DMA requests from the SPU
  - up to 8 simultaneous DMA requests from external devices (PPE and other SPEs)
- Mailboxes
  - three mailboxes (blocking inbound, blocking outbound and non-blocking outbound)
- Signal notifications

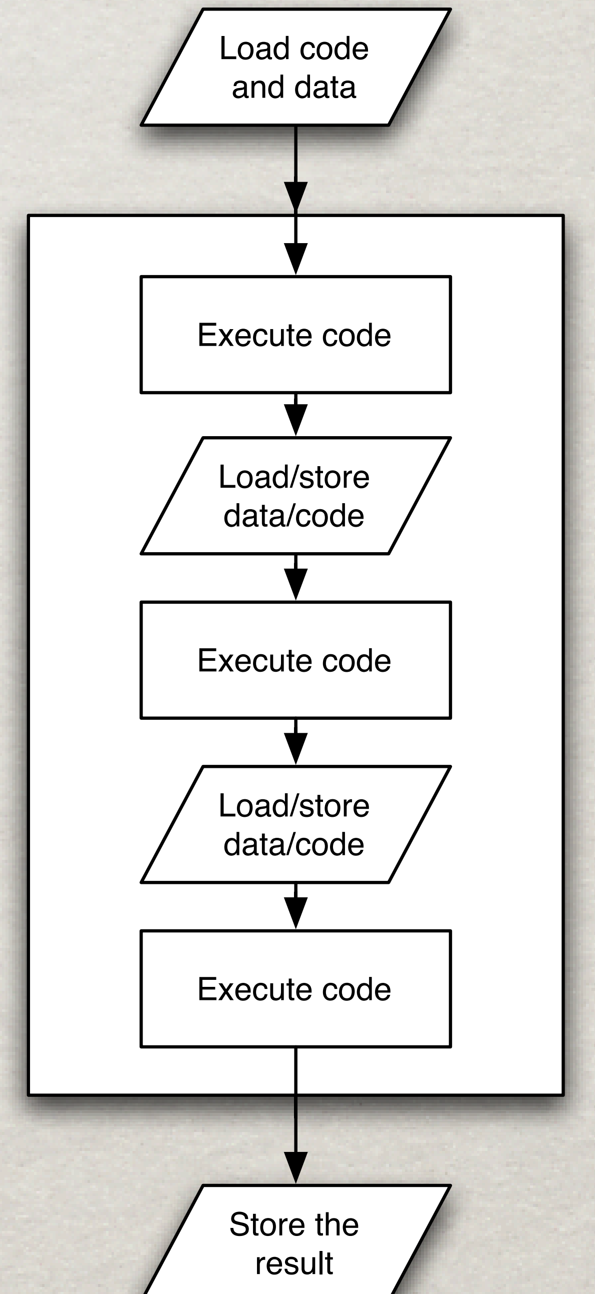
SPU Programming Models

- Code and data fit into 256K
- The code doesn't do any DMA requests
- Easy to port existing code
- One way to implement function offload



