Programming Guide (Tuning)

Mar. 2023

v2.2

FUJITSU LIMITED

Preface

- This document describes how to tune applications for the A64FX processor.
- Note
  - Because of the different compilers in Fortran, C/C++, Trad Mode and Clang Mode, there may be different tuning methods or no corresponding tuning method.
  - In the case of similar tuning methods in Trad Mode and Clang Mode, the tuning in Clang Mode is omitted.
- In addition to this document, also see the following:
  - *Fortran User's Guide*
  - *C/C++ User's Guide*
  - *Programming Guide - Processors*
  - *Programming Guide - Integrated Programming Guide*
  - *Programming Guide – Fortran*
- This document was written with reference to the following documentation:
  - *A64FX Logic Specifications*
  - *A64FX® Microarchitecture Manual*
  - *ARM® Architecture Reference Manual (ARMv8, ARMv8.1, ARMv8.2, ARMv8.3)*
  - *ARM® Architecture Reference Manual Supplement - The Scalable Vector Extension*
- Trademarks
  - Linux® is a trademark or registered trademark of Linus Torvalds in the United States and other countries.
  - Red Hat is a trademark or registered trademark of Red Hat Inc. in the United States and other countries.
  - ARM is a trademark or registered trademark of ARM Ltd. in the United States and other countries.
  - Proper names such as the product name mentioned are trademark or registered trademark of each company.
  - Trademark symbols such as ® and ™ may be omitted from system names and product names in this document.

Scope of This Document

FUJITSU

The tuning of applications has the following aspects and corresponding points. This document describes CPU tuning and thread parallelization tuning.

Scope of this document

	1-Core Tuning	Thread Parallelization Tuning	Process Parallelization Tuning	Ultra-High Parallelization Tuning
Tuning points	✓ Reduce I/O ✓ Reduce operational amount ✓ Facilitate SIMDization ✓ Reduce memory access ✓ Improve cache usage	✓ Increase parallelization ratio ✓ Increase parallelization granularity ✓ Reduce cost of synchronization between threads ✓ Equalize load balance	✓ Increase parallelization ratio ✓ Increase parallelization granularity ✓ Reduce cost of communication between processes ✓ Equalize load balance	✓ Increase parallelization ratio ✓ Increase parallelization granularity ✓ Reduce cost of communication between processes ✓ Equalize load balance ✓ Reduce global communication cost

Contents (1/2)

Contents (2/2)

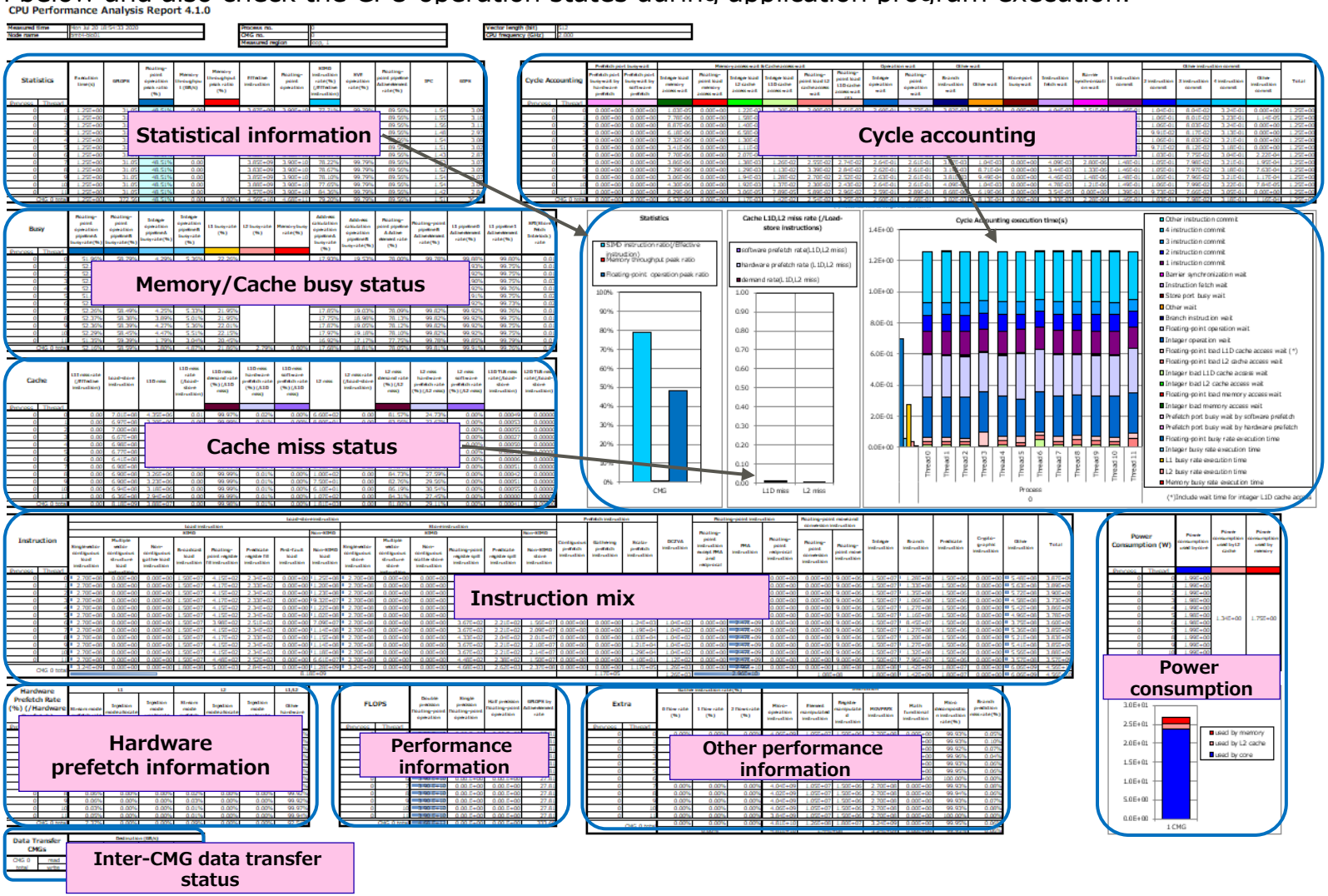
FUJITSU

Investigating Bottlenecks

CPU Performance Analysis Report: Overview

We recommend using the CPU performance analysis report to extract performance bottlenecks.

With the CPU performance analysis report, you can measure a rich variety of PA (Performance Analysis) events as shown below and also check the CPU operation states during application program execution.

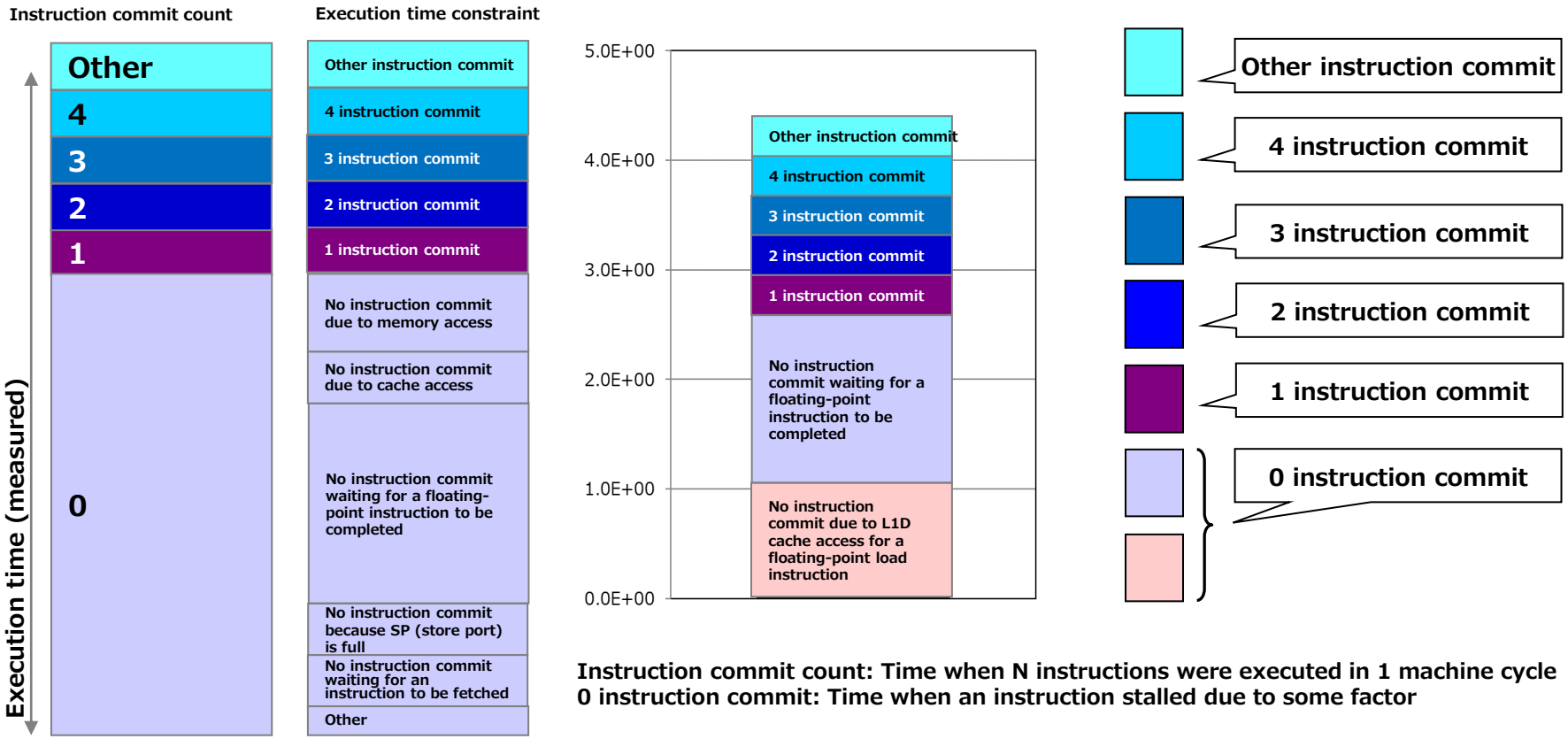


CPU Performance Analysis Report: What is Cycle Accounting?

Cycle accounting is a means to analyze performance bottleneck factors.

The CPU performance analysis report displays cycle accounting information at the upper right.

Cycle accounting divides the total time (number of CPU cycles) taken to execute an application program into CPU operation states and shows this information graphically. Since CPU bottlenecks can be identified from the resulting graphs, you can finely analyze performance and fine-tune the program.



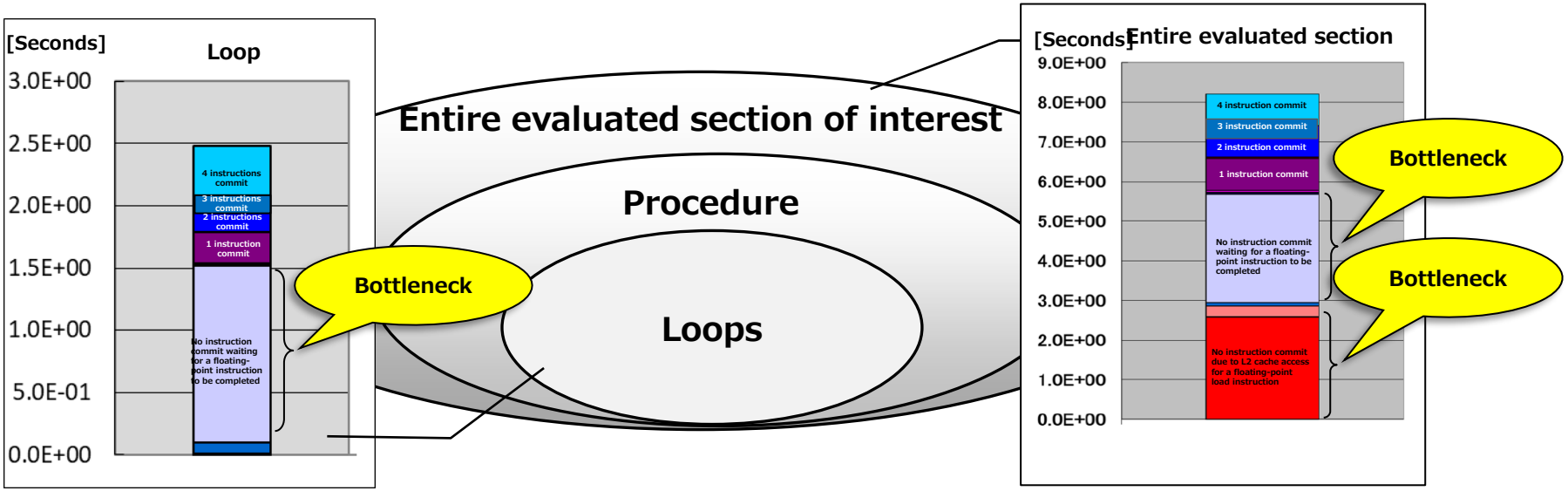
Instruction commit count: Time when N instructions were executed in 1 machine cycle
0 instruction commit: Time when an instruction stalled due to some factor

Bottleneck Extraction Using Cycle Accounting

● Identifying bottlenecks

Identifying bottlenecks is fundamental for tuning.

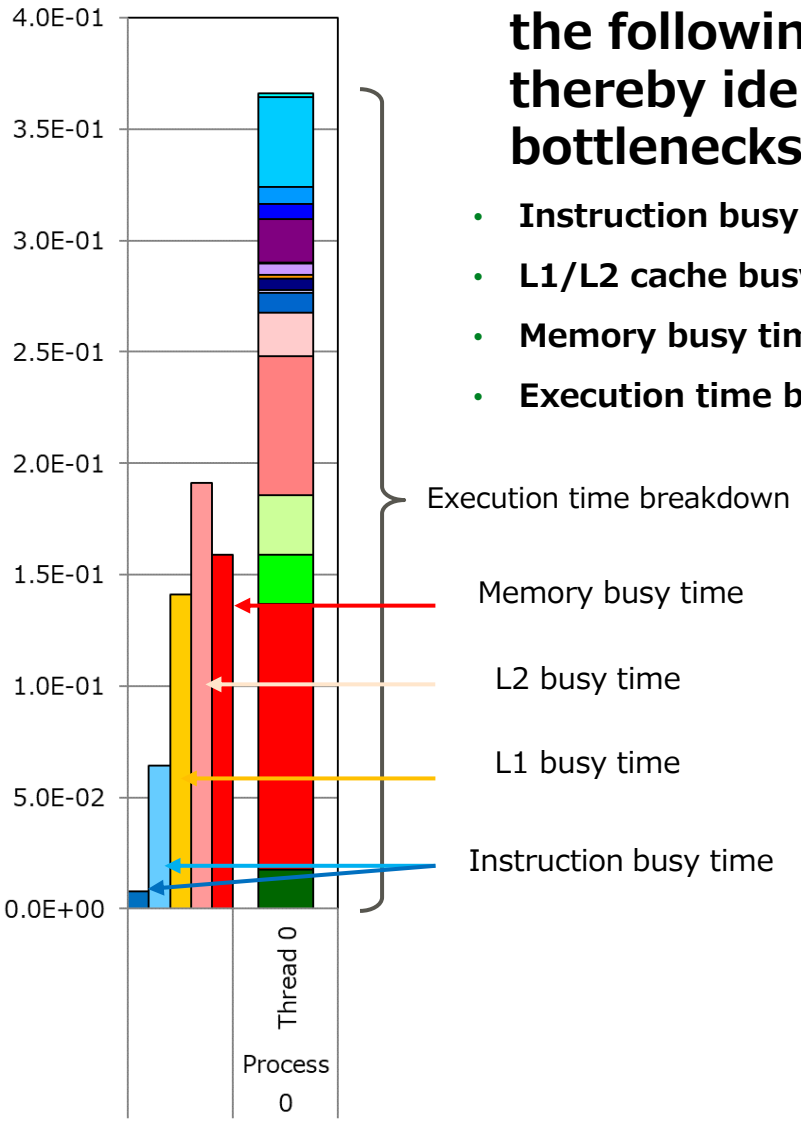
You can determine bottlenecks in the evaluated section after cycle accounting.



● Utilization for tuning

- What measures must be taken to improve bottlenecks?
- How much can they be improved?
To answer these questions,
the people making the analysis should break down the section into loops.

(Bandwidth) Bottleneck Extraction Using Various Busy Times



- **The cycle accounting graph on the left shows the following busy time information. You can thereby identify various bandwidth bottlenecks.**
  - **Instruction busy time**
  - **L1/L2 cache busy time**
  - **Memory busy time**
  - **Execution time breakdown of each thread**

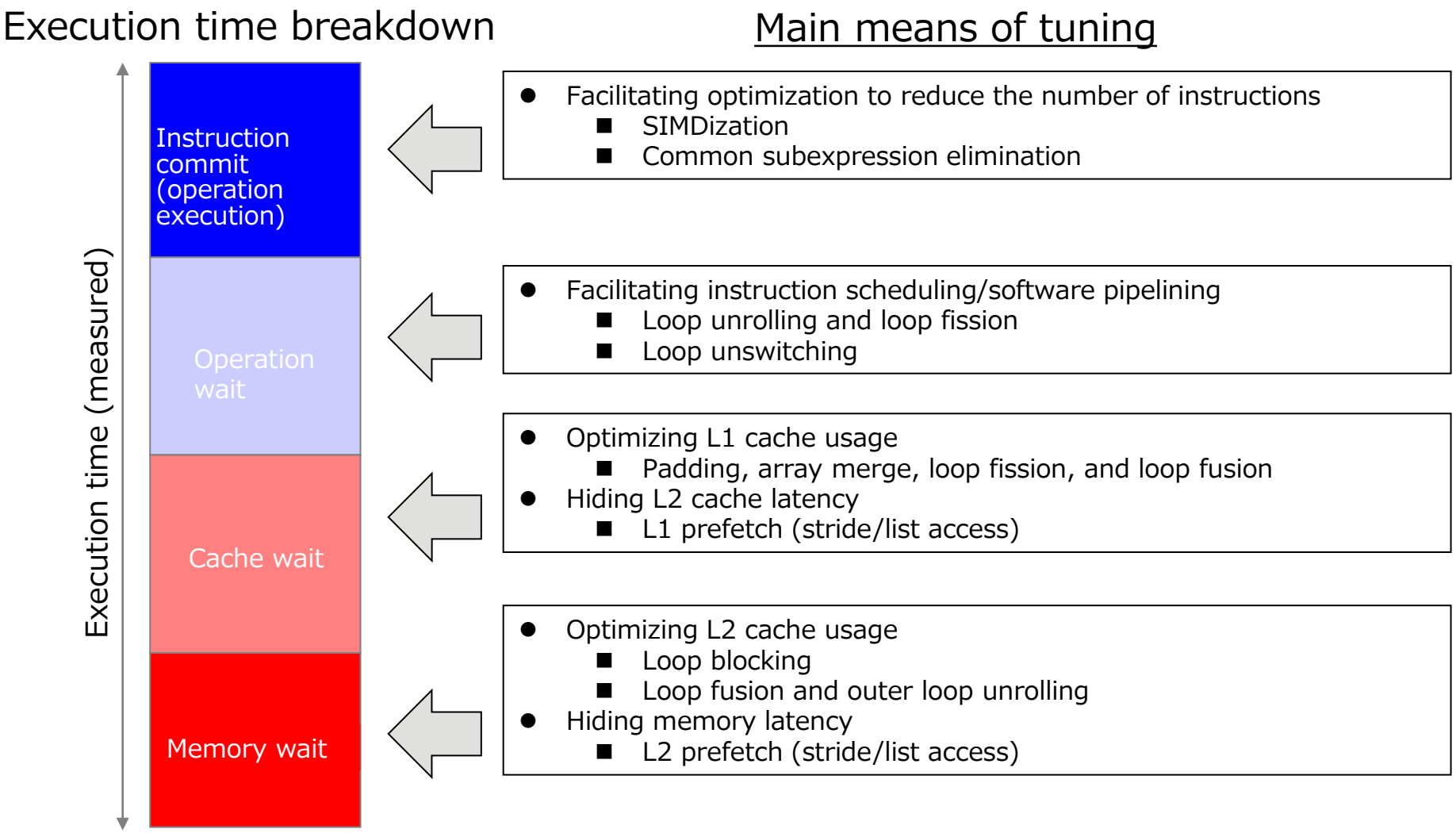
- **Memory busy time**
  Occurs when the amount of data to transfer to memory is large.
- **L1/L2 cache busy time**
  Occurs when the amount of data to transfer to the L1/L2 cache is large.
- **Instruction busy time**
  Occurs when the operational amount is large.

Tuning Methods Using Cycle Accounting

FUJITSU

- **Select a means of tuning, based on cycle accounting results.**

  The following figure shows the main means of tuning.
  For details, see the tuning map.



Tuning Map: Classification and States

Bottleneck Classification	High Cost Seen on PA Graph		High Cost Seen in PA Information	Situation
Memory bottleneck	No instruction commit due to memory access for a floating-point load instruction		-	Memory latency is a bottleneck.
	No instruction commit due to memory access for an integer load instruction		-	
	No instruction commit because SP (store port) is full		-	The cost of store instructions is a bottleneck.
	No instruction commit because memory cache is busy		-	Memory throughput is a bottleneck.
	-		High memory busy rate	
	-		High L2 miss rate High L2 miss dm rate	Memory latency is a bottleneck.
L2 cache bottleneck	No instruction commit due to L2 cache access for a floating-point load instruction		-	L2 cache latency is a bottleneck.
	No instruction commit due to L2 cache access for an integer load instruction		-	
	-		High L2 busy rate	L2 cache throughput is a bottleneck.
	-		High L1D miss rate High L1D miss dm rate	L2 cache latency is a bottleneck.
L1 cache bottleneck	No instruction commit due to L1D cache access for a floating-point load instruction		-	L1 cache latency is a bottleneck.
	No instruction commit due to L1D cache access for an integer load instruction		-	
	-		High L1 busy rate	L1 cache throughput is a bottleneck.
Scheduling bottleneck	No instruction commit waiting for a floating-point instruction to be completed		-	Operation instruction latency is a bottleneck.
	No instruction commit waiting for an integer instruction to be completed		-	
	No instruction commit waiting for a branch instruction to be completed		-	A branch instruction is a bottleneck.
Parallelization bottleneck	Synchronous waiting time between threads		-	A part with no thread parallelization is a bottleneck.
Load imbalance bottleneck	Synchronous waiting time between threads		Large max-min difference in instruction balance	Load balance between threads is a bottleneck.
TLB bottleneck	-		High mDTLB miss rate	TLB misses or thrashing is a bottleneck.
	-		High uDTLB miss rate	TLB misses are a bottleneck.
Instruction fetch	No instruction commit waiting for an instruction to be fetched		-	Instruction cache misses or thrashing is a bottleneck.
Bottleneck due to number of instructions	Instruction commit	Other instruction commit	-	The number of instructions is a bottleneck.
		4 instruction commit		
		3 instruction commit		
		2 instruction commit		
		1 instruction commit		
Other	No instruction commit for other reasons		-	PA data may not have been properly collected.

