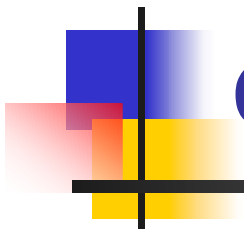
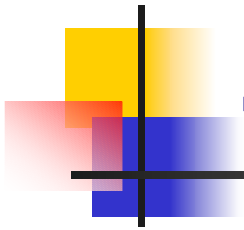


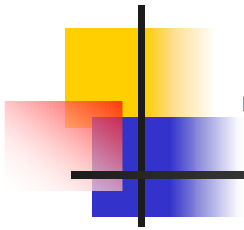
OpenMP on the FDSM software distributed shared memory

Hiroya Matsuba
Yutaka Ishikawa



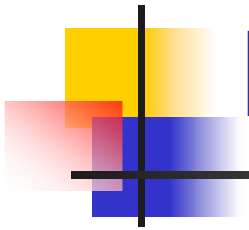
Software DSM

- OpenMP programs usually run on the shared memory computers
- OpenMP programs work on the distributed memory parallel computers
  - Distributed shared memory may be used



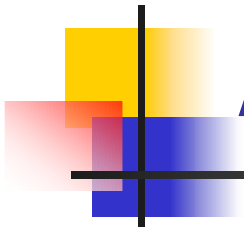
Software DSM

- DSM has been studied for a long time
- Performance problem
  - Many DSM systems make “twin & diff” to know the communication pattern
    - High overhead
  - False sharing



FDSM

- New software DSM system
- Goal
  - To reduce the overhead of the software DSM system



Approach

- OpenMP is often used for a numerical analysis
  - Loops are used
  - The work in the loop is shared among the processors
    - Communication is required at every iteration

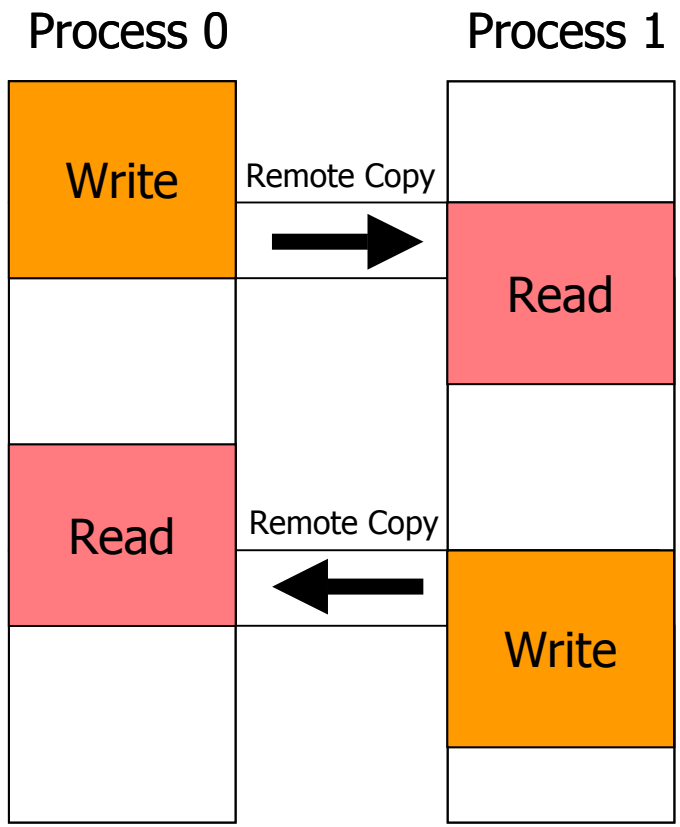
Approach

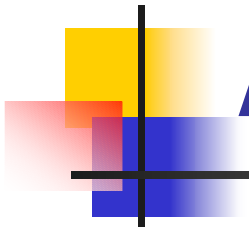
- Important feature of the numerical analysis
  - The access pattern to the shared memory is fixed in every iteration

Communication pattern is also fixed

Approach

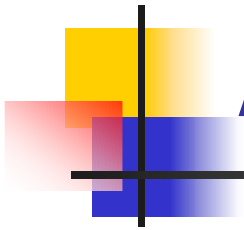
- First iteration
  - FDSM obtains the access pattern
- Later
  - FDSM communicates using the pattern
  - The overhead to know the communication pattern is reduced





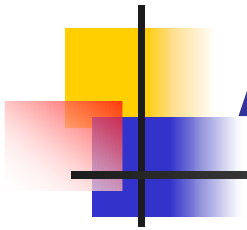
Approach

- Synchronization
  - The writers can send the data directly to the readers
  - That is all