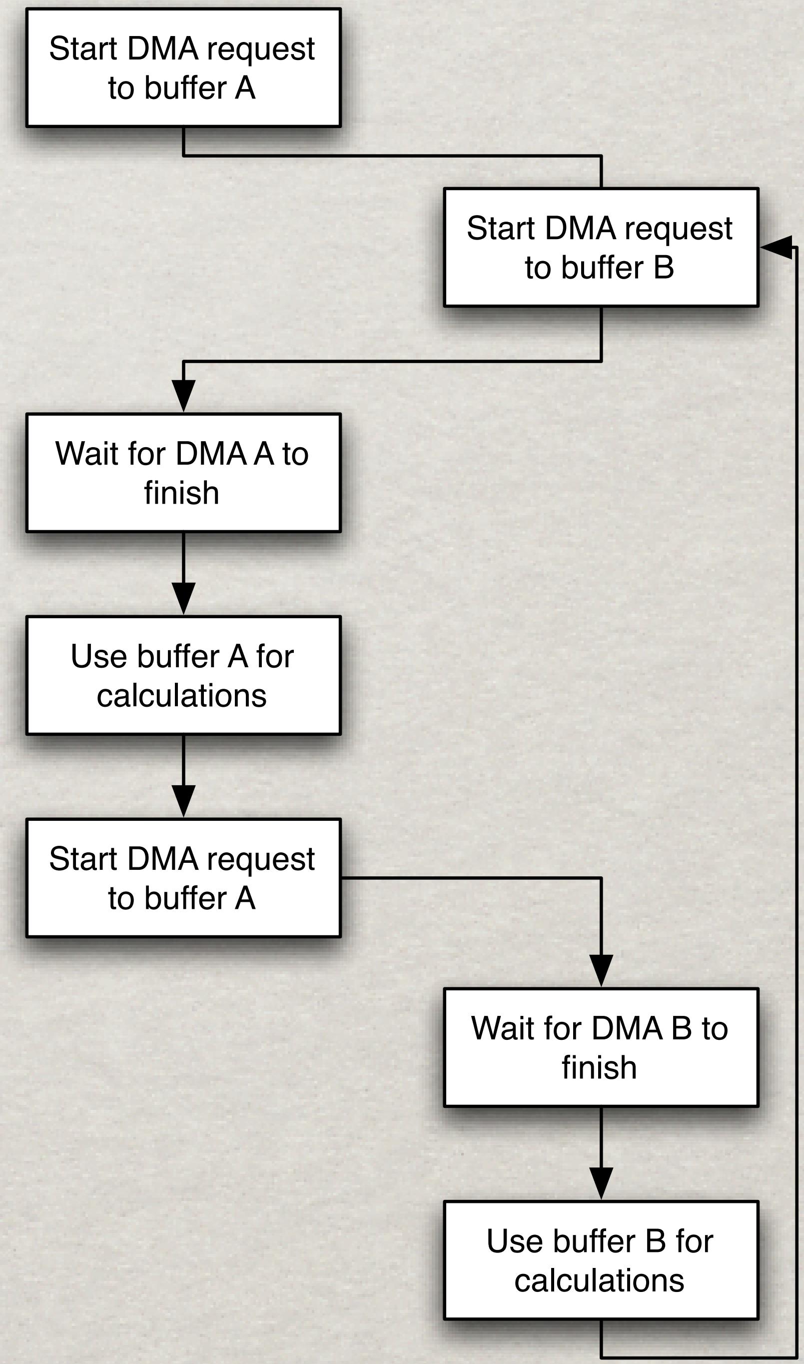
SPU Programming Models

- Code uses DMA and/or other communication primitives
- Execution and I/O don't overlap
- Harder to port existing code