Tuning Map 1/4

FUJITSU

Throughput bottlenecks

High Cost Seen on PA Graph	High Cost Seen in PA Information	Situation	Proposed Tuning
No instruction commit because memory cache is busy	-	Memory throughput is a bottleneck.	Improve the data access wait time. - Array dimension shift - Loop blocking - Strip Mining - High-speed store (ZFILL)
-	High memory busy rate	Memory throughput is a bottleneck.	Improve the data access wait time. - Array dimension shift - Loop blocking - Strip Mining - High-speed store (ZFILL)
-	High L2 miss rate High L2 miss dm rate	Memory latency is a bottleneck.	Improve the data access wait time. - Array dimension shift - Loop blocking - Strip Mining - Sector Cache - Prefetch-related improvement - Improved Thrashing
	High L2 busy rate	L2 cache throughput is a bottleneck.	Improve the data access wait time. - Array dimension shift - Loop blocking - Strip Mining
	High L1 busy rate	L1 cache throughput is a bottleneck.	Improve the data access wait time. - Algorithm review

FUJITSU

Latency bottlenecks

High Cost Seen on PA Graph	High Cost Seen in PA Information	Situation	Proposed Tuning
No instruction commit due to memory access for a floating-point load instruction		Memory latency is a bottleneck.	Improve the data access wait time. - Array dimension shift - Prefetch-related improvement - Loop blocking
No instruction commit due to memory access for an integer load instruction			
No instruction commit due to L2 cache access for a floating-point load instruction		L2 cache latency is a bottleneck.	Improve the data access wait time. - Array dimension shift - Unroll-and-Jam - Prefetch-related improvement
No instruction commit due to L2 cache access for an integer load instruction			
-	High 1D miss rate High L1D miss dm rate		Improve the data access wait time. - Array dimension shift - Improved Thrashing
No instruction commit due to L1D cache access for a floating-point load instruction		L1 cache latency is a bottleneck.	Improve instruction scheduling. Improve microarchitecture-dependent bottlenecks.
No instruction commit due to L1D cache access for an integer load instruction			
No instruction commit waiting for a floating-point instruction to be completed		Operation instruction latency is a bottleneck.	Improve instruction scheduling.
No instruction commit waiting for an integer instruction to be completed			

Tuning Map 3/4

Bottleneck due to the number of instructions

High Cost Seen on PA Graph		High Cost Seen in PA Information	Situation	Proposed Tuning
Instruction commit	Other instruction commit 4 instruction commit 3 instruction commit 2 instruction commit 1 instruction commit		The number of instructions is a bottleneck.	Improve bottlenecks due to the number of instructions. - Facilitation of SIMDization - Facilitation of software pipelining - Prefetch-related improvement - Inline expansion

TLB bottlenecks

High Cost Seen on PA Graph	High Cost Seen in PA Information	Situation	Proposed Tuning
-	High mDTLB miss rate	TLB misses or thrashing is a bottleneck.	Improve TLB bottlenecks. - Thrashing elimination - Change of the area used - Optimization with the large page option
-	High uDTLB miss rate	TLB misses are a bottleneck.	Improve TLB bottlenecks. - Expansion of the page size

Tuning Map 4/4

FUJITSU

Other

High Cost Seen on PA Graph	High Cost Seen in PA Information	Situation	Proposed Tuning
No instruction commit because SP (store port) is full		The cost of store instructions is a bottleneck.	Improve the data access wait time. - Array dimension shift - Prefetch-related improvement - High-speed store (ZFILL)
No instruction commit waiting for a branch instruction to be completed		A branch instruction is a bottleneck.	Improve instruction scheduling. - Elimination of IF statements - Masked SIMD
Synchronous waiting time between threads		A part with no thread parallelization is a bottleneck.	Improve thread parallelization.
	Large max-min difference in instruction balance	Load balance between threads is a bottleneck.	Improve the efficiency of parallel thread execution.
No instruction commit waiting for an instruction to be fetched		Instruction cache misses or thrashing is a bottleneck.	Improve instruction fetch. - Loop body reduction - Algorithm review - Thrashing elimination

FUJITSU

1-Core Tuning

- Data Access Wait Time (Increased Data Locality)
- Data Access Wait Time (Hidden Latency)
- Data Access Wait Time (Reduced Access Amount)
- Data Access Wait Time (Improved Thrashing)
- Operation Wait (Facilitation of SIMDization)
- Operation Wait (Hidden Latency)

What is Data Locality?

FUJITSU

Data locality refers to the degree that data reference and access are concentrated in a narrow range.
Data in cache memory is not effectively used when data locality is low, resulting in a high memory access load.
By improving data locality through source tuning, you can improve the data access wait time to reduce the memory access load.

The following means are effective at increasing data locality:

- **Strip Mining**
- **Loop Blocking**
- **Sector Cache**
- **Loop Interchange**
- **Loop Fusion**
- **Array Merge (Indirect Access)**
- **Array Dimension Shift**
- **Unroll-and-Jam**

FUJITSU

Strip Mining

- What is Strip Mining?
- Strip Mining (Before Improvement)
- Effect of Strip Mining (Source Tuning)

What is Strip Mining?

Strip mining is a means to increase cache efficiency through fragmentation of a loop into smaller segments or strips.

Example: Source Before Improvement

```
integer n,m
real*8 a(n,m),b(n,m),c(n,m),d(n,m),e(n,m)

do j=1,m
  do i=1,n
    a(i,j)=b(i,j)+c(i,j)
  enddo
  do i=1,n-100
    d(i,j)=a(i,j)+e(i,j)
  enddo
enddo
```

n=100*1000/8
m=20

Loop 1

Loop 2

Since the number of iterations is large, the data in Array a in the cache is overwritten.

Array a causes a cache miss.

Example: Source After Improvement

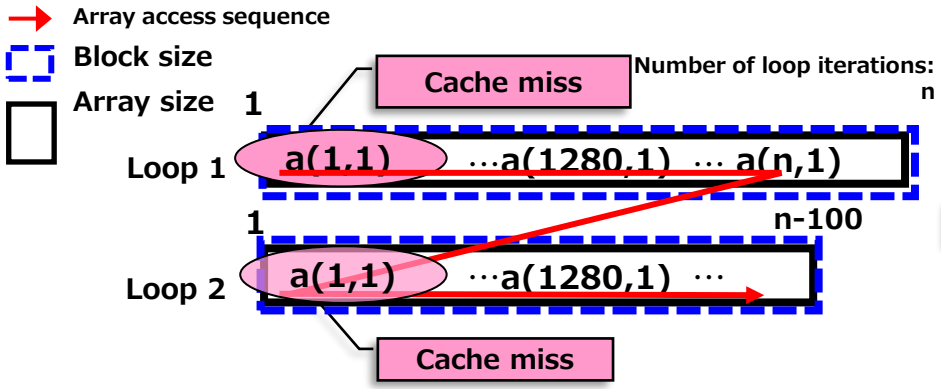
```
integer n,m
real*8 a(n,m),b(n,m),c(n,m),d(n,m),e(n,m)
blki=10*1024/8

do j=1,m
  do ii=1,n,blki
    do i=ii,min(ii+blki-1,n)
      a(i,j)=b(i,j)+c(i,j)
    enddo
    do i=ii,min(ii+blki-1,n-100)
      d(i,j)=a(i,j)+e(i,j)
    enddo
  enddo
enddo
```

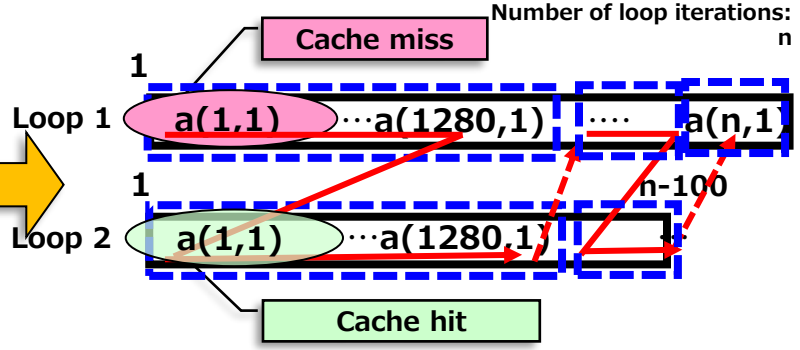
Block size

Loop 1

Loop 2

Data in Array a remains in the cache, causing a cache hit.





Fortran Strip Mining (Before Improvement)

Array data cannot be fully cached and thus cannot be reused in Loop 2 because the number of iterations of Loop 1 is large. Consequently, the "No instruction commit because memory cache is busy" event occurs many times.

Source Before Improvement

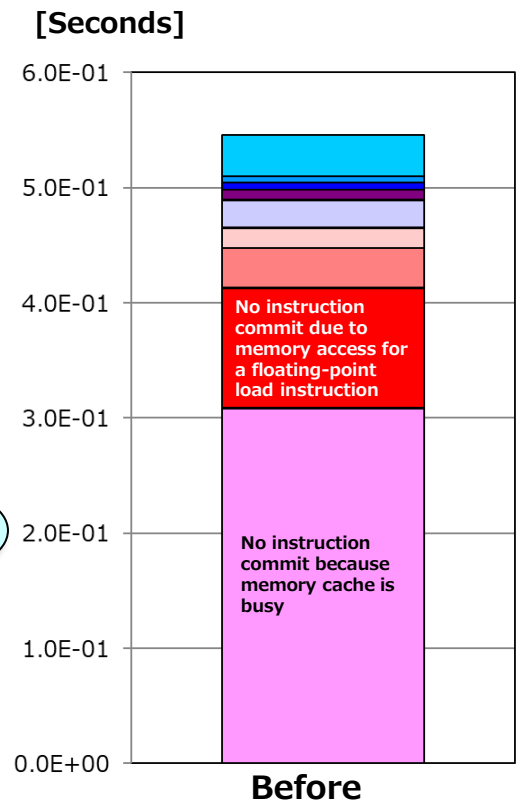
```
32              !$omp parallel do reduction(+:s1)
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      d, c, b, a, e
                <<< Loop-information  End >>>
33    1  p          do j=1,m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, c, a, d
                <<< Loop-information  End >>>
34    2  p  8v        do i=1,n
35    2  p  8v          s1 = s1 + a(i,j) * (s3 * b(i,j) + c(i,j) * (s2 + s3 * d(i,j) ))
36    2  p  8v        enddo
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING
                <<<    PREFETCH(HARD) Expec
                <<<      b, a, d, c, e
                <<< Loop-information  End >>>
37    2  p  2v        do i=1,n-100
38    2  p  2v          e(i,j) = s2 * (a(i,j) + b(i,j) * (s3 + c(i,j) * d(i,j)))
39    2  p  2v        enddo
40    1  p          enddo
```

Number of loop iterations: 375,000
Total array size: 12 MB
Since number of iterations is large, array data cannot be fully cached

Loop 1

Array access causes cache miss

Loop 2



Data cannot be reused, so the L1D and L2 miss rates are around 0.20, which is the theoretical value of stream access.

Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	2.08E+09	4.23E+08	0.20	7.61%	92.40%	-0.01%	4.23E+08	0.20	5.17%	95.41%	0.00%

Fortran Effect of Strip Mining (Source Tuning)

FUJITSU

Strip mining increases cache efficiency, resulting in improvement of the "No instruction commit because memory cache is busy" event.

Source After Improvement

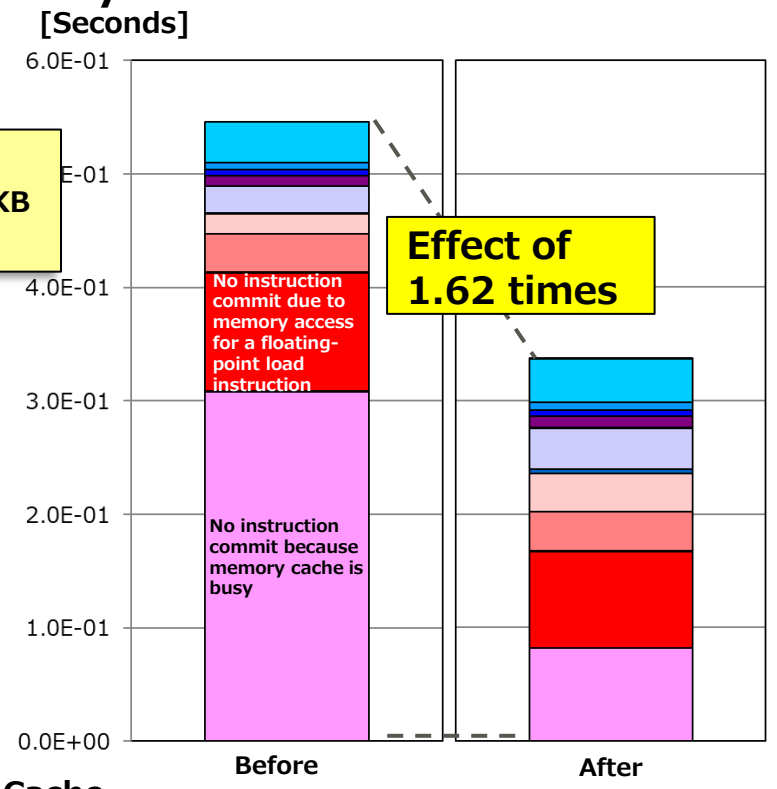
```
32                blki=4*1024/8
33
34              !$omp parallel do reduction(+:s1)
35    1  p          do j=1,m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      d, c, b, a, e
                <<< Loop-information  End >>>
36    2  p            do ii=1,n,blki
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, c, a, d
                <<< Loop-information  End >>>
37    3  p  8v           do i=ii,min(ii+blki-1,n)
38    3  p  8v             s1 = s1 + a(i,j) * (s3 * b(i,j) + &
39    3                              c(i,j) * (s2 + s3 * d(i,j) ))
40    3  p  8v           enddo
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(
                <<<    PREFETCH(HARD) Expec
                <<<      b, a, d, c, e
                <<< Loop-information  End >>>
41    3  p  2v           do i=ii,min(ii+blki-1,n-100)
42    3  p  2v             e(i,j) = s2 * (a(i,j) + &
43    3                              b(i,j) * (s3 + c(i,j) * d(i,j)))
44    3  p  2v           enddo
45    2  p            enddo
46    1  p          enddo
```

Block size: 4 KB
4 KB x 4 streams = 16 KB
-> Size in L1 cache

Loop 1

Array access causes cache hit

Loop 2



Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)
Before	0.00	2.08E+09	4.23E+08	0.20	7.61%
After	0.00	1.95E+09	2.35E+08	0.12	15.24%

L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)
4.23E+08	0.20	5.17%
2.35E+08	0.12	7.84%

L1D and L2 misses reduced

Strip Mining (Before Improvement)

FUJITSU

Array data cannot be fully cached and thus cannot be reused in Loop 2 because the number of iterations of Loop 1 is large. Consequently, the "No instruction commit because memory cache is busy" event occurs many times.

Source Before Improvement

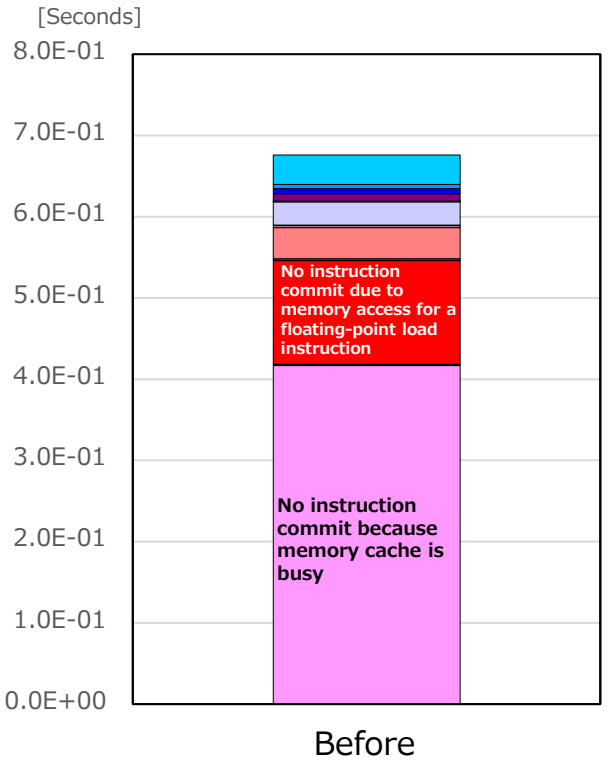
```
35          #pragma omp parallel for reduction(+:s1)
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
36   p        for(j=0;j<m;j++) {
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
37   p   8v      for(i=0;i<n;i++) {
38   p   8v        s1 = s1 + a[j][i] * (s3 * b[j][i] + c[j][i] * (s2 + s3 * d[j][i]));
39   p   8v      }
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.30, ITR: 96, MVE: 2, POL: S)
            <<<    PREFETCH(HARD) Expecte
            <<<      (unknown)
            <<< Loop-information  End >>>
40   p   2v      for(i=0;i<n-100;i++) {
41   p   2v        e[j][i] = s2 * (a[j][i] + b[j][i] * (s3 + c[j][i] * d[j][i]));
42   p   2v      }
43   p         }
44          }
```

Number of loop iterations: 375,000
Total array size: 12 MB
Since number of iterations is large, array data cannot be fully cached

Loop 1

Array access causes cache miss

Loop 2



Data cannot be reused, so the L1D and L2 miss rates are around 0.20, which is the theoretical value of stream access.

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	2.17E+09	4.23E+08	0.19	9.14%	90.86%	0.00%	4.23E+08	0.19	5.42%	95.53%	0.00%

Effect of Strip Mining (Source Tuning)

FUJITSU

Strip mining increases cache efficiency, resulting in improvement of the "No instruction commit because memory cache is busy" event.

Source After Improvement

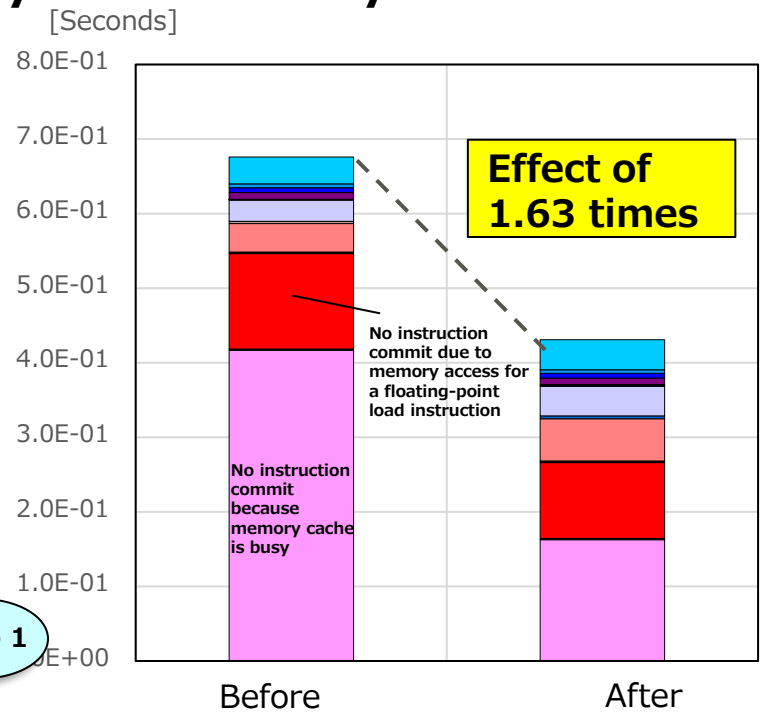
```
37          blki=4*1024/8;
38
39        #pragma omp parallel for reduction(+
40  p       for(j=0;j<m;j++) {
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
41  p         for(ii=0;ii<n;ii+=blki) {
42  p           int min1=MIN(ii+blki,n);
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
43  p   8v        for(i=ii;i<min1;i++) {
44  p   8v          s1 = s1 + a[j][i] * (s3 * b[j][i] + c[j][i] * (s2 + s3 * d[j][i]));
45  p   8v        }
46  p           int min2=MIN(ii+blki,n-100);
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.30, ITR: 96, MVE: 2, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
47  p   2v        for(i=ii;i<min2;i++) {
48  p   2v          e[j][i] = s2 * (a[j][i] + b[j][i] * (s3 + c[j][i] * d[j][i]));
49  p   2v        }
50  p         }
51  p       }
52        }
```

Block size: 4 KB
4 KB x 4 streams = 16 KB
-> Size in L1 cache

Loop 1

Array access causes cache hit

Loop 2



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)
Before	0.00	2.17E+09	4.23E+08	0.19	9.14%
After	0.00	1.99E+09	2.35E+08	0.12	19.90%

L1D and L2 misses reduced

L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)
4.23E+08	0.19	5.42%
2.35E+08	0.12	7.77%

FUJITSU

Loop Blocking

- What is Loop Blocking?
- Loop Blocking (Before Improvement)
- Effect of Loop Blocking (Source Tuning)
- Adverse Effect on Hardware Prefetch

What is Loop Blocking?

FUJITSU

Loop blocking executes source code divided by the specified blocking size in order to increase cache efficiency.

This can be considered as strip mining in two or more dimensions.