Access pattern analysis

- The main issue about FDSM
  - How to obtain the access pattern ?
- Granularity
  - Two processors may write to one page
  - FDSM needs to get the access pattern with the single byte granularity

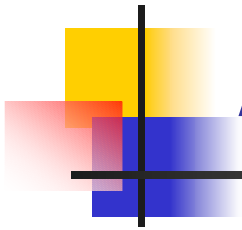


Access pattern analysis

The page protection mechanism is used

1. FDSM forbids all the access to the shared memory
2. When the memory is accessed, page fault occurs
3. The accessed address is recorded in the exception handler in the kernel

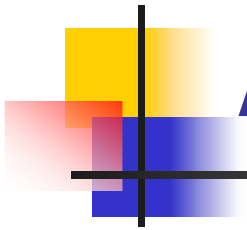
Problem in returning to the user program



Access pattern analysis

Problem 1

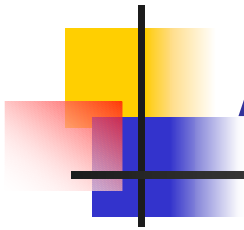
- Exception handler will remove the cause of the fault
  - The page fault handler will remove the access protection



Access pattern analysis

Problem 1

- If the exception handler enables the access to the page
  - The next access to this page will not cause the page fault
  - FDSM cannot obtain the access pattern any more



Access pattern analysis

Problem 2

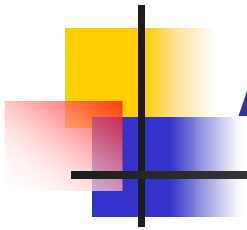
- If the exception handler does not enable the access to the page
  - The same instruction will cause the page fault again
  - The execution cannot be continued

Access pattern analysis

Solution

- The exception handler emulates the fault instruction and increments the PC
  - The memory access is completed
  - The fault instruction will not be executed again

  - Accessed address is recorded
  - Execution continues



Access pattern analysis

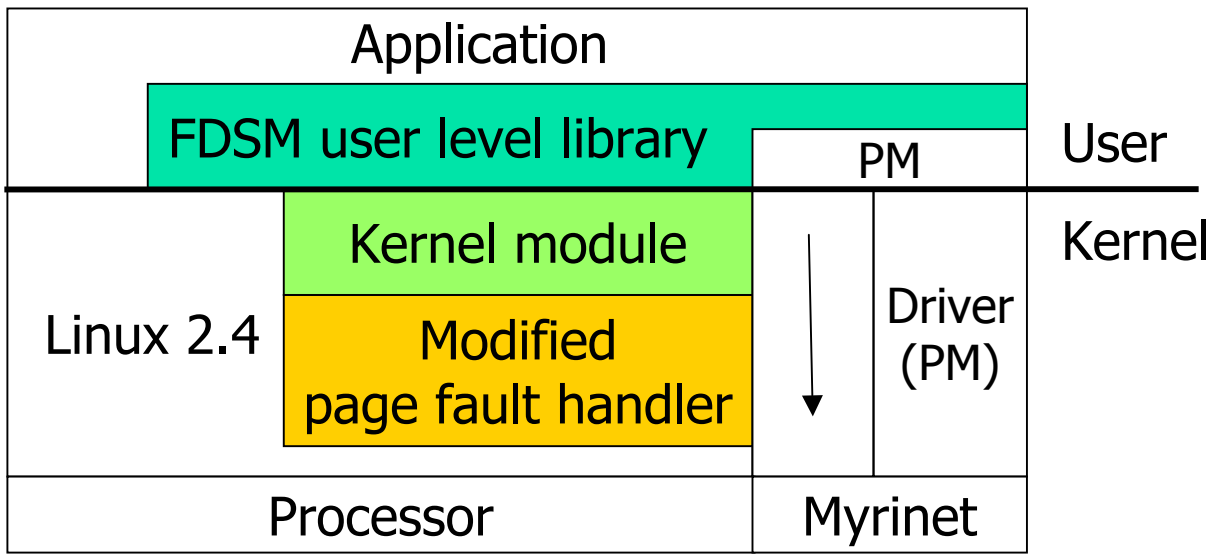
Solution

- The exception handler does not enable the access to the page
  - The next access will cause the page fault