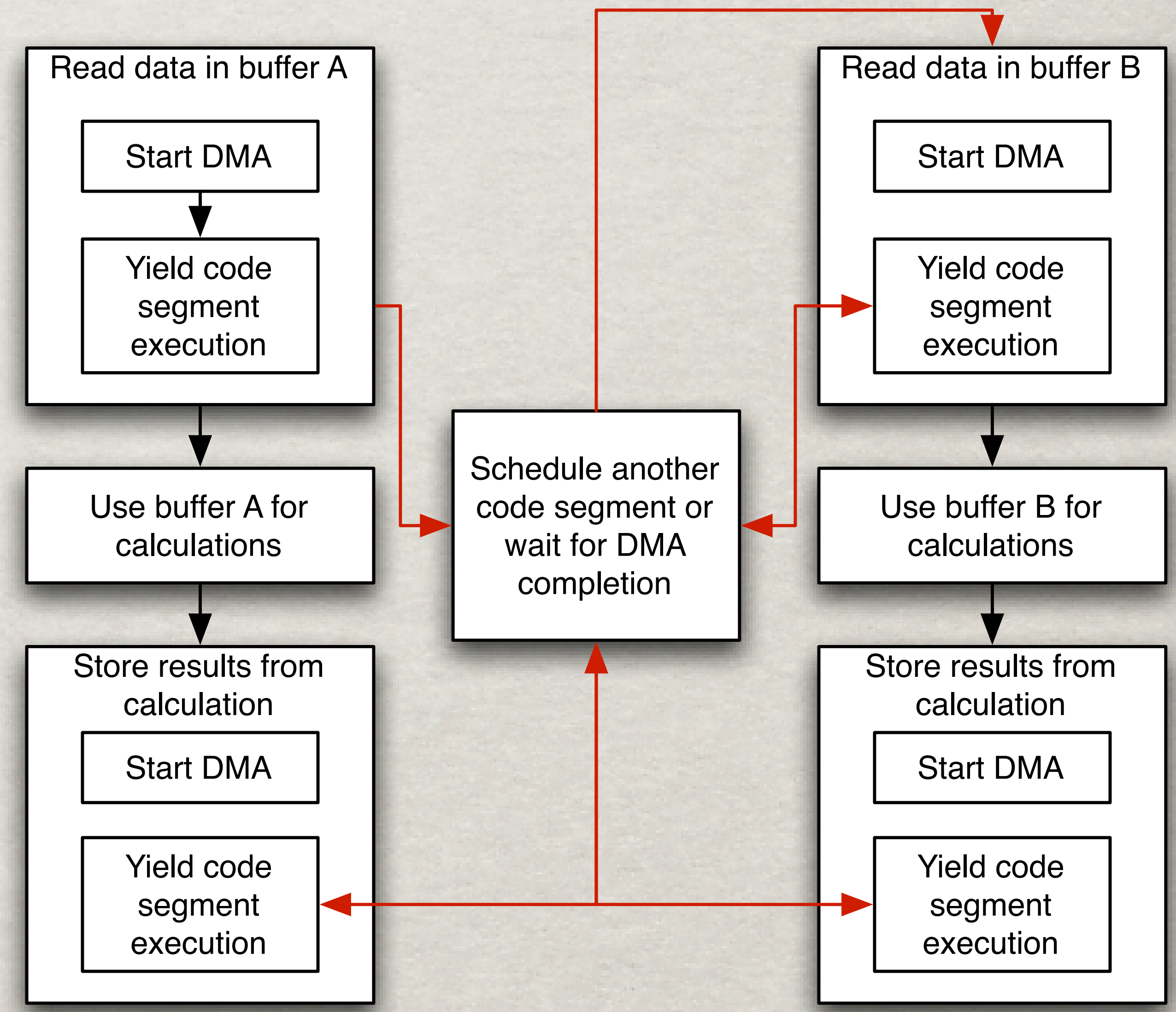
SPU Programming Models

- Overlap execution and DMA requests
- Prefetch the data to work on later
- Breaks the sequence of the algorithm
- Doesn't work if the next chunk of data depends on the result of the execution



SPU Programming Models

- Separate the work into independent code segments
- When a code segment needs to wait for DMA completion, another code segment is scheduled
- The algorithm(s) are sequential again



System Memory vs. Local Store

- System memory is thousands times bigger than the local store
- Transferring data from the system memory to the local store is done by issuing DMA requests

Disk vs. RAM

- Disk capacity is hundreds times bigger than RAM
- Transferring data from disk: issue a command and wait until the data is copied into RAM using DMA
- Very similar to System Memory vs. Local Store

DMA Requests

- Programming with DMA requests is not easy
- Application developers are not used to do DMA, it is something the OS does for them
- To avoid the DMA latency, the requests need to be scheduled long before the data is needed

DMA Requests

- Programmers use pointers and offsets to figure out what to transfer from/to the system memory
- Error-prone approach
- Very similar to the way programmers used disks and tapes long time ago before filesystems were invented

Named segments

- We define named segments in the system memory
- Operations
  - create
  - destroy
  - locate
  - load
  - store
  - dereference

```
hndA = locate("A");
hndB = locate("B");
hndC = locate("C");
load(hndA, bufA, 0, 8192);
load(hndB, bufB, 0, 8192);
matrix_mul(bufA, bufB, bufC);
store(hndC, bufC, 0, 8192);
dereference(hndA);
dereference(hndB);
dereference(hndC);
```

File Interface

- open(name, mode)
- create(name, perm, mode)
- close(fd)
- remove(name)
- pread(fd, buf, buflen, offset)
- pwrite(fd, buf, buflen, offset)

SPU Overlapped Computation

- On startup
  - The library provides implementation of **main** function
  - The library calls the user provided **maincor**

SPU Overlapped Computation

- Executable segment creation

  mkcor(func, aptr, stackptr, stacksize)

  func - function to execute

  aptr - passed to the function

  stackptr - pointer to the start of a buffer that will be used for the segment's stack

  stacksize - size of the stack buffer

SPU Overlapped computation

- Code segment switching
  - every I/O operation yields the execution
  - the code segment can explicitly yield by calling **yield** function

Code Segments vs.Coroutines

- Overlapping code segments model is widely used
- Properties
  - multiple points of entry
  - can return (yield) more than once
  - only a single segment is active at any time
- These constructs are known as coroutines

CellFS

- Access system's resources as files
  - CellFS running on the PPE creates file hierarchy that represents system's resources
  - Programs running on the SPE access the resources by using file I/O operations
- Coroutine Model
  - More than one thread of execution available on the SPE (coroutine)
  - Non-preemtible, yields execution on I/O