Example: Source Before Improvement

```
subroutine sub(a,b,m,n)
  integer n,m
  real*8 a(m,n),b(n,m)
  do j=1,m
    do i=1,n
      b(i,j)=a(j,i)
    enddo
  enddo
end subroutine
```

Array a: Stride access
Array b: Sequential access

Example: Source After Improvement

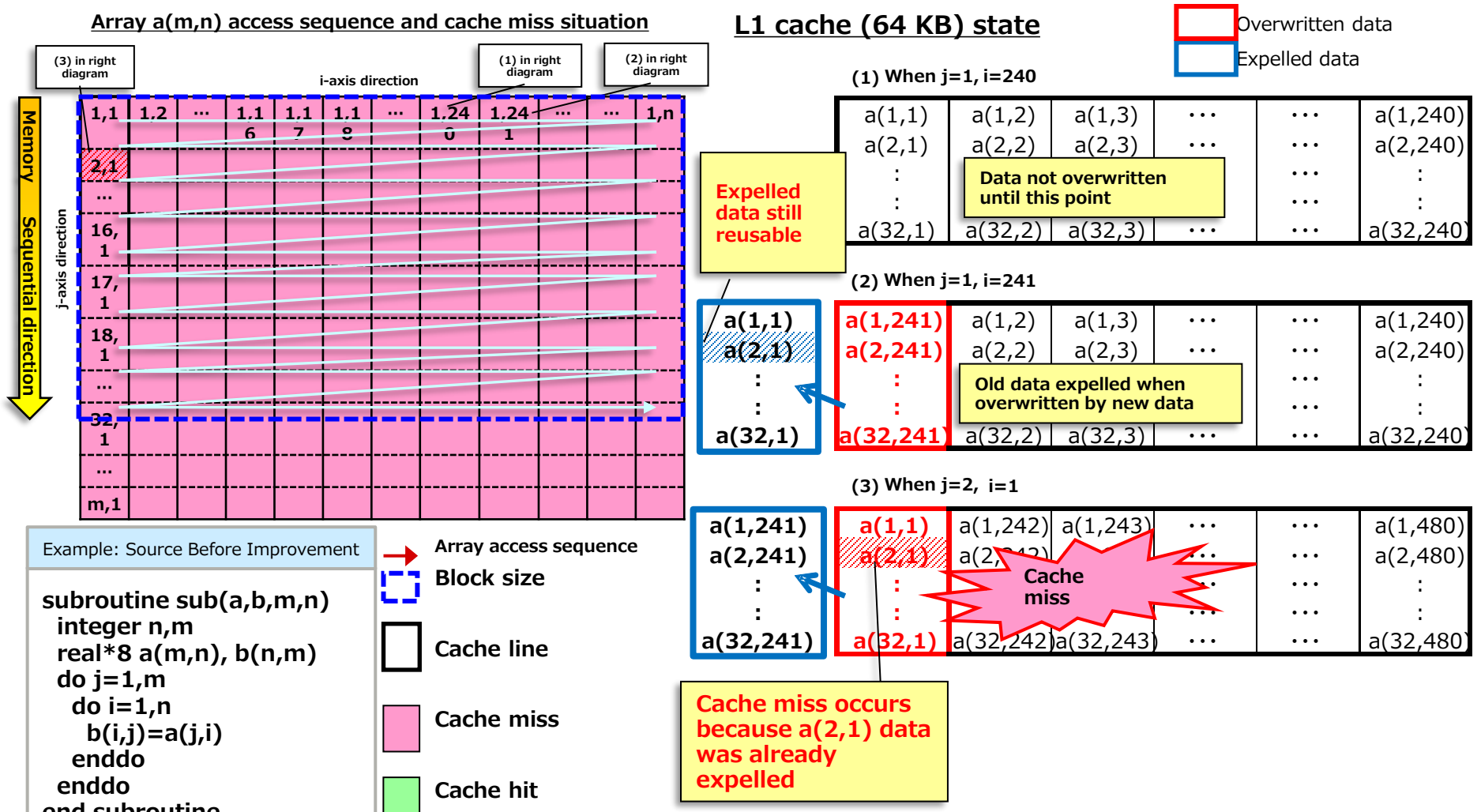
```
subroutine sub(a,b,m,n)
  parameter(blki=96,blkj=16)
  integer n,m
  real*8 a(m,n),b(n,m)
  do jj=1,m,blkj
    do ii=1,n,blki
      do j=jj,min(jj+blkj-1,m)
        do i=ii,min(ii+blki-1,n)
          b(i,j)=a(j,i)
        enddo
      enddo
    enddo
  enddo
end subroutine
```

Block size:
12 KB per array
(= 96 x 16 x 8 bytes)

What is Loop Blocking?

Array access (before improvement)

Memory is accessed every time i is updated because Array a has a stride access pattern. As a result, the data cached at a(1,1) is expelled before a(2,1) is accessed.

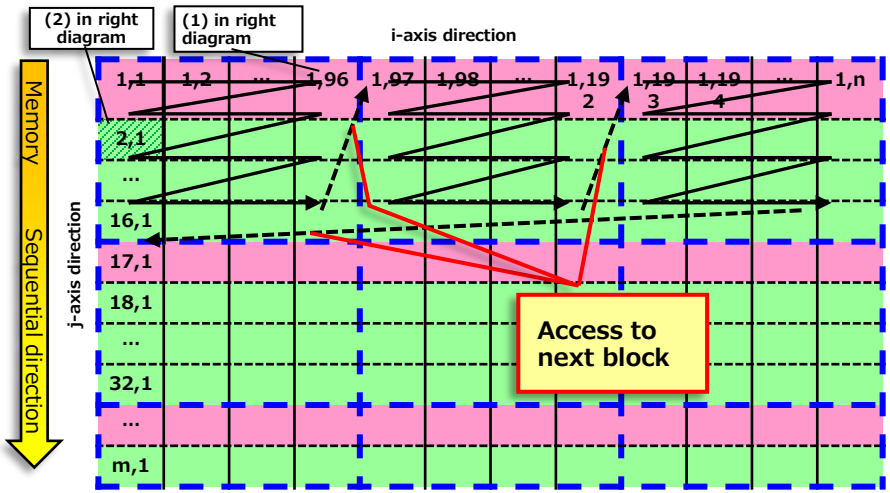


What is Loop Blocking?

- **Array access (after improvement) Block size: 96 x 16**

The array is accessed one block at a time in loop blocking. As a result, cache efficiency increases because a(2,1) access now hits the cache.

Array a(m,n) access sequence and cache miss situation



Example: Source After Improvement

```
subroutine sub(a,b,m,n)
  parameter(blki=96,blkj=16)
  integer n,m
  real*8 a(m,n),b(n,m)
  do jj=1,m, blkj
    do ii=1,n, blki
      do j=jj,min(jj+blkj-1,m)
        do i=ii,min(ii+blki-1,n)
          b(i,j)=a(j,i)
        enddo
      enddo
    enddo
  enddo
end subroutine
```

Array access sequence

Block size

Cache line

Cache miss

Cache hit

L1 cache (64 KB) state

(1) When j=1, i=96

a(1,1)	a(1,2)	a(1,3)	・・・	a(1,96)		
a(2,1)	a(2,2)	a(2,3)	・・・	a(2,96)		
:			・・・	:		
:			・・・	:		
a(32,1)	a(32,2)	a(32,3)	・・・	a(32,96)		

Array data cached

(2) When j=2, i=1

a(1,1)	a(1,2)	a(1,3)	・・・	a(1,96)		
a(2,1)	a(		・・・	a(2,96)		
:			・・・	:		
:			・・・	:		
a(32,1)	a(32,2)	a(32,3)	・・・	a(32,96)		

Cache hit

Cache hit occurs because data is still in cache

Point:
Loop blocking may have a negative impact on data continuity, possibly disabling hardware prefetch.
In such cases, use software prefetch.
For details, see Adverse Effect on Hardware Prefetch.

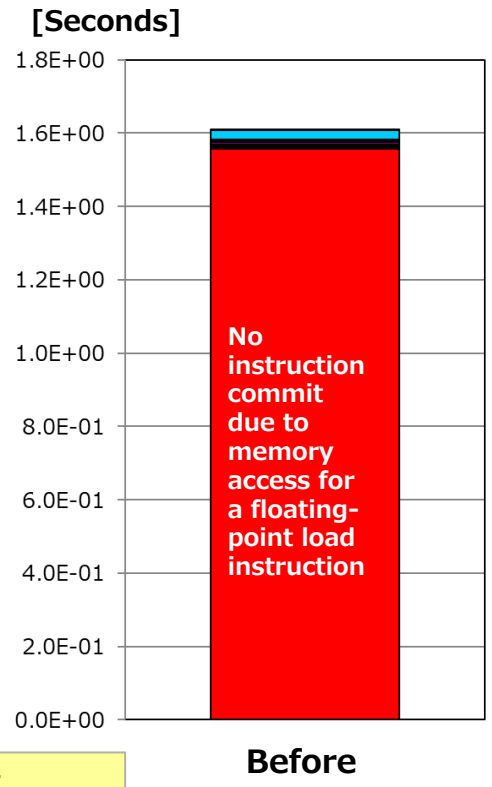
Fortran Loop Blocking (Before Improvement)

FUJITSU

Cache efficiency is low because Array a has a stride access pattern. Consequently, the "No instruction commit due to memory access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
48    1          !$omp do
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b
                <<< Loop-information  End >>>
49    2  p          do j=1,n2
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 1.72, ITR: 176, MVE: 6, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b
                <<< Loop-information  End >>>
50    3  p  2v          do i=1,n1
51    3  p  2v            b(i,j) = c0 + a(j,i)*(c1 + a(j,i)*(c2 + a(j,i)*(c3 + a(j,i)*
52    3                &          (c4 + a(j,i)*(c5 + a(j,i)*(c6 + a(j,i)*(c7 + a(j,i)*
53    3                &          (c8 + a(j,i)*c9))))))))
54    3  p  2v          enddo
55    2  p            enddo
56    1          !$omp enddo
```



Array a data is cached once in L1D at iteration i, but the data is expelled at the next j iteration(s). Consequently, a cache miss occurs.

Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	4.06E+08	1.28E+09	3.16	98.85%	1.15%	0.00%	1.31E+09	3.24	29.75%	71.85%	0.00%

Effect of Loop Blocking (Source Tuning)

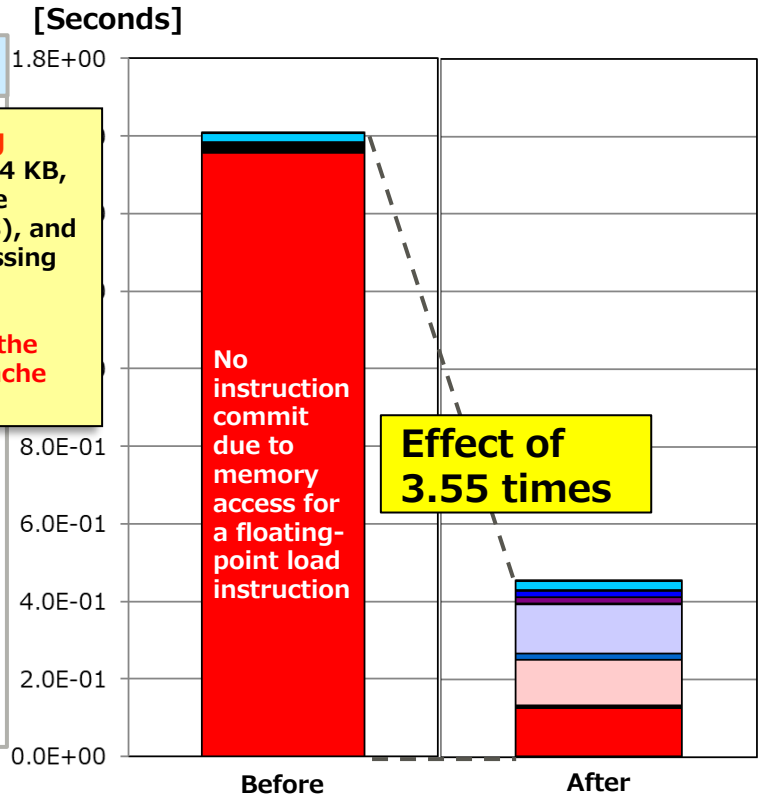
FUJITSU

Loop blocking increases cache efficiency by reusing Array a data. The result is improvement of the "No instruction commit due to memory access for a floating-point load instruction" event.

Source After Improvement (Source Tuning)

```
55    1          !$omp do
56    2  p         do jj=1,n2,16
57    3  p          do ii=1,n1,96
58    4  p           do j=jj,min(jj+16-1,n2)
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 1.7
                <<< Loop-information  End >>>
59    5  p  2v            do i=ii,min(ii+96-1,n1)
60    5  p  2v             b(i,j) = c0 + a(j,i)*(c1 + a(j,i)*(c2 + a(j,i)*(c3 + a(j,i)*
61    5             &          (c4 + a(j,i)*(c5 + a(j,i)*(c6 + a(j,i)*(c7 + a(j,i)*
62    5             &          (c8 + a(j,i)*c9))))))))
63    5  p  2v            enddo
64    4  p              enddo
65    3  p             enddo
66    2  p            enddo
67    1          !$omp enddo
```

Applying loop blocking
Since the L1 cache size is 64 KB, the size of each block in the cache is 12 KB (96 x 16 x 8), and the size required for processing each block is 24 KB (12 x 2 blocks).
The purpose is to increase the use efficiency of the L1D cache and L2 cache.



L1D and L2 misses reduced significantly

Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	4.06E+08	1.28E+09	3.16	98.85%	1.15%	0.00%	1.31E+09	3.24	29.75%	71.85%	0.00%
After	0.00	8.26E+08	1.69E+08	0.20	95.18%	4.82%	0.00%	1.56E+08	0.19	45.06%	61.56%	0.00%

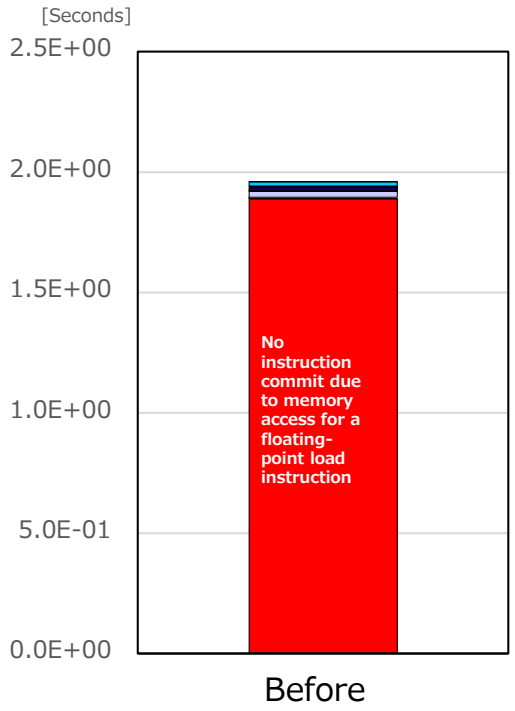
C/C++ Loop Blocking (Before Improvement)

FUJITSU

Cache efficiency is low because Array a has a stride access pattern. Consequently, the "No instruction commit due to memory access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
43          #pragma omp parallel
44           {
45             for(k=0;k<iter;k++) {
46          #pragma omp for
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
47   p           for(j=0;j<n2;j++) {
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 1.08, ITR: 128, MVE: 4, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
48   p    2v         for(i=0;i<n1;i++) {
49   p    2v           b[j][i] = c0 + a[i][j]*(c1 + a[i][j]*(c2 + a[i][j]*(c3 + a[i][j]*
50                         (c4 + a[i][j]*(c5 + a[i][j]*(c6 + a[i][j]*(c7 + a[i][j]*
51                         (c8 + a[i][j]*c9)))))));
52   p    2v         }
53   p             }
54                }
55              }
56            }
```



Array a data is cached once in L1D at iteration i, but the data is expelled at the next j iteration(s). Consequently, a cache miss occurs.

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	4.00E+08	1.28E+09	3.21	98.53%	1.47%	0.00%	1.33E+09	3.32	28.24%	73.08%	0.00%

Effect of Loop Blocking (Source Tuning)

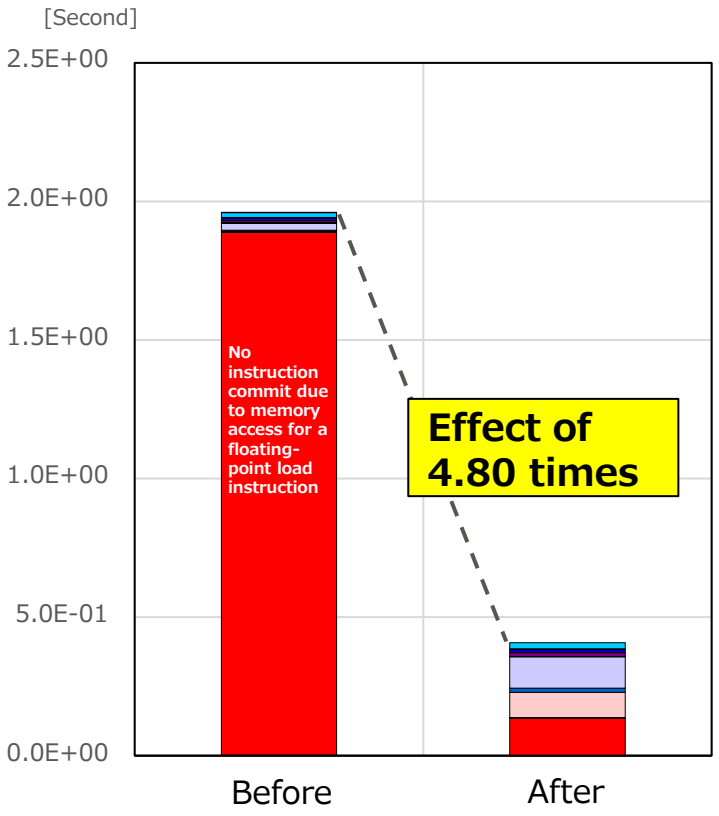
FUJITSU

Loop blocking increases cache efficiency by reusing Array a data. The result is improvement of the "No instruction commit due to memory access for a floating-point load instruction" event.

Source After Improvement (Source Tuning)

```
45          #pragma omp parallel
46          {
47            for(k=0;k<iter;k++) {
48          #pragma omp for
49  p           for(jj=0;jj<n2;jj=jj+16) {
50  p             for(ii=0;ii<n1;ii=ii+96) {
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      b
          <<< Loop-information  End >>>
51  p               for(j=jj;j<MIN(jj+16-1,n2);j++) {
52  p                 int st = MIN(ii+96-1,n1);
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 1.25, ITR: 128, MVE: 4, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      b
          <<< Loop-information  End >>>
53  p   2v            for(i=ii;i<st;i++) {
54  p   2v              b[j][i] = c0 + a[i][j]*(c1 + a[i][j]*(c2 + a[i][j]*(c3 + a[i][j]*
55                          (c4 + a[i][j]*(c5 + a[i][j]*(c6 + a[i][j]*(c7 + a[i][j]*
56                          (c8 + a[i][j]*c9)))))));
57  p   2v            }
58  p               }
59  p             }
60  p           }
61            }
62          }
63        }
```

Applying loop blocking
Since the L1 cache size is 64 KB, the size of each block in the cache is 12 KB (96 x 16 x 8), and the size required for processing each block is 24 KB (12 x 2 blocks).
The purpose is to increase the use efficiency of the L1D cache and L2 cache.



L1D and L2 misses reduced significantly

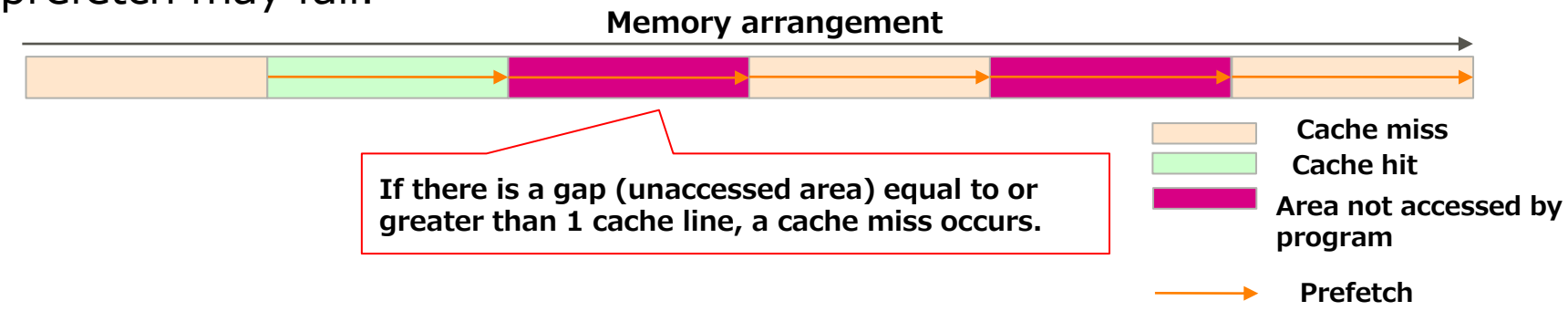
Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	4.00E+08	1.28E+09	3.21	98.53%	1.47%	0.00%	1.33E+09	3.32	28.24%	73.08%	0.00%
After	0.00	4.93E+08	1.70E+08	0.35	93.66%	6.34%	0.00%	1.40E+08	0.28	49.47%	53.12%	0.00%

Adverse Effect on Hardware Prefetch

- Loop blocking, outer loop prefetch, and other means have a negative impact on data continuity by reducing the block size, possibly disabling hardware prefetch. In such cases, you will need to use software prefetch.
- For details on how to specify software prefetch, see Using Software Prefetch.

- **Hardware prefetch**

Hardware prefetches data by predicting data access based on the regularity of memory access by programs.
If there is a gap equal to or greater than one cache line between data, prefetch may fail.



- **Software prefetch**

Software (compiler) analyzes programs and prefetches data by generating a prefetch instruction. Alternatively, from a specified instruction line, it generates a prefetch instruction for the relevant part.

FUJITSU

Sector Cache

- What is the Sector Cache?
- How to Use the Sector Cache
- Sector Cache: Case 1 (Before Improvement)
- Sector Cache: Case 1 (Source Tuning)
- Sector Cache: Case 2 (Before Improvement)
- Sector Cache: Case 2 (Source Tuning)

What is the Sector Cache?