  - The access to this page can be detected again

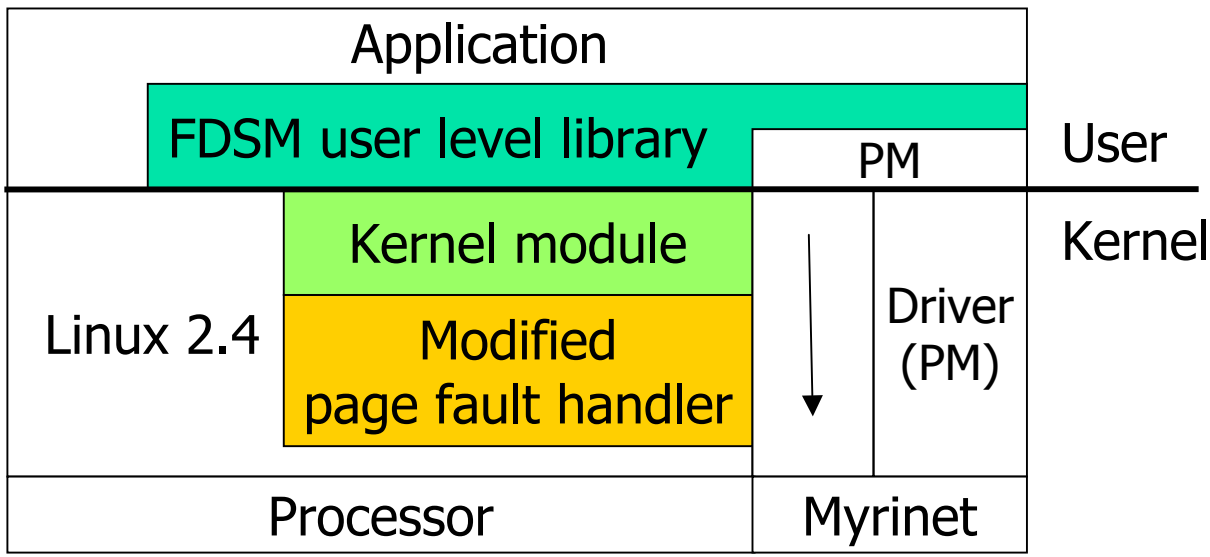
Software architecture

- The page fault handler is modified
- Most part of FDSM is implemented as the kernel module
  - Including the instruction emulator



Software architecture

- The user level library provides the API's and performs the communication
- PM communication library is used
  - PM provides user level communication



Omni OpenMP Compiler Modification

OpenMP

```
#pragma omp for
for (i = 0; i < N; i++) {
    a[i] = b[i] * b[i];
}
```

Run-time system obtains the access pattern

↓

Compiler is simple

Generated code

```
{
    int t1 = 0;
    int t2 = N;
    fdsm_adjust_loop(&t1, &t2)
    fdsm_before_loop();
    for (i = t1; i < t2; i ++) {
       a[i] = b[i] * b[i];
    }
    fdsm_after_loop();
}
```

Omni OpenMP Compiler Modification

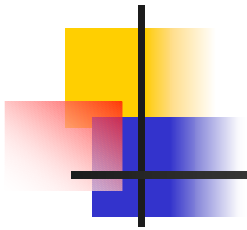
This function modifies the first and the last value of the loop counter

This function tells the run-time system to begin the access pattern analysis

This function performs the barrier

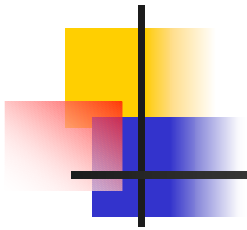
Generated code

```
{
    int t1 = 0;
    int t2 = N;
    fdsm_adjust_loop(&t1, &t2)
    fdsm_before_loop();
    for (i = t1; i < t2; i ++) {
       a[i] = b[i] * b[i];
    }
    fdsm_after_loop();
}
```



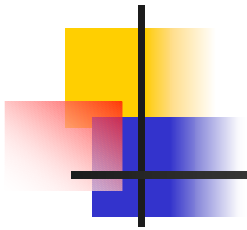
Evaluation

- Hardware environment
  - 16 node cluster
  - PentiumIII 500MHz on each node
  - 512MB on each node
  - Myrinet 1.25Gbps
- Software environment
  - Linux 2.4 (modified)
  - SCore cluster system software