Performance: Memory Bandwidth

#SPU	Double Buffer	# Coroutines						
		1	2	3	4	5	6	7
1	8.19	5.68	10.63	10.71	10.75	10.81	10.78	10.79
2	15.59	11.03	20.48	20.74	20.79	20.88	20.89	20.96
3	20.05	14.75	23.20	23.26	23.27	23.28	23.29	23.28
4	20.48	17.84	23.28	23.20	23.32	21.84	23.32	23.32
5	21.63	20.71	23.26	23.27	23.28	23.28	23.28	23.29
6	21.82	21.89	23.26	23.25	23.26	23.26	23.26	23.26
7	23.05	21.05	23.24	23.23	22.05	23.23	23.24	23.23
8	22.34	21.43	23.22	23.20	23.21	23.21	23.21	23.21

Bandwidth in GB/s for performing 16777216 reads of 8192 byte blocks from a Unix file

Performance: Matrix Multiplication

256x256 elements

#SPU	Standard	CellFS
1	15.56	17.65
2	7.81	8.85
4	3.91	4.45
8	1.98	2.28

512x512 elements

#SPU	Standard	CellFS
1	124.22	141.43
2	62.16	70.75
4	31.09	35.46
8	15.57	17.88

1024x1024 elements

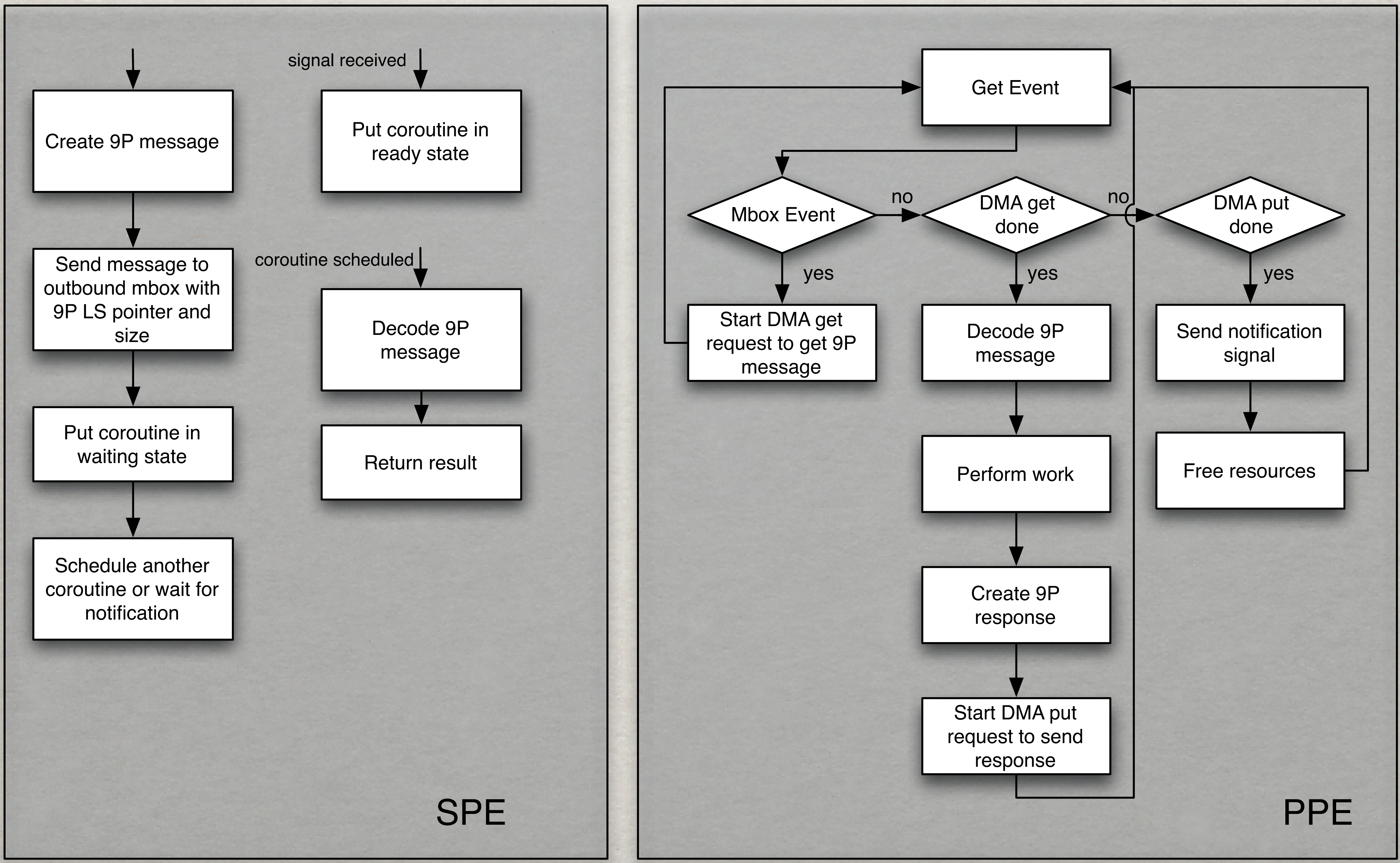
#SPU	Standard	CellFS
1	993.00	1131.98
2	496.71	566.38
4	248.40	283.36
8	124.29	142.54