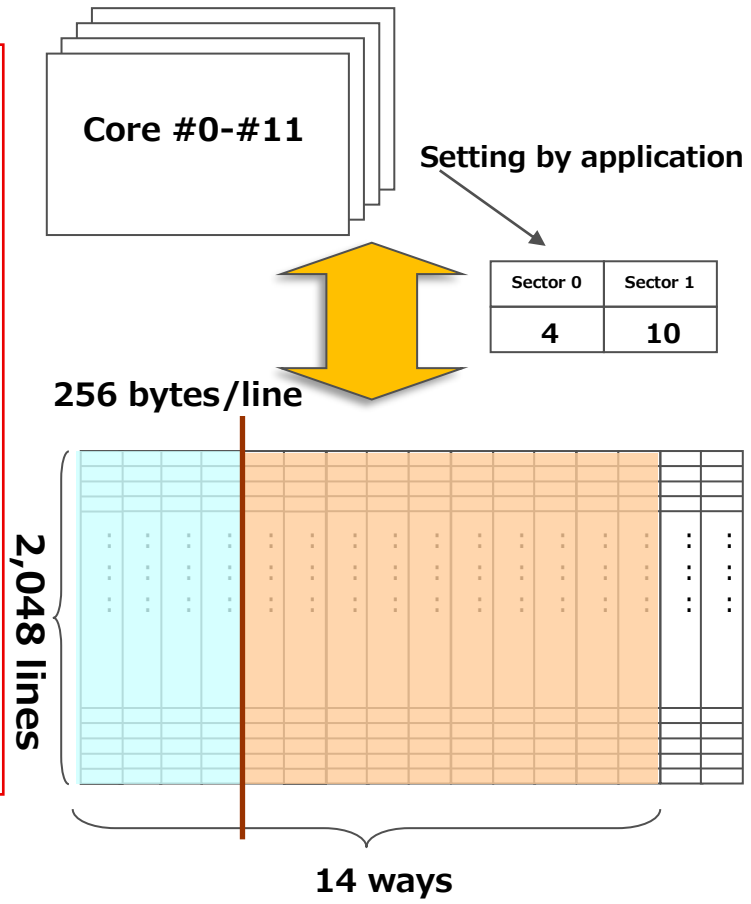
The sector cache is a cache mechanism that can prevent non-reusable data from expelling reusable data from the cache. An application can allocate reusable data and non-reusable to different sectors. (Reusable arrays use Sector 1, and others use Sector 0.)

Sector cache details

- You can set multiple sectors in both the L1D cache and L2 cache. The maximum number of sectors is 4 (※1) in L1D and 2 in L2.
- The number of ways specifies the capacity of each sector.
- The capacity works as a target value.
  Hardware controls sectors so that they approach the specified capacity at the line replacement time.
  -> Not forcibly disabled even when over the capacity
- Use the LRU (least recently used) algorithm to control expulsion within a sector.
- Applications can decide the usage of sectors 0 and 1. However, Sector 0 stores instruction sequences.
- In a secondary cache, the assistant core always uses two ways.

(※1) The L1D cache currently has 2 sectors. This specification is designed for easy use by customers.

Conceptual image of L2 cache usage



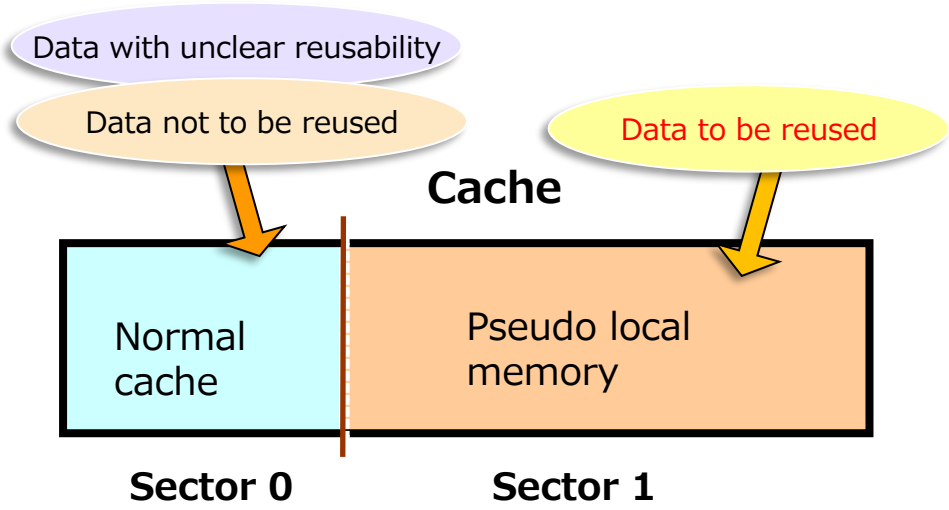
How to Use the Sector Cache (1/2)

FUJITSU

Sector cache: Pseudo local memory

Software can use sectors separately according to data reusability.

- **Data used -> Use Sector 1**
- **Other data -> Use Sector 0**
- **Data in Sector 1 is not expelled by other data.**
- **Instruction lines can specify the arrays stored in Sector 1.**



Example using compiler instruction lines to specify the sector cache

```
!OCL SCACHE_ISOLATE_WAY(L2=10)
!OCL SCACHE_ISOLATE_ASSIGN(a)
do j=1,m
   do i=1,n
      a(i) = a(i) + b(i,j)*c(i,j)
   enddo
enddo
!OCL END_SCACHE_ISOLATE_ASSIGN
!OCL END_SCACHE_ISOLATE_WAY
```

How they are specified under the old specifications (K computer, FX100)

```
!OCL CACHE_SECTOR_SIZE(4,10)
!OCL CACHE_SUBSECTOR_ASSIGN(a)
do j=1,m
   do i=1,n
      a(i) = a(i) + b(i,j)*c(i,j)
   enddo
enddo
!OCL END_CACHE_SUBSECTOR
!OCL END_CACHE_SECTOR_SIZE
```

<Purpose>
To prevent Array a, which is reusable, from being expelled from the cache due to access to Arrays b and c during the loop

How to Use the Sector Cache (2/2)

FUJITSU

To use the sector cache, specify the following optimization control lines.

Optimization Specifier (Fortran)	Meaning	Optimization Control Line Specifiable?			
		By Program	By DO Loop	By Statement	By Array Assignment Statement
SCACHE_ISOLATE_WAY(L2=n1[,L1=n2]) END_SCACHE_ISOLATE_WAY	Specifies the maximum number of ways for Sector 1 of the primary cache and secondary cache.	Yes	No	Yes	No
SCACHE_ISOLATE_ASSIGN(array1[,array2]…) END_SCACHE_ISOLATE_ASSIGN	Specifies the arrays stored in Sector 1 of the cache.	Yes	No	Yes	No

Optimization Specifier (C/C++)	Meaning	Optimization Control Line Specifiable?			
		global	procedure	loop	statement
scache_isolate_way(L2=n1[,L1=n2]) end_scache_isolate_way	Specifies the maximum number of ways for Sector 1 of the primary cache and secondary cache.	No	Yes	No	Yes
scache_isolate_assign(array1[,array2]…) end_scache_isolate_assign	Specifies the arrays stored in Sector 1 of the cache.	No	Yes	No	Yes

- **Note**
- In the secondary cache, the assistant core always uses two ways. Therefore, the ranges of values that can be specified in n1 and n2 are as follows:
  $0 \leqq n1 \leqq$ *maximum number of ways of secondary cache* - 2
  $0 \leqq n2 \leqq$ *maximum number of ways of primary cache*
- For a CMG that contains an assistant core, the assistant core uses part (2 ways = 1 MiB) of the L2 cache. Therefore, for the CMG, the maximum number of ways of the secondary cache is 14 and the size is 7 MiB.
- Sector Cache optimization is not available in Clang Mode.

A64FX Specifications	
Number of CMGs	4
L1I cache size	64 KiB/4 ways
L1D cache size	64 KiB/4 ways
L2 cache size	32 MiB/16 ways (8 MiB/CMG)

Fortran Sector Cache: Case 1 (Before Improvement)

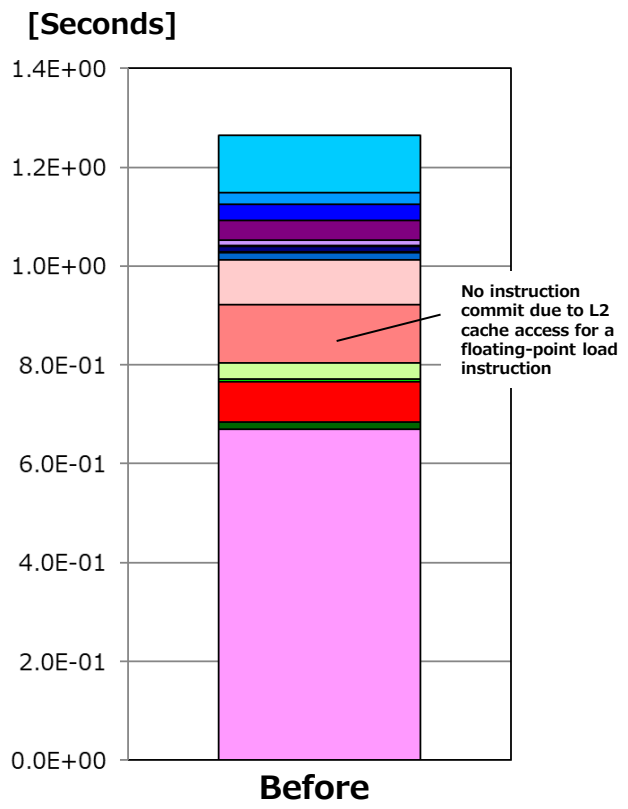
FUJITSU

Array b data is expelled from the cache and thus cannot be reused. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
66                parameter(n=8*1024*1024, m=9*512*1024/8)
67                real*8    a(n), b(m), s
68                integer*8 c(n)
69                real*8    dummy1(140),dummy2(140)
70                common /data/a,dummy1,c,dummy2,b
71
                  <<< Loop-information Start >>>
                  <<<  [PARALLELIZATION]
                  <<<    Standard iteration count: 843
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 176, MVE: 4, POL: S)
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      c, a
                  <<< Loop-information  End >>>
72    1  pp   2v          do i=1,n
73    1   p   2v            a(i) = a(i) + s * b(c(i))
74    1   p   2v          enddo
```

Array size
a: 64 MiB
b: 4.5 MiB
c: 64 MiB



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	4.76E+09	7.89E+08	0.17	0.89%	99.11%	0.00%	7.34E+08	0.15	0.77%	100.00%	0.00%

	Memory throughput (GB/s)
Before	203.11

Memory throughput is bottleneck

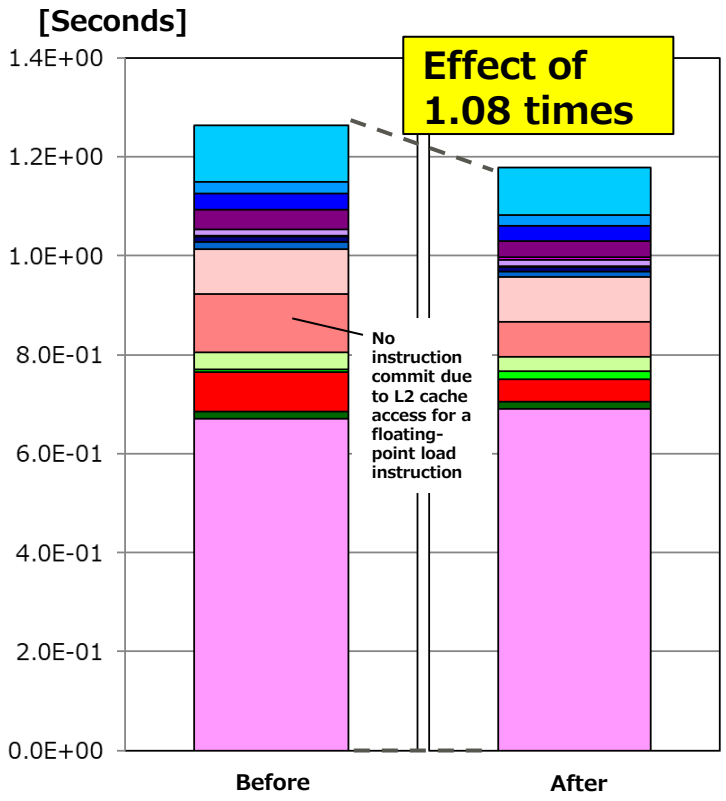
High L2 cache miss rate

Fortran Sector Cache: Case 1 (Source Tuning)

Storing Array b in Sector 1 increases cache efficiency. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement (Optimization Control Line Tuning)

```
58                parameter(n=8*1024*1024, m=9*512*1024/8)
59                 real*8    a(n), b(m), s
60                 integer*8 c(n)
61                 real*8    dummy1(140),dummy2(140)
62                 common /data/a,dummy1,c,dummy2,b
63
64              !OCL SCACHE_ISOLATE_WAY(L2=10)
65              !OCL SCACHE_ISOLATE_ASSIGN(b)
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 843
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 176, MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, a
                <<< Loop-information  End >>>
66    1  pp   2v          do i=1,n
67    1   p   2v            a(i) = a(i) + s * b(c(i))
68    1   p   2v          enddo
69              !OCL END_SCACHE_ISOLATE_ASSIGN
70              !OCL END_SCACHE_ISOLATE_WAY
```



	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	4.76E+09	7.89E+08	0.17	0.89%	99.11%	0.00%	7.34E+08	0.15	0.77%	100.00%	0.00%
After	0.00	5.19E+09	7.93E+08	0.15	1.19%	98.81%	0.01%	5.99E+08	0.12	1.93%	99.69%	0.00%

	Memory throughput (GB/s)
Before	203.11
After	188.07

L2 misses reduced

C/C++ Sector Cache: Case 1 (Before Improvement)

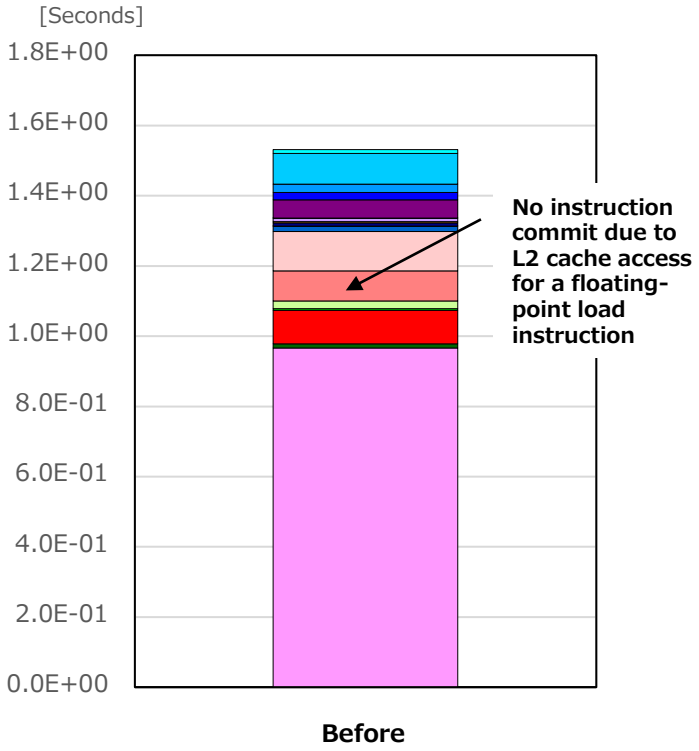
FUJITSU

Array b data is expelled from the cache and thus cannot be reused. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
56          void sub(double s) {
57            long long int i;
58
59          #pragma omp parallel for
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.33, ITR: 160,
                                MVE: 4, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      c, a
            <<< Loop-information  End >>>
60  p     2v    for(i=0;i<n;i++) {
61  p     2v      a[i] = a[i] + s * b[c[i]];
62  p     2v    }
63          }
```

Array declaration: size
double a[8388608]: 64MiB
double b[589824]: 4.5MiB
long long int c[8388608]: 64MiB



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	4.85E+09	7.89E+08	0.16	1.07%	98.92%	0.01%	7.39E+08	0.15	0.78%	100.00%	0.00%

Statistics	Memory throughput (GB/s)
Before	167.47

High memory throughput

High L2 cache miss rate

Sector Cache: Case 1 (Source Tuning)

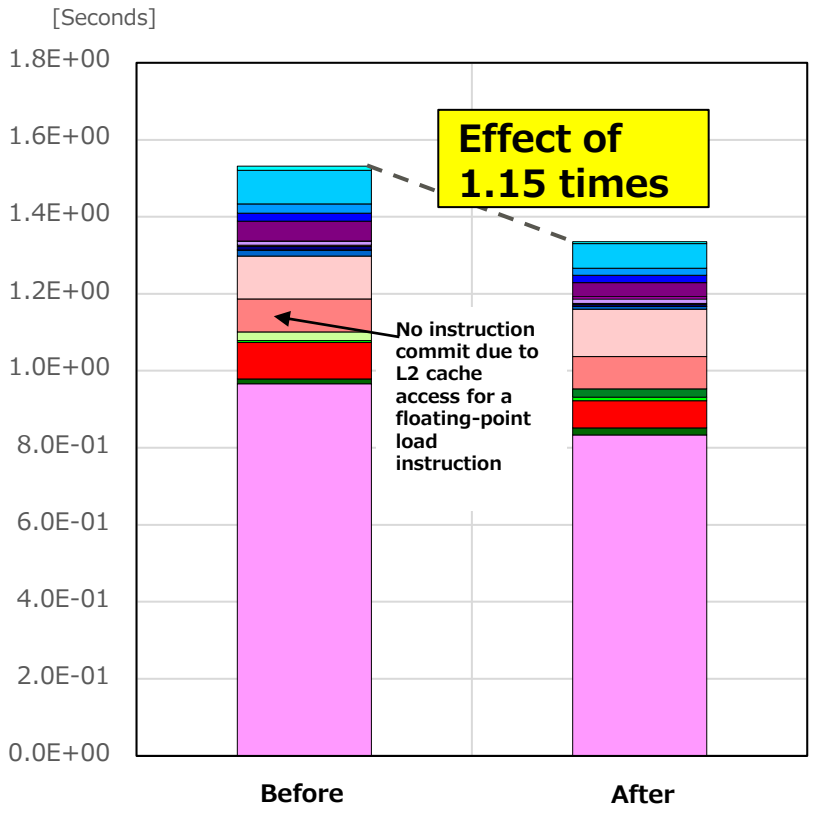
FUJITSU

Storing Array b in Sector 1 increases cache efficiency. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement (Optimization Control Line Tuning)

Array declaration: size
double a[8388608]: 64MiB
double b[589824]: 4.5MiB
long long int c[8388608]: 64MiB

```
56          void sub(double s){
57            long long int i;
58          #pragma statement scache_isolate_way L2=10
59          #pragma statement scache_isolate_assign b
60          #pragma omp parallel for
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.33, ITR: 160,
                                       MVE: 4, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      c, a
            <<< Loop-information  End >>>
61   p    2v    for(i=0;i<n;i++) {
62   p    2v      a[i] = a[i] + s * b[c[i]];
63   p    2v    }
64          #pragma statement end_scache_isolate_assign
65          #pragma statement end_scache_isolate_way
66          }
```



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	4.85E+09	7.89E+08	0.16	1.07%	98.92%	0.01%	7.39E+08	0.15	0.78%	100.00%	0.00%
After	0.00	5.03E+09	7.91E+08	0.16	1.23%	98.78%	0.00%	5.48E+08	0.11	1.99%	98.64%	0.00%

Statistics	Memory throughput (GB/s)
Before	167.47
After	155.55

L2 misses reduced

Fortran Sector Cache: Case 2 (Before Improvement)

FUJITSU

Array u data is expelled from the cache and thus cannot be reused. Consequently, the "No instruction commit because memory cache is busy" event occurs many times.

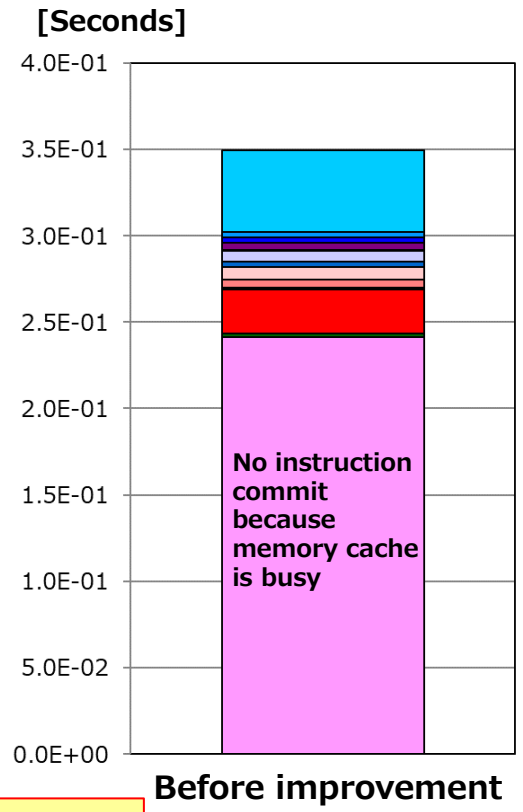
Source Before Improvement

```
167   1  s          do iter = 1, niter
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration cou
                <<< Loop-information  End >
168   2  pp           do k=1,n3-2
                <<< Loop-information Start >
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      u, rhs, unew
                <<< Loop-information  End >>
169   3  p             do j=1,n2-2
                <<< Loop-information Start >>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.71, ITR: 136, MVE: 9, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      u, rhs, unew
                <<< Loop-information  End >>>
170   4  p  v          do i=1,n1-2
171   4  p  v            unew(i,j,k) = &
172   4                    ((u(i+1,j,k) + u(i-1,j,k)) * h1sqinv  &
173   4                    +(u(i,j+1,k) + u(i,j-1,k)) * h2sqinv &
174   4                    +(u(i,j,k+1) + u(i,j,k-1)) * h3sqinv &
175   4                    -rhs(i,j,k)) * hhhinv
176   4  p  v          end do
177   3  p            end do
178   2  p          end do
179   1            end do
```

n1=452
n2=52
n3=322

Array size
unew: 60.5 MB
u: 60.5 MB
rhs: 60.5 MB

Preferably, Array u is cached so that dimensions i and j of Array u are reusable.