Evaluation

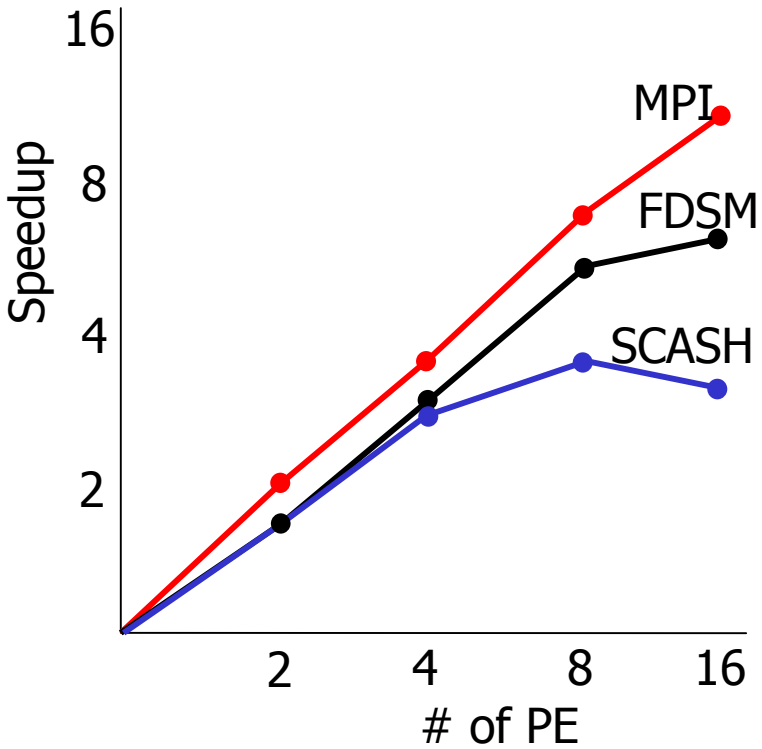
- Benchmark application
  - CG
    - NAS parallel benchmark OpenMP version
  - Class A
  - Hand compiled
    - We have not implemented the compiler yet



Evaluation

- Comparison
  - Omni/SCASH
    - Software DSM system
    - Implemented at the user level using twin & diff
    - Runs on SCore
  - MPI
    - MPI version of CG

Result



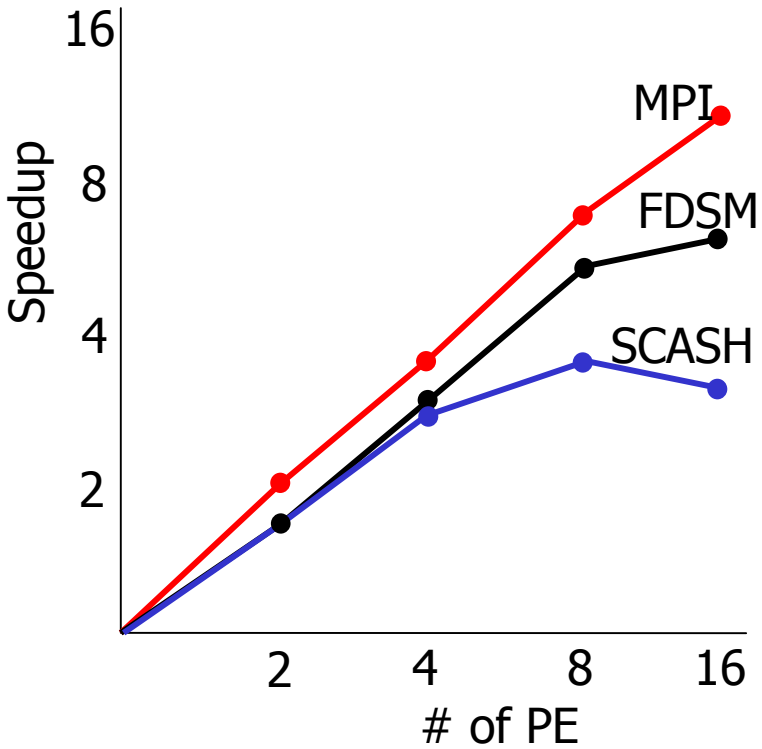
Speedup

CG result (Mops)