Time in seconds for 10000 iterations.

Implementation: File I/O

- 9P Protocol
  - simple
  - robust
  - architecture independent
  - support for multiple transports
- 9P data types
  - fid
  - qid
    - type
    - path
    - version
- 9P Operations
  - version
  - attach
  - walk
  - clunk
  - read
  - write
  - remove
  - stat
  - wstat
  - flush
  - error

9P Message Transport



Read/Write Optimizations

- Special file type -- Dmem, qidpath point to file header
- File content is 4GB
- If SPE tries to read/write and file offset, count and buffer are aligned to 16 bytes:
  - DMA get for the file header (32 bytes)
  - DMA get/put from/to file content
- If offset+count > file size
  - If "read", the count is adjusted
  - If "write", the file size is adjusted (additional DMA put)

Example: Read 2056 byte file

`char buf[8192] __attribute__((aligned(128)));`	`<- Tversion "9P2000" 1024`
	`-> Rversion "9P2000" 1024`
	`<- Tattach fid 0 afid -1 aname "" uname ""`
	`-> Rattach (...)`
`int maincor()`	
`{`	
`        int fd, n;`	
`        fd = open("#u/tmp/test", Oread);`	`<- Twalk fid 0 newfid 1 "#u" "tmp" "test"`
	`-> Rwalk (...) (...) (...)`
	`<- Topen fid 1 mode 1`
	`-> Ropen (...)`
`        n = read(fd, buf, sizeof(buf));`	`<- Tread fid 1 offset 0 count 1000`
	`-> Rread count 1000 data ...`
	`<- Tread fid 1 offset 1000 count 1000`
	`-> Rread count 1000 data ...`
	`<- Tread fid 1 offset 2000 count 1000`
	`-> Rread count 56 data ...`
`        close(fd);`	`<- Tclunk fid 1`
	`-> Rclunk`
`}`	

Example: Optimized Reading

`char buf[8192] __attribute__((aligned(128)));`	`<- Tversion "9P2000" 1024`
	`-> Rversion "9P2000" 1024`
	`<- Tattach fid 0 afid -1 aname "" uname ""`
	`-> Rattach (...)`
`int maincor()`	
`{`	
`        int fd, n;`	
`        fd = open("#U/tmp/test", Oread);`	`<- Twalk fid 0 newfid 1 "#u" "tmp" "test"`
	`-> Rwalk (...) (...) (Dmem fhptr)`
	`<- Topen fid 1 mode 1`
	`-> Ropen (Dmem fhptr)`
	DMA get `ptr fhptr count 32`
	DMA wait
`        n = read(fd, buf, sizeof(buf));`	*DMA get* `ptr fhptr count 32`
	DMA get `ptr fptr count 8192`
	DMA wait
	adjust the read count to the size from the
`        close(fd);`	`<- Tclunk fid 1`
	`-> Rclunk`
`}`	

CellFS: Resources Represented As Files

- Filenames with resource type (#*letter*)
- Memory files (#r)
    - multi-level directories
    - regular files -- memory areas
- Unix Files (#u, #U and #R)
    - #U and #R unix files are **mmap**-ed to memory
    - #u unix files are accessed via **pread** and **pwrite**
- Pipes (#p)
    - create files with arbitrary names
    - similar to Unix FIFO files
- Log (#l)
- Execution (#x) *(not implemented yet)*

Conclusions

- Advantages
  - Architecture independent interface
  - Familiar I/O functions
  - Coroutine model is simple, non-preemtive, and no locking necessary
- Disadvantages
  - The library implementation takes some of the limited LS memory
  - Coroutine model requires some more memory for stack

Future Work

- Integration with XCPU
- Direct pipes - point-to-point between SPUs
- Implementation of the I/O and coroutine library for the PPE