Memory throughput is bottleneck

	Memory throughput (GB/s)
Before	215.62

Cache	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	2.46E+08	0.15	2.57%	98.19%	0.00%

Fortran Sector Cache: Case 2 (Source Tuning)

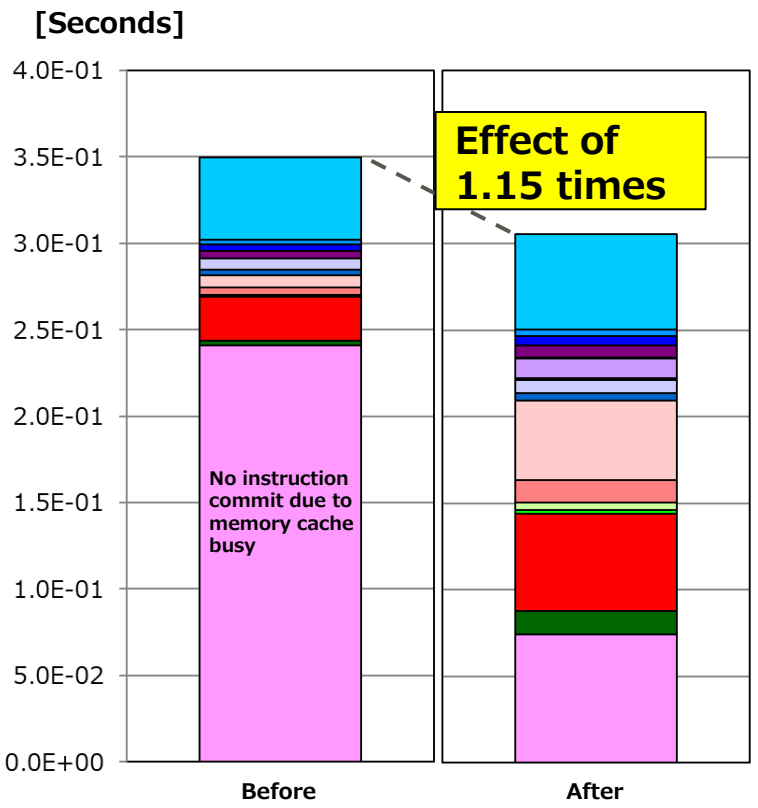
FUJITSU

Storing part of dimension k of Array u in Sector 1 increases cache efficiency. The result is improvement of the "No instruction commit because memory cache is busy" event.

Source After Improvement

```
166              !OCL SCACHE_ISOLATE_WAY(L2=13)
167              !OCL SCACHE_ISOLATE_ASSIGN(u)
168    1  s          do iter = 1, niter
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 2
                <<< Loop-information  End >>>
169    2  pp           do k=1,n3-2
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      u, rhs, unew
                <<< Loop-information  End >>>
170    3  p             do j=1,n2-2
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.71, ITR: 136, MVE: 9, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      u, rhs, unew
                <<< Loop-information  End >>>
171    4  p  v            do i=1,n1-2
172    4  p  v              unew(i,j,k) = &
173    4                     ((u(i+1,j,k) + u(i-1,j,k)) * h1sqinv  &
174    4                     +(u(i,j+1,k) + u(i,j-1,k)) * h2sqinv &
175    4                     +(u(i,j,k+1) + u(i,j,k-1)) * h3sqinv &
176    4                     -rhs(i,j,k)) * hhhinv
177    4  p  v            end do
178    3  p              end do
179    2  p            end do
180    1            end do
181              !OCL END_SCACHE_ISOLATE_ASSIGN
182              !OCL END_SCACHE_ISOLATE_WAY
```

Array u reusability increased



L2 misses reduced

	Memory throughput (GB/s)
Before	215.62
After	205.39

	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	2.46E+08	0.15	2.57%	98.19%	0.00%
After	1.95E+08	0.10	13.43%	87.39%	0.00%

Sector Cache: Case 2 (Before Improvement)

FUJITSU

Array u data is expelled from the cache and thus cannot be reused. Consequently, the "No instruction commit because memory cache is busy" event occurs many times.

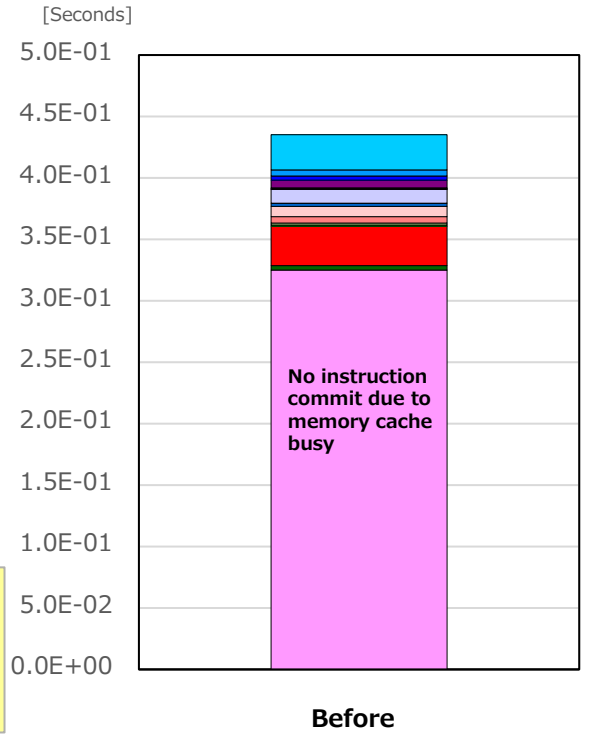
Source Before Improvement

```
107            for (iter=0; iter<niter; iter++){
108           #pragma omp parallel for private(i,j,k)
109   p         for(k=1;k<=n3-2; k++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
110   p         for(j=1;j<=n2-2;j++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.12, ITR: 120, MVE: 8, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
111   p    v        for(i=1;i<=n1-2;i++){
112   p    v          unew[k][j][i] =
113                   ((u[k][j][i+1] + u[k][j][i-1]) * h1sqinv
114                   +(u[k][j+1][i] + u[k][j-1][i]) * h2sqinv
115                   +(u[k+1][j][i] + u[k-1][j][i]) * h3sqinv
116                   -rhs[k][j][i]) * hhhinv;
117   p    v        }
118   p           }
119   p         }
120
121           }
```

n1=452
n2=52
n3=322

Array size
unew: 60.5MB
u: 60.5MB
rhs: 60.5MB

Preferably, Array u is cached so that dimensions i and j of Array u are reusable.



High memory throughput

Statistics	Memory throughput (GB/s)
Before	173.23

Cache	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	2.46E+08	0.15	2.38%	98.32%	0.00%

Sector Cache: Case 2 (Source Tuning)

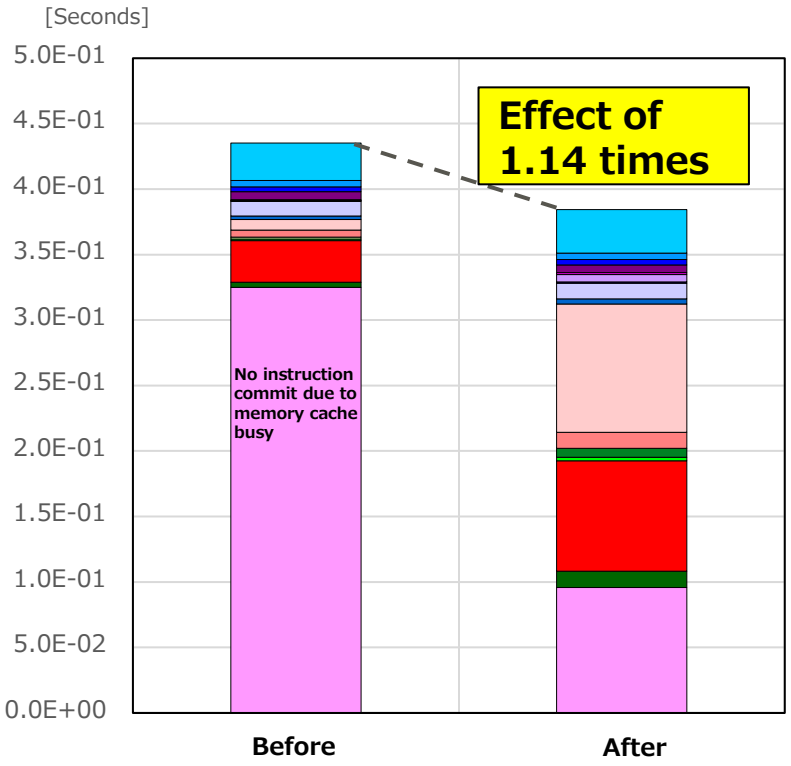
FUJITSU

Storing part of dimension k of Array u in Sector 1 increases cache efficiency. The result is improvement of the "No instruction commit because memory cache is busy" event.

Source After Improvement

```
107          #pragma statement scache_isolate_way L2=13
108          #pragma statement scache_isolate_assign u
109            for (iter=0; iter<niter; iter++){
110          #pragma omp parallel for private(i,j,k)
111    p           for(k=1;k<=n3-2; k++){
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
112    p             for(j=1;j<=n2-2;j++){
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 2.12, ITR: 120, MVE: 8, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
113    p     v          for(i=1;i<=n1-2;i++){
114    p     v            unew[k][j][i] =
115                       ((u[k][j][i+1] + u[k][j][i-1]) * h1sqinv
116                       +(u[k][j+1][i] + u[k][j-1][i]) * h2sqinv
117                       +(u[k+1][j][i] + u[k-1][j][i]) * h3sqinv
118                       -rhs[k][j][i]) * hhhinv;
119    p     v          }
120    p              }
121    p            }
122
123            }
124          #pragma statement end_scache_isolate_assign
125          #pragma statement end_scache_isolate_way
```

Array u reusability increased



Statistics	Memory throughput (GB/s)
Before	173.23
After	167.47

L2 misses reduced

Cache	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	2.46E+08	0.15	2.38%	98.32%	0.00%
After	1.99E+08	0.12	13.90%	86.99%	0.00%

FUJITSU

Loop Interchange

- What is Loop Interchange?
- Loop Interchange (Before Improvement)
- Loop Interchange Tuning Details
- Effect of Loop Interchange (Source Tuning)

What is Loop Interchange?

FUJITSU

Loop interchange is a means to increase data access efficiency by changing the order of loops in a multi-loop task.
In Fortran, arrays are stored in column-major order. Therefore, operation will be faster when the order of loops is changed as shown below for sequential access.

(In C, arrays are stored in row-major order, so the order is reversed compared to Fortran.)

Example: Source Before Improvement

```
do j=1,n1
  do i=1,n2
    a(j,i) = b(j,i) * c(j,i)
  enddo
enddo
```

Low cache efficiency since Arrays a, b, and c have stride access patterns

Example: Source After Improvement

```
do i=1,n2
  do j=1,n1
    a(j,i) = b(j,i) * c(j,i)
  enddo
enddo
```

Cache efficiency improved by changing order of loops to sequential access

- **Points**
  - **Note that if the number of iterations of the innermost loop is small, software pipelining may not be performed.**
    However, if the innermost loop can be fixed at the SIMD length, software pipelining is performed in its outer loops.
  - **If the access direction is different between stored and loaded arrays, performance will increase more through sequential access to the stored array.**
- **Note**
  - **Loop interchange optimization is not available in Clang Mode.**

Loop Interchange Tuning Details

Source Before Improvement

```
do j=1,n1
   do i=1,n2
     a(i) = s1 + c(j,i) / (s1 + s2 / d(j,i))
   enddo
   do i=2,n2
     b(j,i) = a(i) / (s2 + s1 / a(i-1))
   enddo
enddo
```

Loop 1

Loop 2

Low cache efficiency since Arrays b, c, and d have stride access pattern

(1) Array a Converted to 2-Dimensional Array

```
do j=1,n1
   do i=1,n2
     a(j,i) = s1 + c(j,i) / (s1 + s2 / d(j,i))
   enddo
   do i=2,n2
     b(j,i) = a(j,i) / (s2 + s1 / a(j,i-1))
   enddo
enddo
```

Eliminated dependency on Array a, which inhibited loop fission

(2) Loops 1 and 2 Separated

```
do j=1,n1
   do i=1,n2
     a(j,i) = s1 + c(j,i) / (s1 + s2 / d(j,i))
   enddo
enddo

do j=1,n1
   do i=2,n2
     b(j,i) = a(j,i) / (s2 + s1 / a(j,i-1))
   enddo
enddo
```

(3) Loops Interchanged

```
do i=1,n2
   do j=1,n1
     a(j,i) = s1 + c(j,i) / (s1 + s2 / d(j,i))
   enddo
enddo
- - - - - - - - - - - - - - - - - - - - - - -
do j=2,n2
   do j=1,n1
     b(j,i) = a(j,i) / (s2 + s1 / a(j,i-1))
   enddo
enddo
```

Cache efficiency improved since Arrays b, c, and d now have sequential access patterns

Fortran Loop Interchange (Before Improvement)

FUJITSU

Cache efficiency is low because Arrays b, c, and d have stride access patterns. Consequently, the "No instruction commit due to access for a floating-point load instruction" events occur many times.

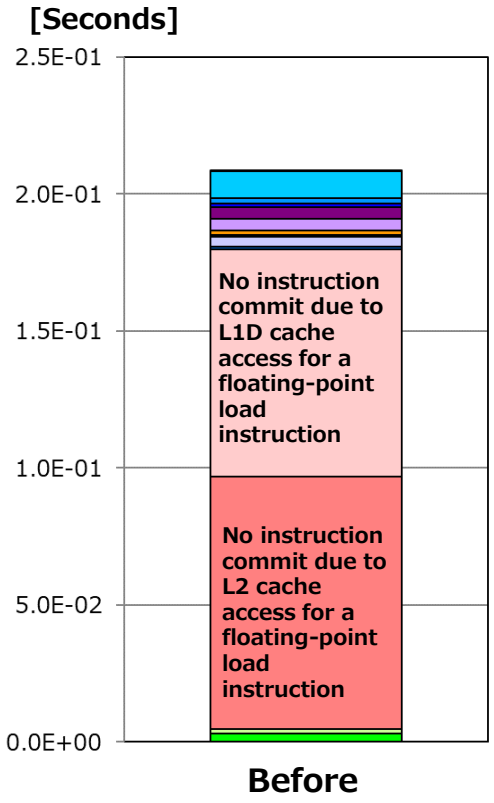
Source Before Improvement

```
 45                 real*8 a(n2),b(n1,n2),c(n1,n2),d(n1,n2)
 46                 real*8 s1,s2
 47                 integer n1,n2
 48                !$omp parallel
 49                !$omp do private(a)
 50    1   p          do j=1,n1
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 1.96, ITR: 112, MVE: 3, POL: S)
                  <<< Loop-information  End >>>
 51    2   p  2v         do i=1,n2
 52    2   p  2v           a(i) = s1 + c(j,i) / (s1 + s2 / d(j,i))
 53    2   p  2v         enddo
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 2.27, ITR: 96, MVE: 2, POL: S)
                  <<< Loop-information  End >>>
 54    2   p  2v         do i=2,n2
 55    2   p  2v           b(j,i) = a(i) / (s2 + s1 / a(i-1))
 56    2   p  2v         enddo
 57    1   p          enddo
 58                !$omp end do
 59                !$omp end parallel
```

Low cache efficiency since Arrays b, c, and d have stride access pattern

Loop 1

Loop 2



High L1D miss rates

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.60E+08	3.89E+08	1.08	98.28%	1.72%	0.00%	1.07E+04	0.00	62.54%	46.87%	0.00%

Fortran Effect of Loop Interchange (Source Tuning)

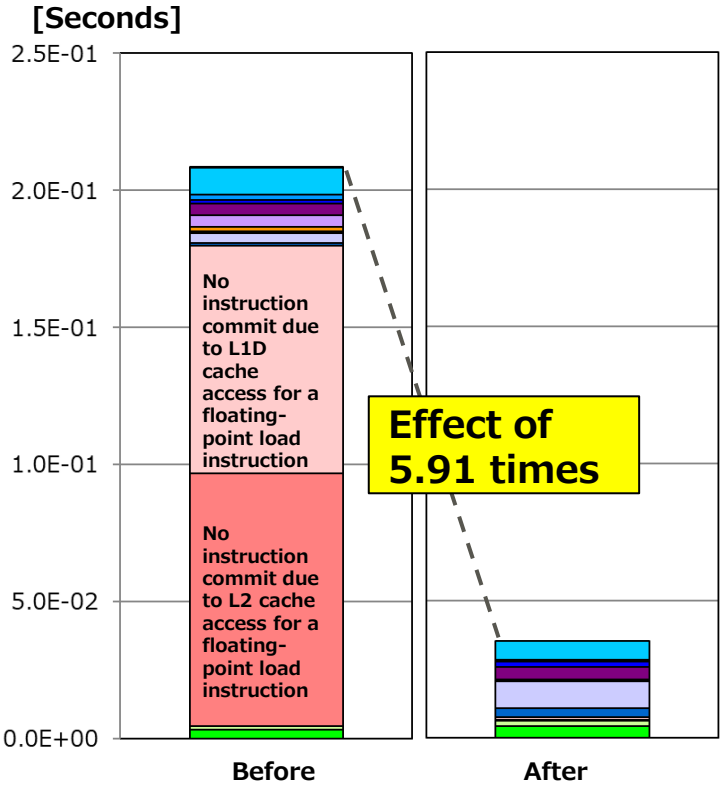
FUJITSU

Loop interchange increases cache efficiency through sequential access to arrays. The result is improvement of the "No instruction commit due to access for a floating-point load instruction" event.

Source After Improvement

```
45                  real*8 a(n1,n2),b(n1,n2),c(n1,n2),d(n1,n2)
46                  real*8 s1,s2
47                  integer n1,n2
48                 !$omp parallel
49                 !$omp do
50    1  p             do i=1,n2
                  <<< Loop-information Sta
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 2.13, ITR: 112, MVE: 2, POL: S)
                  <<< Loop-information  End >>>
51    2  p  2v          do j=1,n1
52    2  p  2v            a(j,i) = s1 + c(j,i) / (s1 + s2 / d(j,i))
53    2  p  2v          enddo
54    1  p           enddo
55                 !$omp end do
56                 !$omp do
57    1  p             do i=2,n2
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 2.04, ITR: 96, MVE: 2, POL: S)
                  <<< Loop-information  End >>>
58    2  p  2v          do j=1,n1
59    2  p  2v            b(j,i) = a(j,i) / (s2 + s1 / a(j,i-1))
60    2  p  2v          enddo
61    1  p           enddo
62                 !$omp end do
63                 !$omp end parallel
```

Tuning details
(1) Array a converted to 2-dimensional array
(2) Loops 1 and 2 separated
(3) Loops interchanged



Number of L1D misses reduced significantly

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.60E+08	3.89E+08	1.08	98.28%	1.72%	0.00%	1.07E+04	0.00	62.54%	46.87%	0.00%
After	0.00	1.94E+08	2.15E+07	0.11	9.28%	90.72%	0.00%	8.66E+03	0.00	17.98%	87.84%	0.00%

Loop Interchange (Before Improvement)

FUJITSU

Cache efficiency is low because Arrays b, c, and d have stride access patterns. Consequently, the "No instruction commit due to access for a floating-point load instruction" events occur many times.

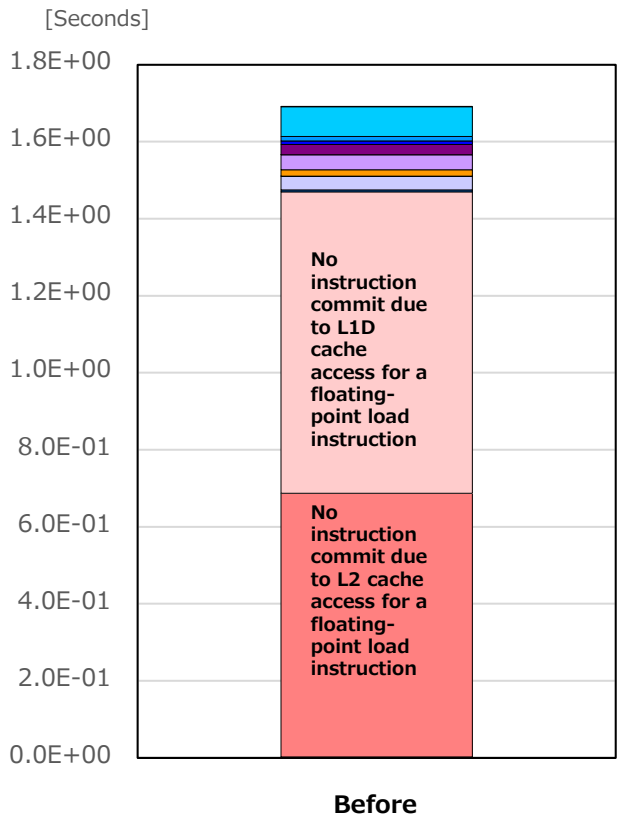
Source Before Improvement

```
33          void sub(int n,int m,double s1,double s2) {
34            double a[n2];
35            int i,j;
36
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      a
          <<< Loop-information  End >>>
37            for(j=0;j<n1;j++) {
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 1.92, ITR: 112, MVE: 3, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
          <<<      a
          <<< Loop-information  End >>>
38        2v      for(i=0;i<n2;i++) {
39        2v        a[i] = s1 + c[i][j] / (s1 + s2 / d[i][j]);
40        2v      }
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.18, ITR: 96, MVE: 2, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      a
          <<< Loop-information  End >>>
41        2v      for(i=1;i<n2;i++) {
42        2v        b[i][j] = a[i] / (s2 + s1 / a[i-1]);
43        2v      }
44            }
45          }
```

Low cache efficiency since Arrays b, c, and d have stride access pattern

Loop 1

Loop 2



High L1D miss rates

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	1.87E+09	3.89E+08	0.21	97.94%	2.06%	0.00%	2.18E+04	0.00	71.23%	38.77%	0.00%

Effect of Loop Interchange (Source Tuning)

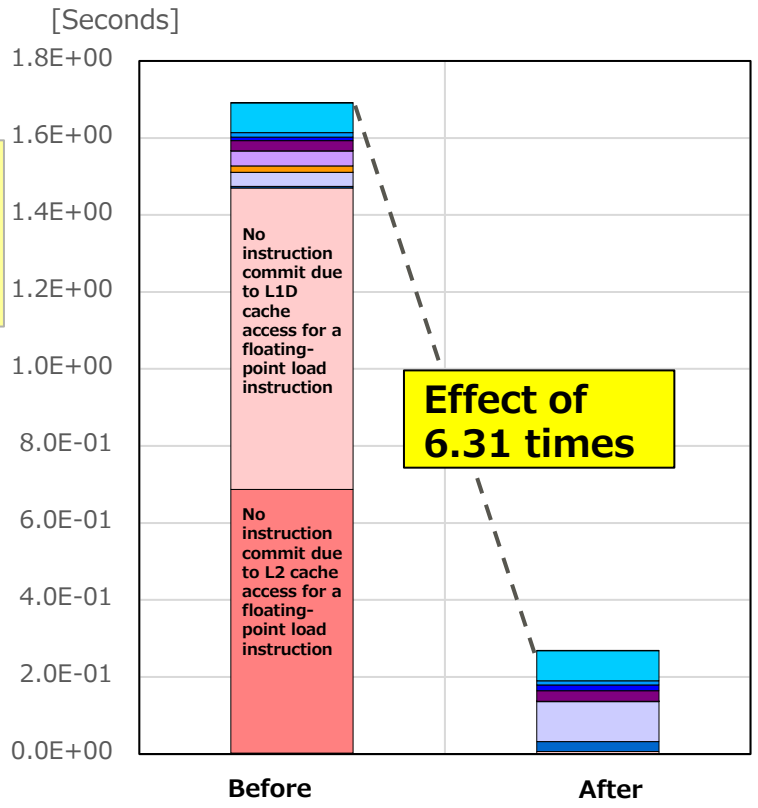
FUJITSU

Loop interchange increases cache efficiency through sequential access to arrays. The result is improvement of the "No instruction commit due to access for a floating-point load instruction" event.