	1	2	4	8	16
FDSM	30.1	51.4	88.3	134	160
MPI	30.1	58.9	107	200	334
SCASH	30.1	51.4	80.7	95.5	84.8

- FDSM scales up to 8 processors
  - Close to MPI
- FDSM is faster than SCASH

Result

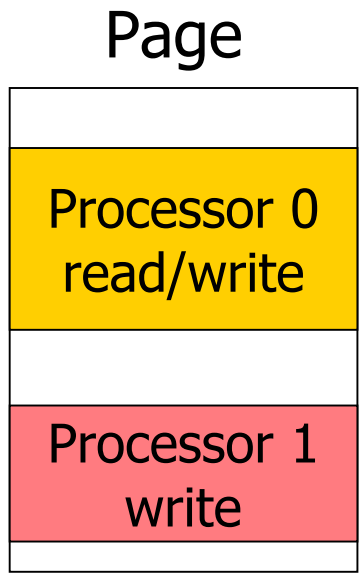


Speedup

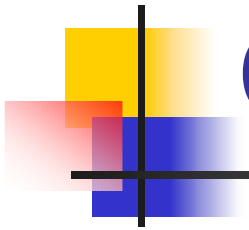
- FDSM does not perform well on 16 nodes

Discussion

- Current implementation
  - The single byte granularity is used only for the write access
  - The pattern of the read access is obtained with the page size granularity
    - Unnecessary communication
    - It may be the reason why FDSM is slow on 16 nodes



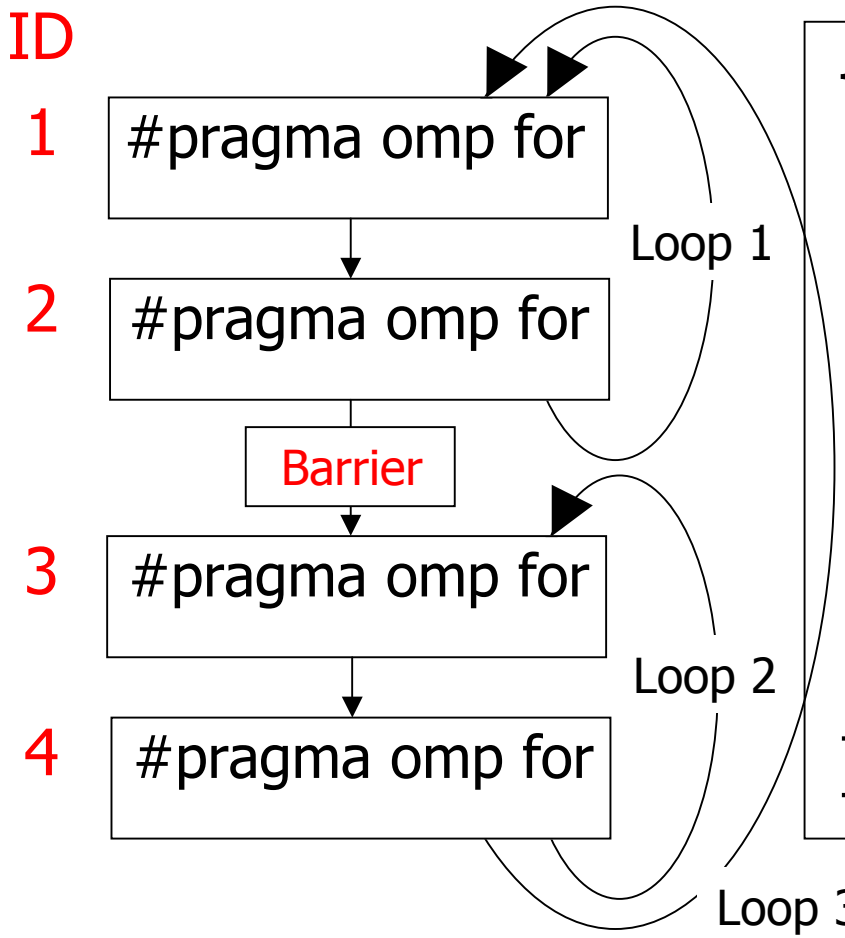
The communication from 1 to 0 occurs
But it is not necessary



Conclusion

- FDSM succeeds to speed up the execution of an OpenMP application on a cluster
  - By obtaining the access pattern of the application
- The performance is close to MPI up to 8 nodes
- FDSM does not require the analysis by a compiler
- FDSM can work as normal DSM
  - All the OpenMP directives can be supported

Nested loop detection (1/2)




```
{
    int t1 = 0;
    int t2 = N;
    adjust_loop(&t1, &t2);
    FDSM_before_loop(1);
    for (i = t1; i < t2; i ++) {
       a[i] = b[i] * b[i];
    }
    barrier();
}
```

Nested loop detection (2/2)

- Example of the nested loop detection
  - The argument of “fdsm_before_loop”
    - 1, 2, 1, 2, 3, 4, 3, 4, 1, 2, ……

1, 2, 1, 2, 3, 4, 3, 4, 1, 2,