Source After Improvement

```
34          void sub(int n1,int n2,double s1,double s2)
35          {
36            double a[m][n];
37            int i,j;
38
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      d, c, a
           <<< Loop-information  End >>>
39            for(i=0;i<n2;i++) {
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 1.88, ITR: 96, MVE: 2, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      c, d, a
           <<< Loop-information  End >>>
40        2v      for(j=0;j<n1;j++) {
41        2v        a[i][j] = s1 + c[i][j] / (s1 + s2 / d[i][j]);
42        2v      }
43            }
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      a, b
           <<< Loop-information  End >>>
44            for(i=0;i<n2;i++) {
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 2.09, ITR: 96, MVE: 2, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      a, b
           <<< Loop-information  End >>>
45        2v      for(j=0;j<n1;j++) {
46        2v        b[i][j] = a[i][j] / (s2 + s1 / a[i-1][j]);
47        2v      }
48            }
49          }
```

Tuning details
(1) Array a converted to 2-dimensional array
(2) Loops 1 and 2 separated
(3) Loops interchanged



Number of L1D misses reduced significantly

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	1.87E+09	3.89E+08	0.21	97.94%	2.06%	0.00%	2.18E+04	0.00	71.23%	38.77%	0.00%
After	0.00	1.87E+09	2.27E+07	0.01	14.23%	85.77%	0.00%	3.63E+04	0.00	42.14%	62.84%	0.00%

FUJITSU

Loop Fusion

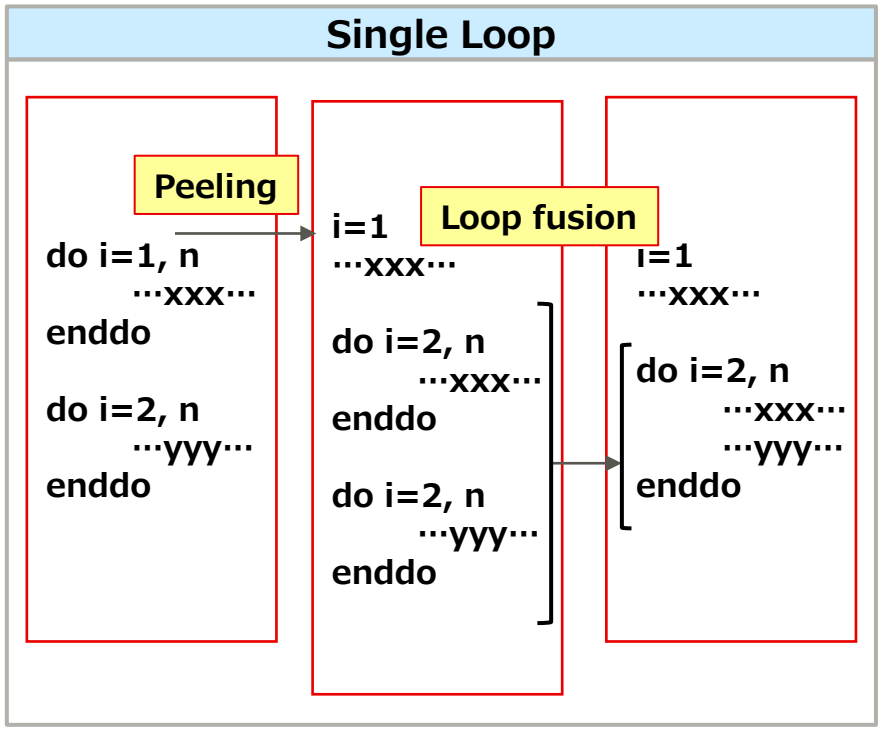
- What is Loop Fusion?
- Loop Fusion (Before Improvement)
- Loop Fusion (Source Tuning)

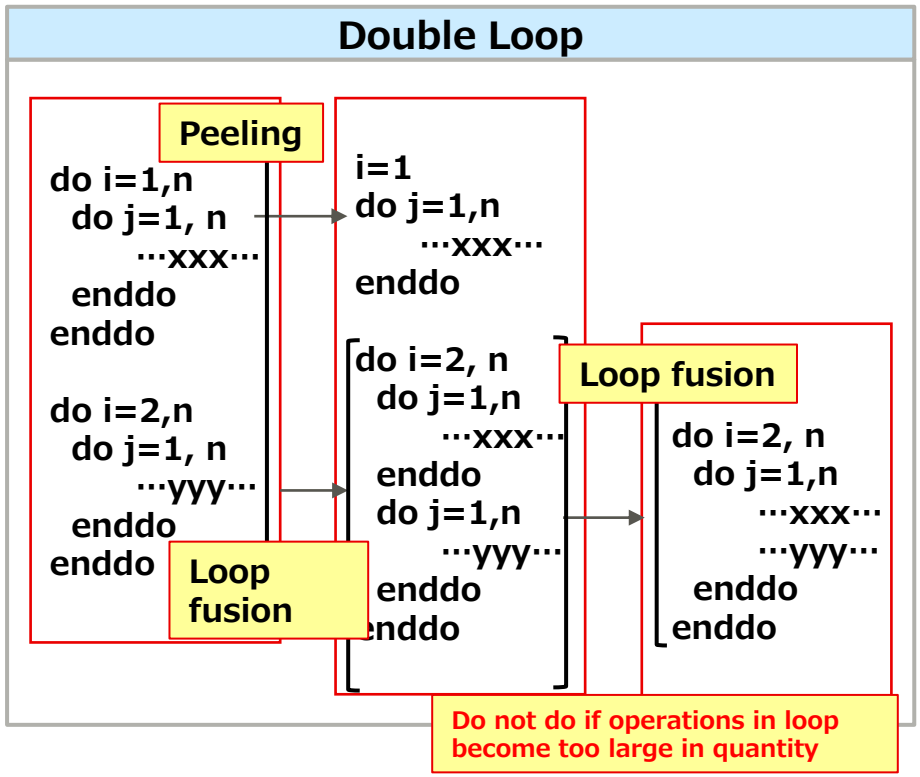
What is Loop Fusion?

FUJITSU

Loop fusion is a means to connect loops to achieve the following effects:

- **Data localization: To reuse arrays**
- **Higher parallelism at instruction level: To increase the number of instructions in a loop to increase parallelism at the instruction level**





- **Points**
- **The compiler automatically fuses loops that have the same loop length.**
- **Software pipelining may not be facilitated when a loop contains too many operations.**

Fortran Loop Fusion (Before Improvement)

FUJITSU

Array data is not fully cached and cannot be reused in Loop 2 because Loop 1 has a large number of iterations. Consequently, the "No instruction commit because memory cache is busy" event occurs many times.

Source Before Improvement

```
42   1 pp   v      do j=1,m-1
                   <<< Loop-information Start >>>
                   <<<  [OPTIMIZATION]
                   <<<    COLLAPSED
                   <<< Loop-information  End >>>
43   2  p            do i=1,n
44   2  p   v           s1 = s1 + a(i,j) * (s3 * b(i,j) + c(i,j) * (s2 + s3 * d(i,j) ))
45   2  p   v         enddo
46   1  p   v       enddo
47
                   <<< Loop-information Start >>>
                   <<<  [PARALLELIZATION]
                   <<<    Standard iteration count: 572
                   <<<  [OPTIMIZATION]
                   <<<    COLLAPSED
                   <<<    SIMD(VL: 8)
                   <<<    SOFTWARE PIPELINING(IPC: 2.30, ITR: 96, MVE: 2, POL: S)
                   <<<    PREFETCH(HARD) Expected by compiler :
                   <<<      b, a, d, c, e
                   <<< Loop-information  End >>>
48   1 pp   2v      do j=1,m
                   <<< Loop-information Start >>>
                   <<<  [OPTIMIZATION]
                   <<<    COLLAPSED
                   <<< Loop-information  End >>>
49   2  p   2         do i=1,n
50   2  p   2v          e(i,j) = s2 * (a(i,j) + b(i,j) * (s3 + c(i,j) * d(i,j)))
51   2  p   2v        enddo
52   1  p         enddo
```

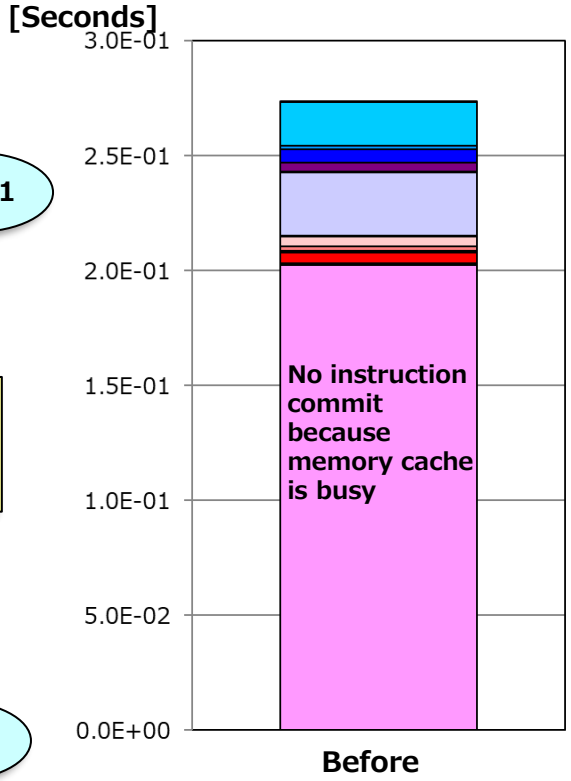
m = 50
n = 150,000
Array type: real*8

Loop 1

Total array data: Approx. 200 MB
Array data not fully cached

Array access causes cache miss

Loop 2



The L1 and L2 cache miss rates are 0.24, which is the theoretical value of stream access. However, misses occur in both Loops 1 and 2. This means Loop 2 cannot use the data cached in Loop 1.

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	8.57E+08	2.09E+08	0.24	0.64%	99.35%	0.01%	2.09E+08	0.24	0.66%	99.50%	0.00%

Fortran Loop Fusion (Source Tuning)

FUJITSU

Loop fusion increases cache efficiency. The result is improvement of the "No instruction commit because memory cache is busy" event.

Source After Improvement (Source Tuning)

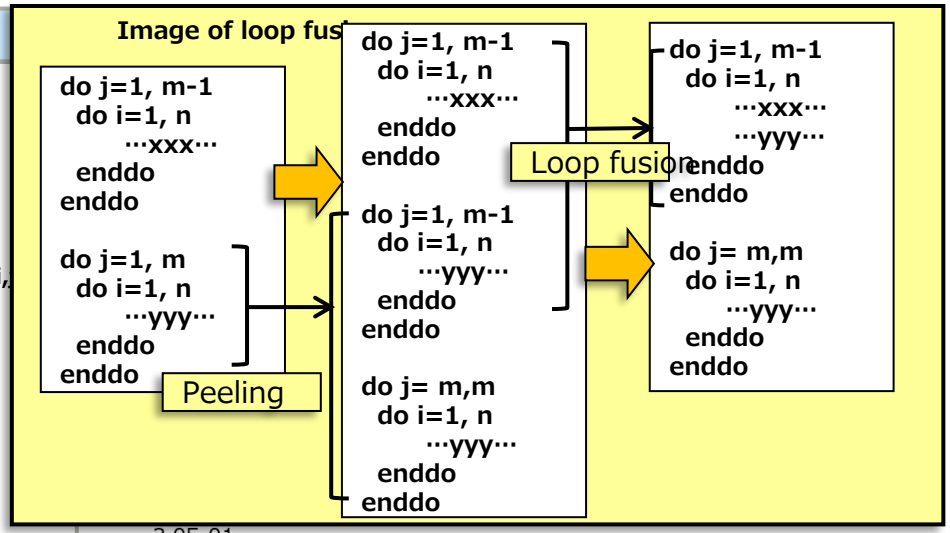
```
42   1 pp   v      do j=1,m-1
                   <<< Loop-information Start >>>
                   <<<  [OPTIMIZATION]
                   <<<    COLLAPSED
                   <<< Loop-information  End >>>
43   2  p            do i=1,n
44   2  p   v           s1 = s1 + a(i,j) * (s3 * b(i,j) + c(i,j) * (s2 + s3 * d(i,
45   2  p   v           e(i,j) = s2 * (a(i,j) + b(i,j) * (s3 + c(i,j) * d(i,j)))
46   2  p   v         enddo
47   1  p   v       enddo
 :
49   1  s         do j=m,m
                   <<< Loop-information Start >>>
                   <<<  [PARALLELIZATION]
                   <<<    Standard iteration count: 364
                   <<<  [OPTIMIZATION]
                   <<<    SIMD(VL: 8)
                   <<<    SOFTWARE PIPELINI                          S)
                   <<<    PREFETCH(HARD) Ex
                   <<<      b, a, d, c, e
                   <<< Loop-information  End >>>
50   2 pp  2v         do i=1,n
51   2  p  2v           e(i,j) = s2 * (a(i,j) + b(i,j) * (s3 + c(i,j) * d(i,j)))
52   2  p  2v         enddo
53   1  p          enddo
```

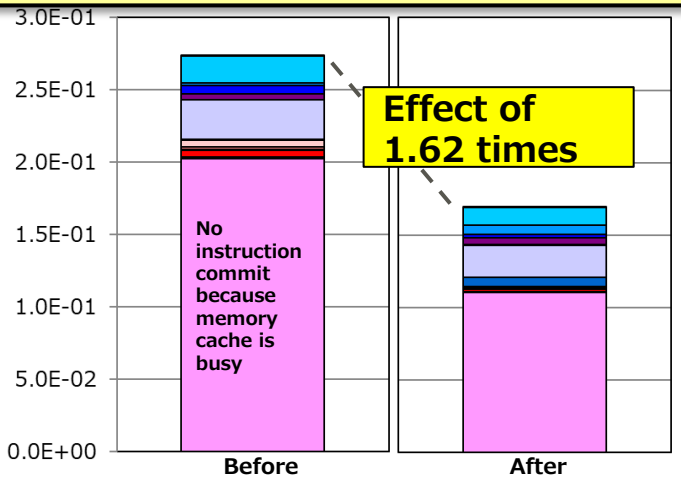
Loop fusion

Peeling

Array access causes cache hit

Cutting out (peeling) for loop fusion





Number of L1D and L2 misses reduced significantly

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	8.57E+08	2.09E+08	0.24	0.64%	99.35%	0.01%	2.09E+08	0.24	0.66%	99.50%	0.00%
After	0.00	4.92E+08	1.18E+08	0.24	0.36%	99.63%	0.00%	1.17E+08	0.24	0.41%	99.75%	0.00%

C/C++ Loop Fusion (Before Improvement)

FUJITSU

Array data is not fully cached and cannot be reused in Loop 2 because Loop 1 has a large number of iterations. Consequently, the "No instruction commit because memory cache is busy" event occurs many times.

Source Before Improvement

```
39          #pragma omp parallel for
40   p        for (j=0;j<m-1;j++) {
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
41   p    8v     for (i=0;i<n;i++) {
42   p    8v       *s1 = *s1 + a[j][i] * (*s3 * b[j][i] + c[j][i] * (*s2 + *s3 * d[j][i] ));
43   p    8v     }
44   p        }
45
46          #pragma omp parallel for
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
47   p        for (j=0;j<m;j++) {
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC:
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
48   p    2v     for (i=0;i<n;i++) {
49   p    2v       e[j][i] = *s2 * (a[j][i] + b[j][i]* (*s3 + c[j][i] * d[j][i] ));
50   p    2v     }
51   p        }
```

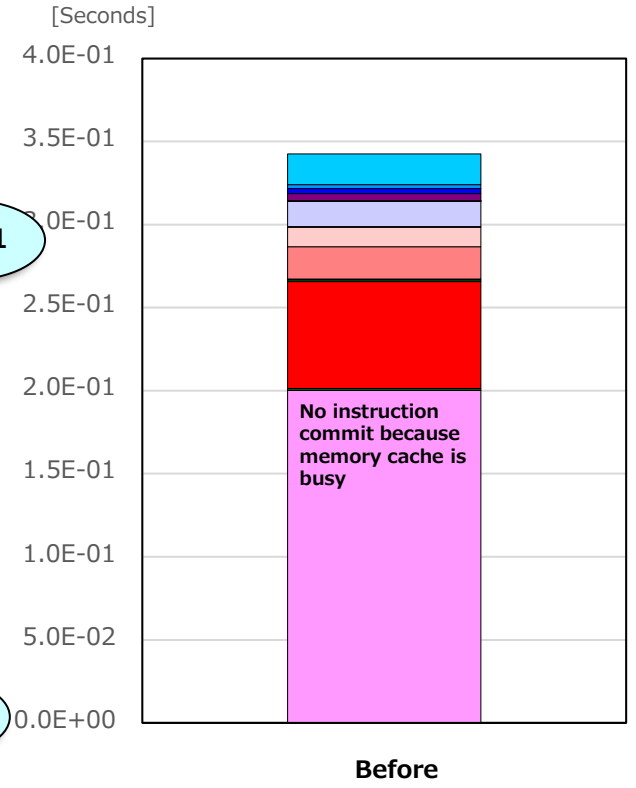
m = 50
n = 150000
Array type : double

Loop 1

Total array data: Approx. 200 MB
Array data not fully cached

Array access causes cache miss

Loop 2



The L1 and L2 cache miss rates are 0.21, which is the theoretical value of stream access. However, misses occur in both Loops 1 and 2. This means Loop 2 cannot use the data cached in Loop 1.

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	9.87E+08	2.10E+08	0.21	9.82%	90.20%	-0.02%	2.10E+08	0.21	5.83%	95.03%	0.00%

C/C++ Loop Fusion (Source Tuning)

FUJITSU

Loop fusion increases cache efficiency. The result is improvement of the "No instruction commit because memory cache is busy" event.

Source After Improvement (Source Tuning)

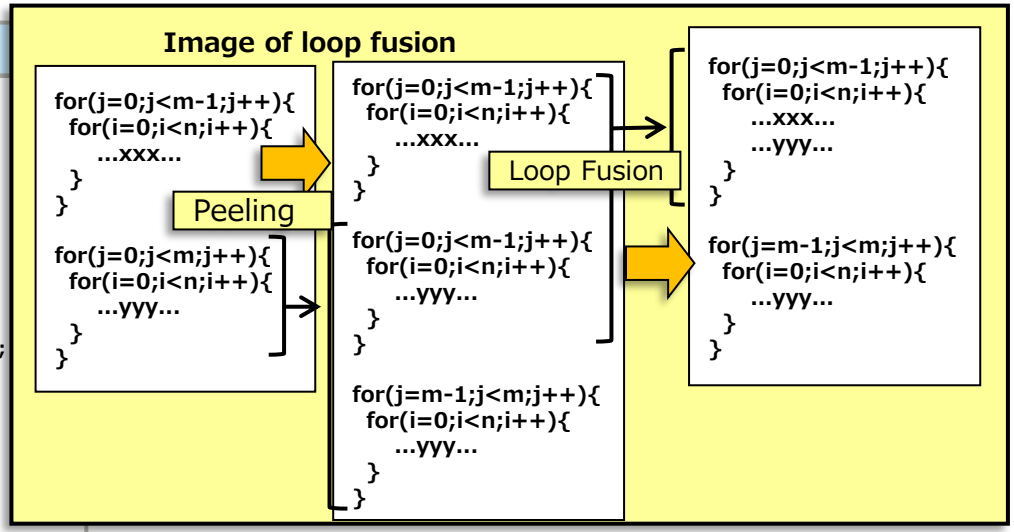
```
40  p        for (j=0;j<m-1;j++) {
             <<< Loop-information Start >>>
             <<<  [OPTIMIZATION]
             <<<    SIMD(VL: 8)
             <<<    SOFTWARE PIPELINING(IPC: 1.98, ITR: 192, MVE: 2, POL: S)
             <<<    PREFETCH(HARD) Expected by compiler :
             <<<    ...
             <<< Loop-information  End >>>
41  p   8v     for (i=0;i<n;i++) {
42  p   8v       *s1 = *s1 + a[j][i] * (*s3 * b[j][i] + c[j][i] * (*s2 + *s3 * d[j][i] ));
43  p   8v       e[j][i] = *s2 * (a[j][i] + b[j][i]* (*s3 + c[j][i] * d[j][i] ));
44  p   8v     }
45  p        }
...
48  p        for (j=m-1;j<m;j++) {
             <<< Loop-information Start >>>
             <<<  [OPTIMIZATION]
             <<<    SIMD(VL: 8)
             <<<    SOFTWARE PIPELINING(IPC: 3.83, ITR: 176, MVE: 3, POL: S)
             <<<    PREFETCH(HARD) Expected by compiler :
             <<<    ...
             <<< Loop-information  End >>>
49  p   2v     for (i=0;i<n;i++) {
50  p   2v       e[j][i] = *s2 * (a[j][i] + b[j][i]* (*s3 + c[j][i] * d[j][i] ));
51  p   2v     }
52  p        }
```

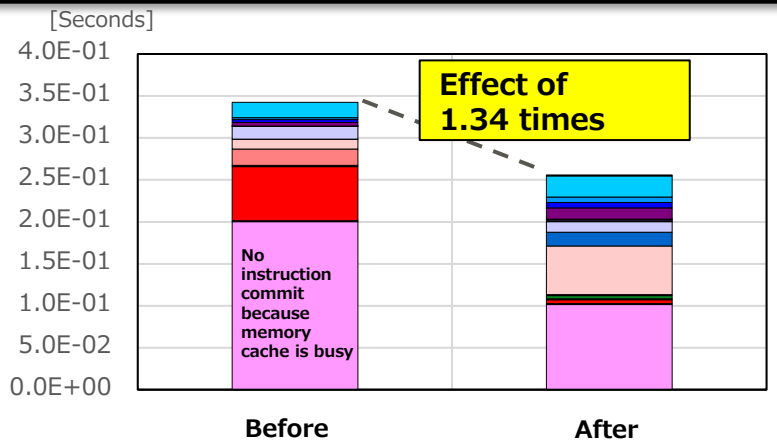
Loop Fusion

Peeling

Array access causes cache hit

Cutting out (peeling) for loop fusion





Number of L1D and L2 misses reduced significantly

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	9.87E+08	2.10E+08	0.21	9.82%	90.20%	-0.02%	2.10E+08	0.21	5.83%	95.03%	0.00%
After	0.00	1.94E+09	1.19E+08	0.06	2.55%	97.54%	-0.09%	1.18E+08	0.06	1.46%	98.73%	0.00%

FUJITSU

Array Merge (Indirect Access)

- What is Array Merge?
- Array Merge (Before Improvement)
- Effect of Array Merge (Source Tuning)

What is Array Merge?

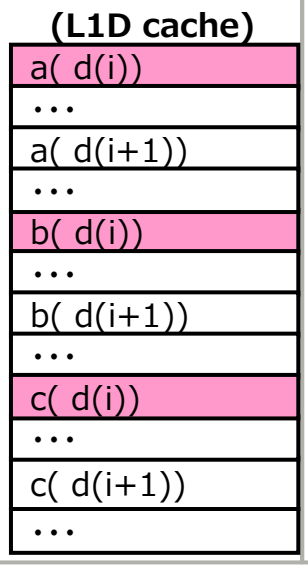
Array merge is a means to merge multiple arrays in the same loop into one only if they have a common access pattern. Data access becomes sequential, which increases cache efficiency.

Source Before Improvement	Source After Improvement

```
parameter(n=1000000)
  real*8  a(n), b(n),
  integer d(n+10)
    :
  do iter = 1, 100
    do i = 1 , n
      a(d(i)) = b(d(i))
              + scalar * c(d(i))
    enddo
  enddo
    :
```

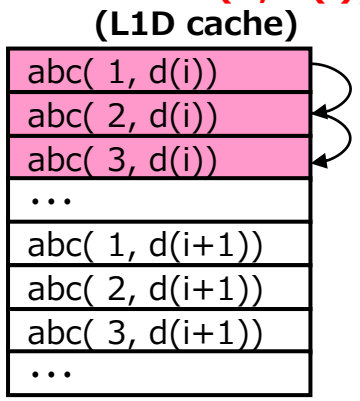
Array merge

```
parameter(n=1000000)
  real*8  abc(3, n)
  integer d(n+10)
    :
  do iter = 1, 100
    do i = 1 , n
      abc(1, d(i)) = abc(2, d(i))
                   + scalar * abc(3, d(i))
    enddo
  enddo
    :
```



Access to different cache lines

(when array d values are not sequential)



Access to same cache line

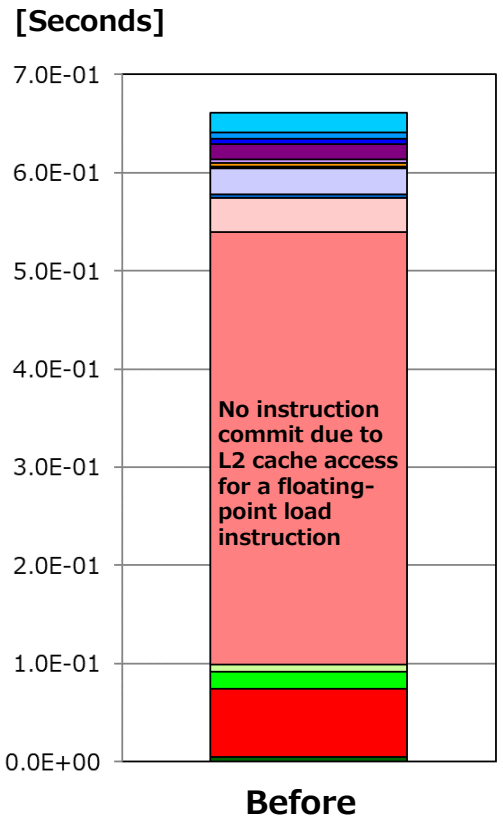
Fortran Array Merge (Before Improvement)

FUJITSU

Cache use efficiency is low (indirect access) because the L1D miss rate is high. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
 1                    parameter(n=2*1000*1000/8)
 2                    real*8  a(n),b(n),c(n),e(n),f(n),s
 3                    integer d(n)
 :
14    1   s             call sub(a,b,c,d,e,f,s,n)
 :
25                    subroutine sub(a,b,c,d,e,f, s, n)
26                    real*8  a(n),b(n),c(n),e(n),f(n),s
27                    integer d(n), ii
28
29                 !$omp parallel do schedule (static,96)
                   <<< Loop-information Start >>>
                   <<<  [OPTIMIZATION]
                   <<<    SIMD(VL: 8)
                   <<<    SOFTWARE PIPELINING(IPC: 1.07, ITR: 80, MVE: 3, POL: S)
                   <<<    PREFETCH(HARD) Expected by compiler :
                   <<<      d
                   <<< Loop-information  End >>>
30    1   p   2v           do i = 1 , n
31    1   p   2v             ii = d(i)
32    1   p   2v           a(ii) = s / (s + f(ii) / ( s + e(ii)  / ( b(ii) + s / c(ii)))
33    1   p   2v           enddo
34                    !$omp end parallel do
```



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	6.44E+08	1.27E+09	1.97	99.98%	0.01%	0.00%	4.82E+07	0.07	60.81%	50.33%	0.00%

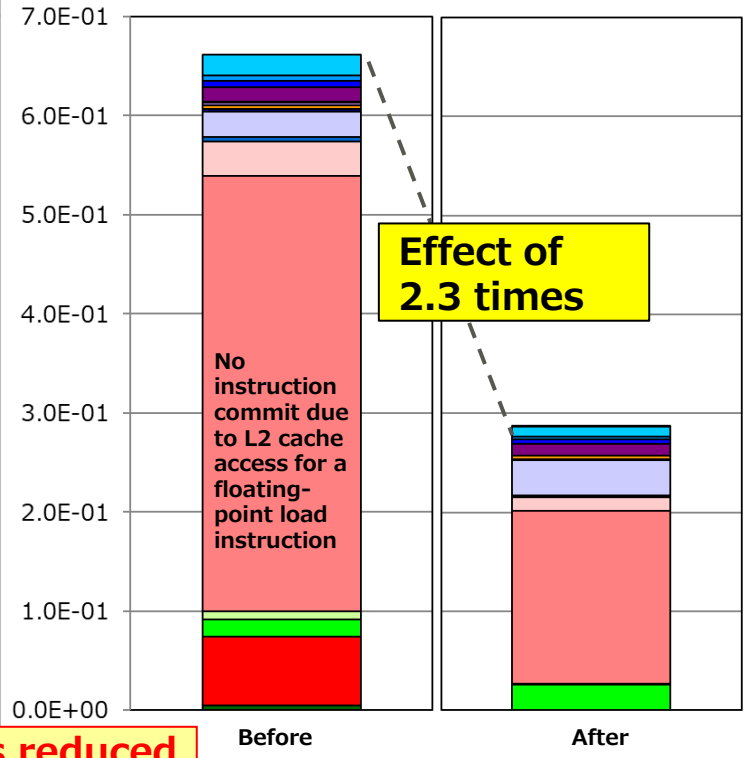
Fortran Effect of Array Merge (Source Tuning)

FUJITSU

Array merge increases cache efficiency by merging the list access arrays. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement (Source Tuning)

```
 1                    parameter(n=2*1000*1000/8)
 2                    real*8  abcef(5,n),s
 3                    integer d(n)
 :
14    1   s            call sub(abcef,d,s,n)
 :
24                    subroutine sub(abcef,d, s, n)
25                    real*8  abcef(5,n),s
26                    integer d(n), ii
27
28                 !$omp parallel do schedule (static,96)
                   <<< Loop-information Start >>>
                   <<<  [OPTIMIZATION]
                   <<<    SIMD(VL: 8)
                   <<<    SOFTWARE PIPELINING(IPC: 1.09, ITR: 80, MVE: 3, POL: S)
                   <<<    PREFETCH(HARD) Expected by compiler :
                   <<<      d
                   <<< Loop-information  End >>>
29    1   p   2v          do i = 1 , n
30    1   p   2v            ii = d(i)
31    1   p   2v          abcef(1,ii) = s / (s + abcef(5,ii) / ( s + abcef(4,ii)
32    1                 *  / ( abcef(2,ii) + s / abcef(3,ii))))
33    1   p   2v          enddo
34                 !$omp end parallel do
```



L1D miss rates reduced significantly

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	6.44E+08	1.27E+09	1.97	99.98%	0.01%	0.00%	4.82E+07	0.07	60.81%	50.33%	0.00%
After	0.00	3.19E+08	2.97E+08	0.93	99.68%	0.32%	0.00%	1.58E+04	0.00	63.74%	54.78%	0.00%

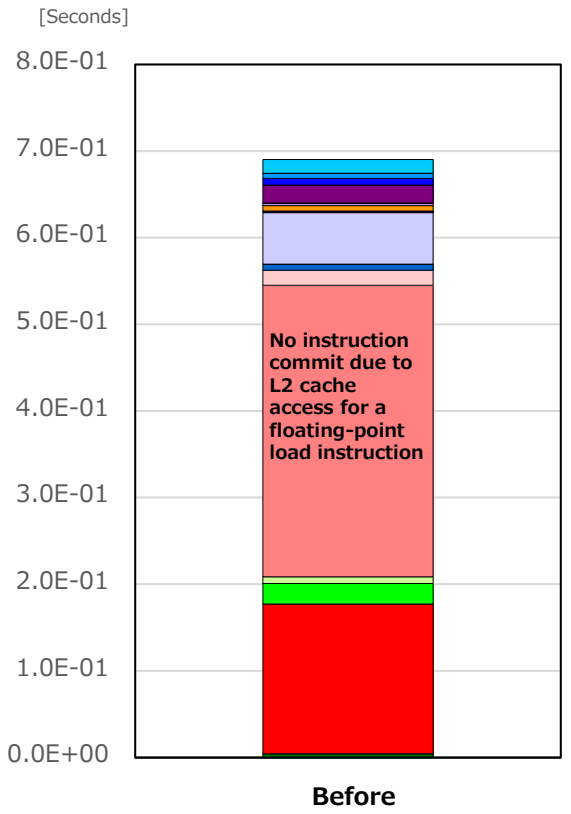
C/C++ Array Merge (Before Improvement)

Cache use efficiency is low (indirect access) because the L1D miss rate is high. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

Array declaration
double a[250000];
double b[250000];
double c[250000];
double e[250000];
double f[250000];

```
24 void sub(double (* restrict a),double (* restrict b),
           double (* restrict c),int (* restrict d),double (* restrict e),
           double (* restrict f),double s,int n) {
25           int i,ii;
26
27           #pragma omp parallel for schedule (static,96)
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 1.33, ITR: 112, MVE: 4, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
28   p    2v    for(i=0;i<n;i++) {
29   p    2v      ii = d[i];
30   p    2v      a[ii] = s / (s + f[ii] / ( s + e[ii]  / ( b[ii] + s / c[ii])));
31   p    2v    }
32           }
```



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	6.58E+08	1.27E+09	1.93	99.94%	0.06%	0.00%	4.24E+07	0.06	56.24%	56.98%	0.00%

Effect of Array Merge (Source Tuning)

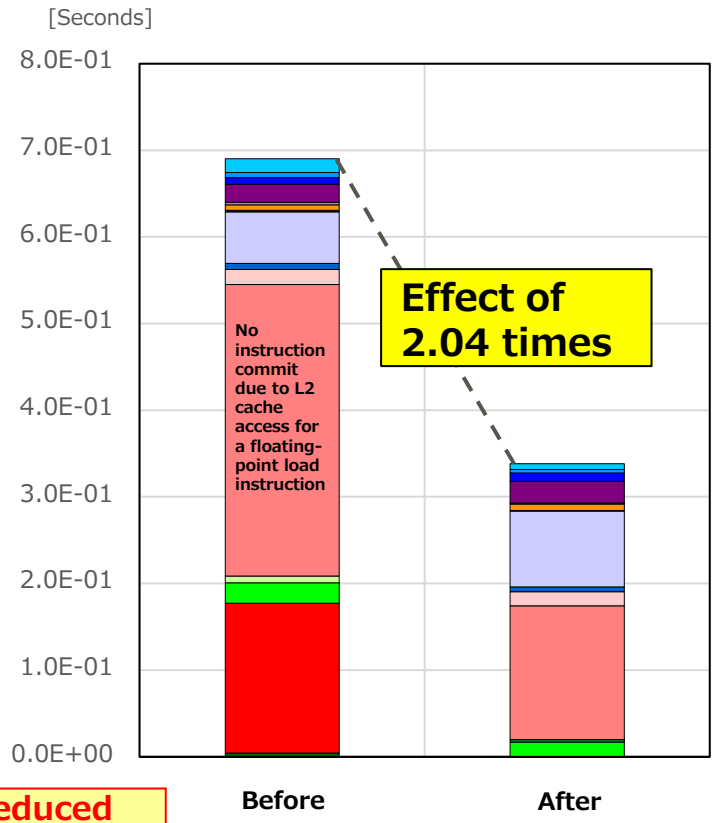
FUJITSU

Array merge increases cache efficiency by merging the list access arrays. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement (Source Tuning)

Array declaration
double a[250000];
double b[250000];
double c[250000];
double e[250000];
double f[250000];

```
24          void sub(double (* restrict abcef)[5],int (* restrict d),double s,int n) {
25            int i,ii;
26
27          #pragma omp parallel for schedule (static,96)
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 1.36, ITR: 112, MVE: 4, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
28   p     2v   for(i=0;i<n;i++) {
29   p     2v     ii = d[i];
30   p     2v     abcef[ii][0] = s / (s + abcef[ii][4] / ( s + abcef[ii][3]  /
                                  ( abcef[ii][1] + s / abcef[ii][2])));
31   p     2v   }
32           }
```



L1D miss rates reduced significantly

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	6.58E+08	1.27E+09	1.93	99.94%	0.06%	0.00%	4.24E+07	0.06	56.24%	56.98%	0.00%
After	0.00	3.67E+08	2.94E+08	0.80	99.69%	0.30%	0.00%	1.41E+04	0.00	76.45%	61.05%	0.00%

FUJITSU

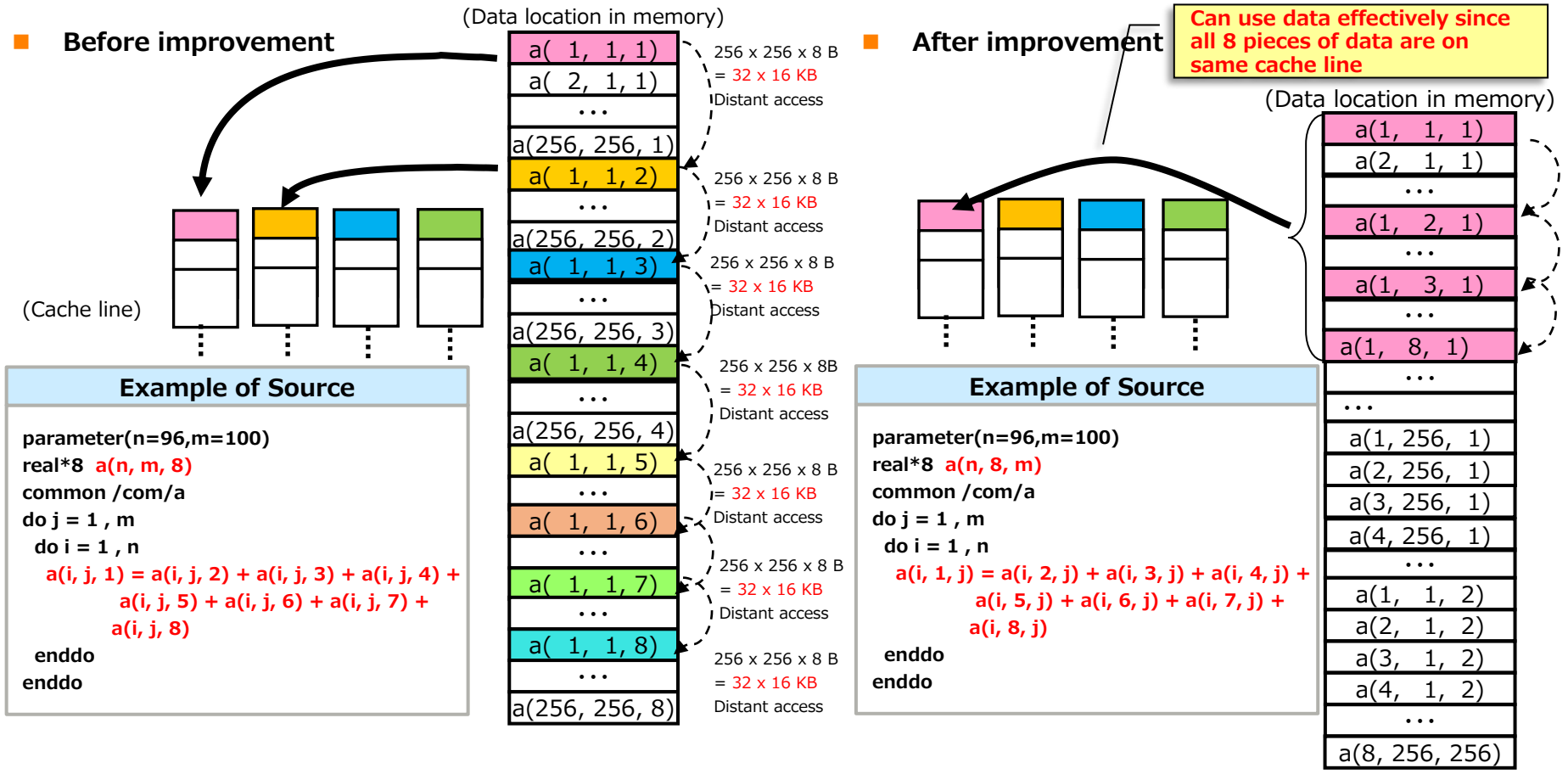
Array Dimension Shift

- What is Array Dimension Shift?
- Array Dimension Shift (Before Improvement)
- Effect of Array Dimension Shift (Source Tuning)
- Effect of Array Dimension Shift (Compiler Option Tuning)

What is Array Dimension Shift?

Array dimension shift is a tuning method that changes the access dimension of an array to improve cache utilization.

As shown in the following example, shifting the changed dimension inward can increase cache use efficiency.



Fortran Array Dimension Shift (Before Improvement)

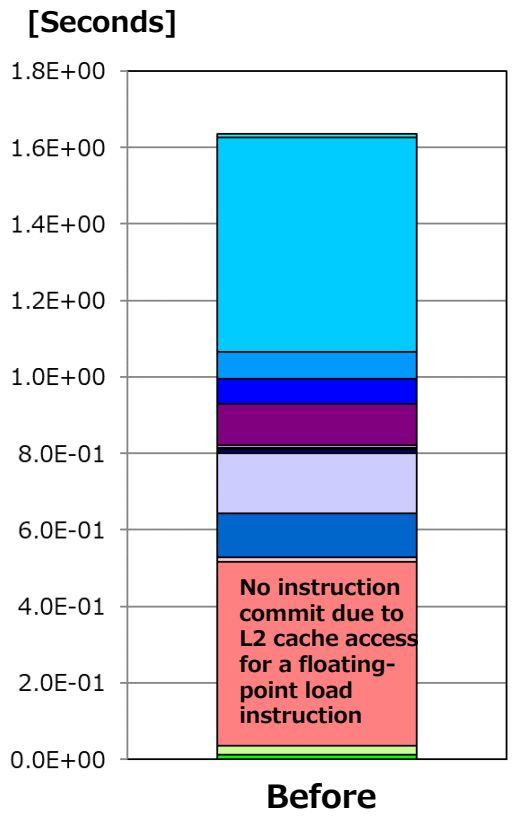
FUJITSU

Data locality for access in the innermost loop of Array a is low. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs.

Source Before Improvement

```
16                    real(8)::a(N,M,8)
 :
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a
                <<< Loop-information  End >>>
21    2  p              do j=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.87, ITR: 56,
                                                MVE: 2, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a
                <<< Loop-information  End >>>
22    3  p  v            do i=1,N
23    3  p  v             a(i,j,1)=a(i,j,2)+a(i,j,3)+a(i,j,4)+&
24    3                        a(i,j,5)+a(i,j,6)+a(i,j,7)+a(i,j,8)
25    3  p  v            enddo
26    2  p              enddo
```

N = 96
M = 100



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	2.12E+10	8.20E+08	0.04	78.62%	21.38%	0.00%	5.45E+03	0.00	87.63%	20.43%	0.00%

Fortran Effect of Array Dimension Shift (Source Tuning)

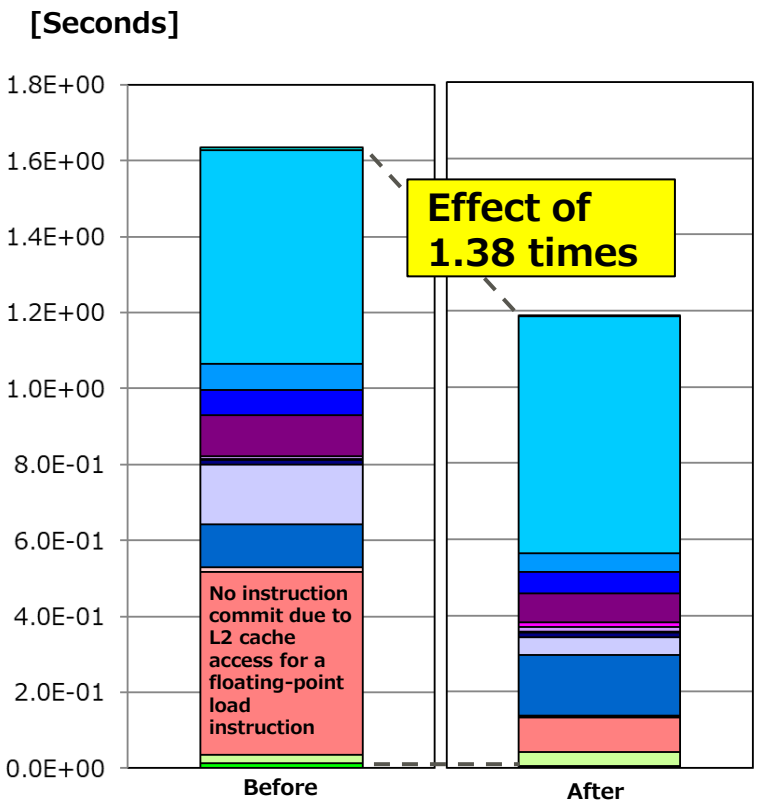
FUJITSU

Array dimension shift increases data locality. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement (Source Tuning)

```
16                real(8)::a(N,8,M)
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a
                <<< Loop-information  End >>>
21    2  p            do j=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.87, ITR: 56,
                                           MVE: 2, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a
                <<< Loop-information  End >>>
22    3  p  v          do i=1,N
23    3  p  v           a(i,1,j)=a(i,2,j)+a(i,3,j)+a(i,4,j)+&
24    3                     a(i,5,j)+a(i,6,j)+a(i,7,j)+a(i,8,j)
25    3  p  v          enddo
26    2  p            enddo
```

N = 96
M = 100



Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	2.12E+10	8.20E+08	0.04	78.62%	21.38%	0.00%	5.45E+03	0.00	87.63%	20.43%	0.00%
After	0.00	2.11E+10	7.39E+07	0.00	99.99%	0.01%	0.00%	4.12E+03	0.00	80.75%	32.43%	0.00%

C/C++ Array Dimension Shift (Before Improvement)

FUJITSU

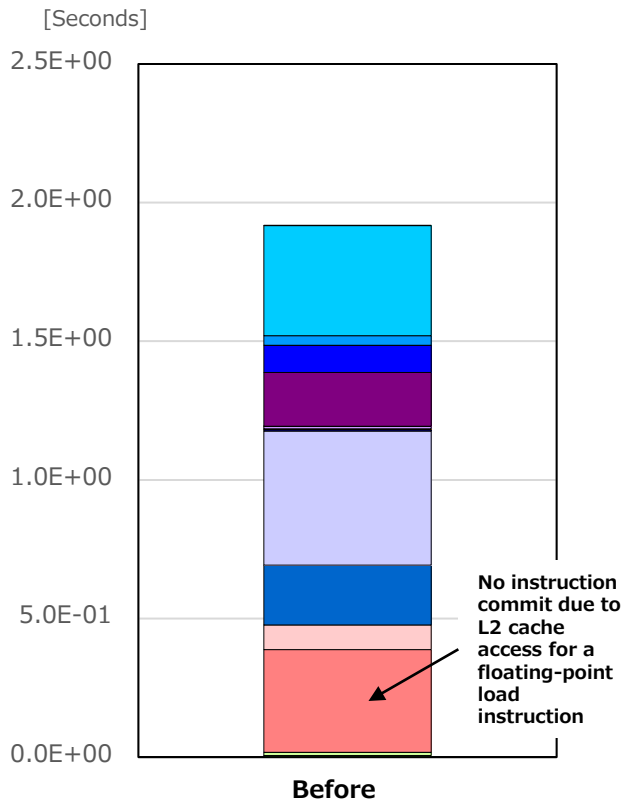
Data locality for access in the innermost loop of Array a is low. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs.

Source Before Improvement

```
36          void sub(int N,int M,int ITER, double (* restrict a)[M][N])
37          {
38            int i,j,k;
39          #pragma omp parallel private(i,j,k)
40          {
41            for(k=0; k<ITER; k++)
42            {
43          #pragma omp for nowait
          <<< Loop-information Start >>>
 :
          <<< Loop-information  End >>>
44  p         for(j=0; j<M; j++)
45  p         {
          <<< Loop-information Start >>>
 :
          <<< Loop-information  End >>>
46  p   v       for(i=0; i<N; i++)
47  p   v       {
48  p   v         a[0][j][i]=a[1][j][i]+a[2][j][i]+a[3][j][i]+
49                      a[4][j][i]+a[5][j][i]+a[6][j][i]+a[7][j][i];
50  p   v       }
51  p         }
52          }
53          }
55            return;
56          }
```

N=96
M=100

Array declaration
double a[8][M][N];



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	2.18E+10	8.20E+08	0.04	82.13%	17.87%	0.00%	1.49E+04	0.00	81.54%	38.68%	0.00%

Effect of Array Dimension Shift (Source Tuning)

FUJITSU

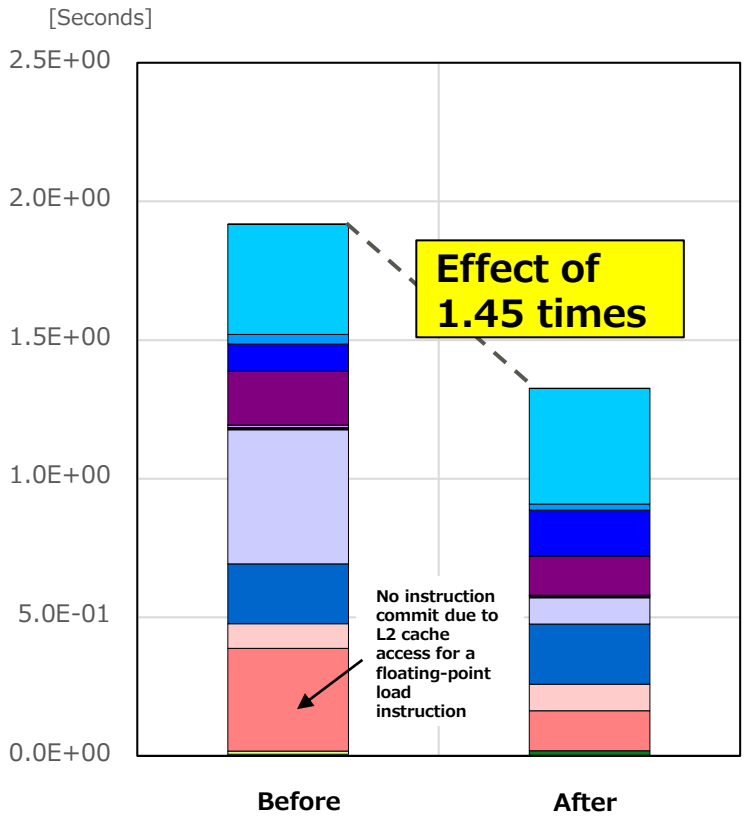
Array dimension shift increases data locality. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement (Source Tuning)

```
36          void sub(int N,int M,int ITER, double (* restrict a)[8][N]) {
38            int i,j,k;
39          #pragma omp parallel private(i,j,k)
40          {
41            for(k=0; k<ITER; k++) {
43          #pragma omp for nowait
          <<< Loop-information Start >>>
 :
          <<< Loop-information  End >>>
44  p         for(j=0; j<M; j++) {
          <<< Loop-information Start >>>
 :
          <<< Loop-information  End >>>
46  p   v     for(i=0; i<N; i++) {
48  p   v       a[j][0][i]=a[j][1][i]+a[j][2][i]+a[j][3][i]+
49                    a[j][4][i]+a[j][5][i]+a[j][6][i]+a[j][7][i];
50  p   v     }
51  p       }
52          }
53          }
55            return;
56          }
```

N=96
M=100

Array declaration
double a[8][M][N];



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	2.18E+10	8.20E+08	0.04	82.13%	17.87%	0.00%	1.49E+04	0.00	81.54%	38.68%	0.00%
After	0.00	2.09E+10	8.63E+07	0.00	99.97%	0.02%	0.00%	1.56E+04	0.00	86.83%	39.15%	0.00%

Effect of Array Dimension Shift (Compiler Option Tuning)

FUJITSU

You can obtain an effect equivalent to that of source tuning by specifying the following compiler options (Fortran-specific).

Compiler Option	Functional Description
-Karray_subscript	Specifies a dimensional shift in an allocatable array of 4 or more dimensions or in an array of 4 or more dimensions with 10 or fewer elements in the last dimension and 100 or more elements in the other dimensions. -Karray_subscript_element=100, -Karray_subscript_elementlast=10, and -Karray_subscript_rank=4 too are enabled at the same time.
-Karray_subscript_element=N (2≦*N*≦2,147,483,647)	Specifies *N* or more as the number of elements in dimensions other than the last dimension in the array undergoing the dimensional shift. This option is valid when the -Karray_subscript option is enabled. However, this option is not valid for allocatable arrays.
-Karray_subscript_elementlast=N (2≦*N*≦2,147,483,647)	Specifies *N* or less as the number of elements in the last dimension in the array undergoing the dimensional shift. This option is valid when the -Karray_subscript option is enabled. However, this option is not valid for allocatable arrays.
-Karray_subscript_rank=*N* (2≦*N*≦30)	Specifies *N* or more as the number of dimensions in the array undergoing the dimensional shift. This option is valid when the -Karray_subscript option is enabled.

■ **Use example (for source before improvement)**

```
$frtpx –Kfast,parallel sample.f90
  –Karray_subscript,array_subscript_rank=2,array_subscript_element=2
```

■ **Note**

- **These options must be specified in all source code using the target array.**
- **The effect of the shift varies depending on the program.**
- **If not used correctly, computational results may vary.**

FUJITSU

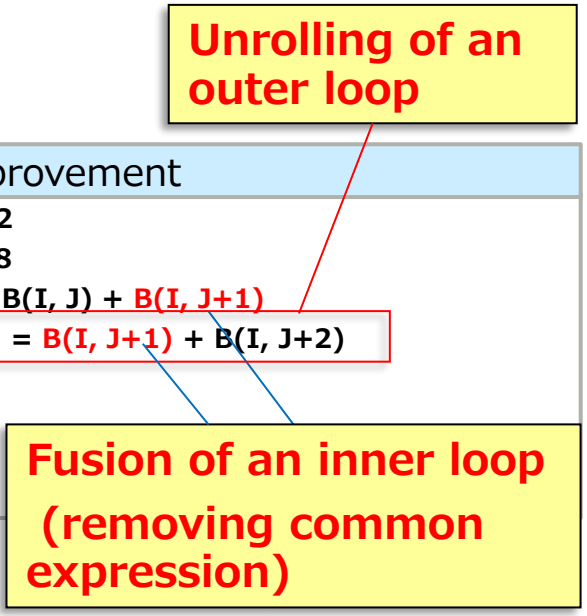
Unroll-and-Jam

- What is Unroll-and-Jam?
- Unroll-and-Jam (Before Improvement)
- Effect of Unroll-and-Jam (Optimization Control Line Tuning)

What is Unroll-and-Jam?

Unroll-and-jam is the optimization to unroll an outer loop of a nested loop n times and jam the unrolled statements into the inner loop as loop fusion.

Source Before Improvement	Source After Improvement
!OCL UNROLL_AND_JAM_FORCE(2) DO J=1, 128 DO I=1, 128 A(I, J) = B(I, J) + B(I, J+1) … END DO END DO :	DO J=1, 128, 2 DO I=1, 128 A(I, J) = B(I, J) + B(I, J+1) A(I, J+1) = B(I, J+1) + B(I, J+2) … END DO END DO



Unroll-and-jam promotes removing common expression and improves the execution performance. The increase of data stream and change of the order of data access may decrease the cache efficiency and the execution performance.

What is Unroll-and-Jam?

FUJITSU

Specify the following optimization control lines.

Optimization Specifier (Fortran)	Meaning	Optimization Control Line Specifiable?			
		By Program	By DO Loop	By Statement	By Array Assignment Statement
UNROLL_AND_JAM[(*n*)]	Enables unroll-and-jam for loops to be as effectively optimized as determined for the loops. *n* is a decimal number (2 to 100) that represents the number of unrolls (multiplicity).	Yes	Yes	No	No
UNROLL_AND_JAM_FORCE[(*n*)]	Enables unroll-and jam. *n* is a decimal number (2 to 100) that represents the number of unrolls (multiplicity).	No	Yes	No	No
NOUNROLL_AND_JAM	Disables unroll-and-jam.	Yes	Yes	No	No

Optimization Specifier (C/C++)	Meaning	Optimization Control Line Specifiable?			
		global	procedure	loop	statement
unroll_and_jam[(*n*)]	Enables unroll-and-jam for loops to be as effectively optimized as determined for the loops. *n* is a decimal number (2 to 100) that represents the number of unrolls (multiplicity).	Yes	Yes	Yes	No
unrool_and_jam_force[(*n*)]	Enables unroll-and jam. *n* is a decimal number (2 to 100) that represents the number of unrolls (multiplicity).	No	No	Yes	No
nounroll_and_jam	Disables unroll-and-jam.	Yes	Yes	Yes	No

- **Note**
- The UNROLL_AND_JAM specifier does not result in optimization in cases where no effect can be expected from optimization or there is data dependency across iterations.
- If the UNROLL_AND_JAM_FORCE specifier is mistakenly specified (there is data dependency across iterations), the execution results are not guaranteed.
- The innermost loop is not subject to unroll-and-jam.
- If the number of iterations of the innermost loop is small, cache use efficiency and execution performance may drop, depending on an increase in the number of data streams or changes in the data access sequence.
- Prefetch may compensate for an increase in cache misses, improving the situation.

What is Unroll-and-Jam?

FUJITSU

You can obtain an effect equivalent to that of optimization control line tuning by specifying the following compiler option.

Compiler Option	Functional Description
-K{ unroll_and_jam[=N] | nounroll_and_jam } 2 ≦ *N* ≦ 100	Specifies whether or not to perform unroll-and-jam optimization. You can specify a value from 2 to 100 in *N* to set the upper limit on the number of loop unrolls. If *N* is not specified, the compiler automatically decides the best value. The default is -Knounroll_and_jam. Applying unroll-and-jam facilitates the elimination of common expressions and may raise execution performance. However, an increase in the number of data streams or a change in the access sequence may decrease cache use efficiency, resulting in a decrease in execution performance. In addition, the impact of unroll-and-jam optimization on execution performance varies from loop to loop. Therefore, we recommend not applying this optimization with the -Kunroll_and_jam[=*N*] option to an entire program but instead applying it with the optimization specifier UNROLL_AND_JAM or UNROLL_AND_JAM_FORCE to individual loops.

■ **Use example (for source before improvement)**

```
$ frtpx -Kfast,parallel sample.f90 -Kunroll_and_jam
$ fccpx -Kfast,parallel sample.c -Kunroll_and_jam
```

◆**Note**

Unroll-and jam optimization is not available in Clang Mode.

Fortran Unroll-and-Jam (Before Improvement)

FUJITSU

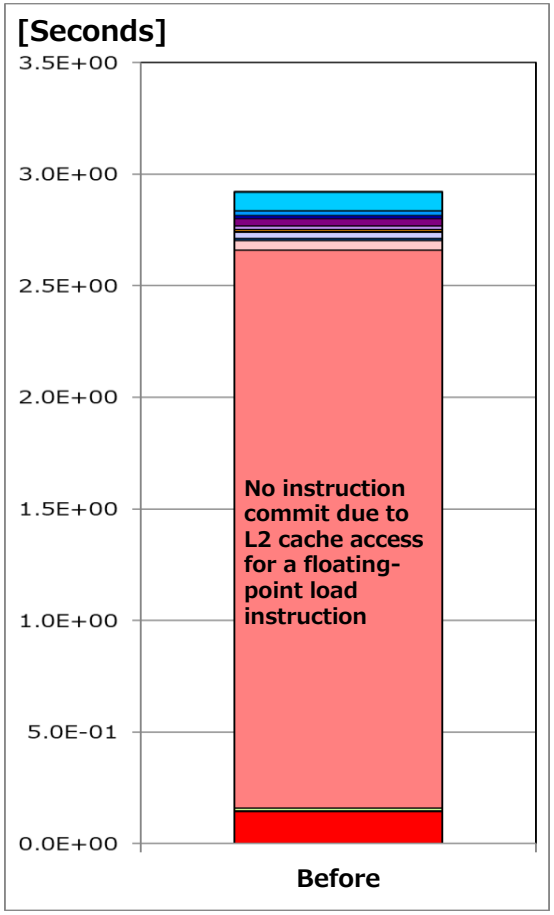
Cache use efficiency is low because the L1D miss rate is high. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
11            DATA_TYPE,dimension(IMAX,JMAX,KMAX)::a
12            DATA_TYPE,dimension(JMAX,KMAX)::b
13            DATA_TYPE,dimension(JMAX,IMAX)::c
 :
15 1  pp        do k=1, KMAX
16 1
17 2   p         do j=1, JMAX - 3
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 1.85, ITR: 80, MVE: 2, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      a
            <<< Loop-information  End >>>
18 3   p   2v        do i =1, IMAX
19 3   p   2v          a(i,j,k) = a(i,j+1,k) + a(i,j+2,k) + a(i,j+3,k) &
20 3                       + (b(j+2,k) / c(j,i))
21 3   p   2v        end do
22 2   p           end do
23 1   p         end do
```

```
#define DATA_TYPE real(kind=8)
#define IMAX 512
#define JMAX 512
#define KMAX 128
```

Low cache use efficiency since Array c has stride access pattern



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)
Before	0.00	3.39E+09	3.82E+09	1.13	99.50%	0.50%	0.00%	1.15E+08	0.03	46.67%

Fortran Effect of Unroll-and-Jam (Optimization Control Line Tuning)

FUJITSU

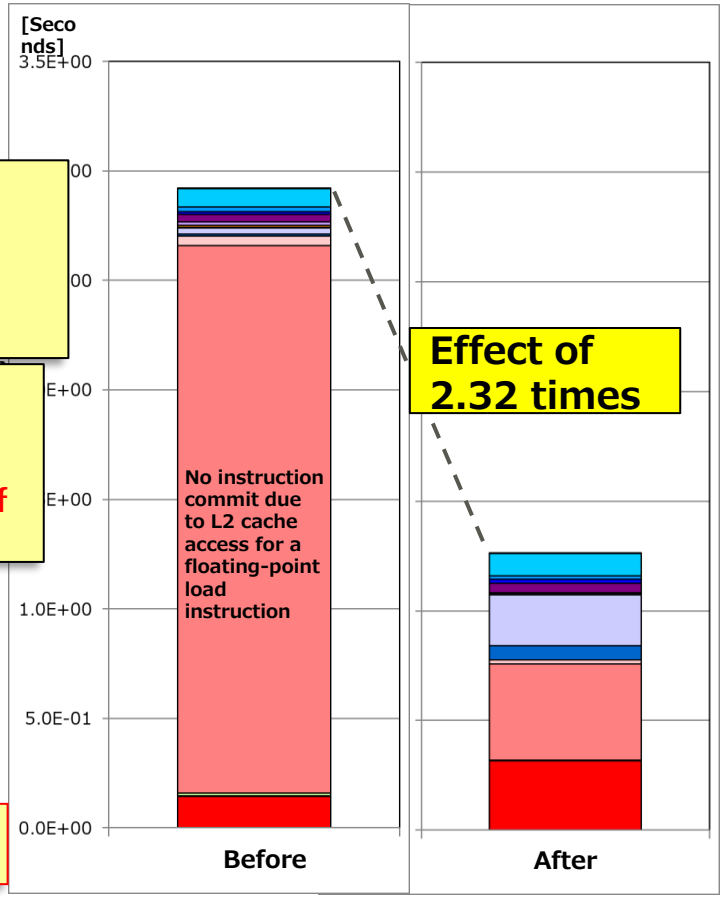
Unroll-and-jam increases cache use efficiency, reducing the number of the L1D misses. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement (Optimization Control Line Tuning)

```
11                DATA_TYPE,dimension(IMAX,JMAX,KMAX)::a
12                DATA_TYPE,dimension(JMAX,KMAX)::b
13                DATA_TYPE,dimension(JMAX,IMAX)::c
 :
15    1  pp           do k=1, KMAX
16    1            !ocl unroll_and_jam_force(8)
17    2  p            do j=1, JMAX - 3
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELINING
                 <<<    PREFETCH(HARD) Expe
                 <<<      a
                 <<<    PREFETCH(SOFT) : 32
                 <<<     SEQUENTIAL : 32
                 <<<      a: 32
                 <<<    SPILLS :
                 <<<      GENERAL   : SPILL 0  FILL 4
                 <<<      SIMD&FP   : SPILL 0  FILL 0
                 <<<      SCALABLE  : SPILL 0  FILL 0
                 <<<      PREDICATE : SPILL 0  FILL 0
                 <<< Loop-information  End >>>
18    3  p  2v         do i =1, IMAX
19    3  p  2v          a(i,j,k) = a(i,j+1,k) + a(i,j+2,k) + a(i,j+3,k) &
20    3                        + (b(j+2,k) / c(j,i))
21    3  p  2v         end do
22    2  p            end do
23    1  p          end do
```

```
#define DATA_TYPE
real(kind=8)
#define IMAX 512
#define JMAX 512
#define KMAX 128
```

Unrolling of an outer loop reduced instructions by eliminating the common expressions of Array a and increased the cache use efficiency of Arrays a and c.



L1D misses reduced significantly

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)
Before	0.00	3.39E+09	3.82E+09	1.13	99.50%	0.50%	0.00%	1.15E+08	0.03	46.67%
After	0.00	2.68E+09	7.37E+08	0.27	89.34%	2.55%	8.12%	1.15E+08	0.04	44.44%

C/C++ Unroll-and-Jam (Before Improvement)

FUJITSU

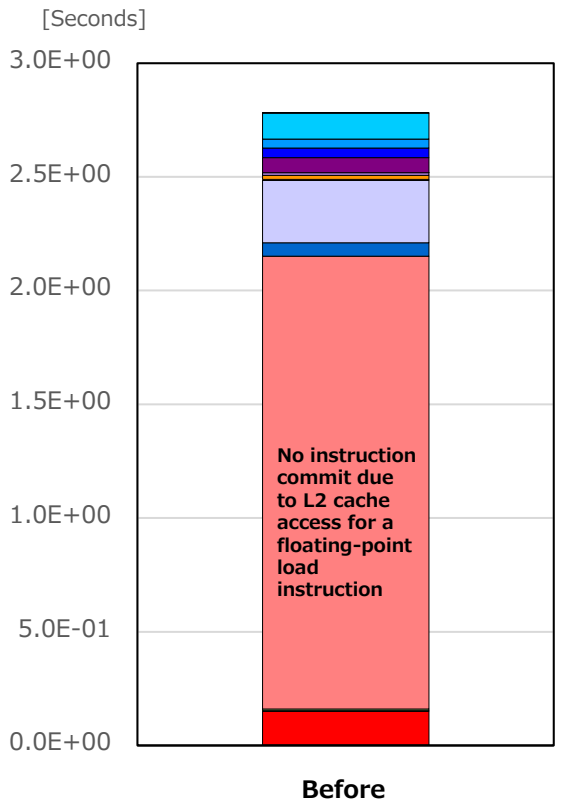
Cache use efficiency is low because the L1D miss rate is high. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
69          #pragma omp parallel for
70   p        for (k=0;k<KMAX;k++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by comp
          <<<      (unknown)
          <<< Loop-information  End >>>
71   p          for (j=0;j<JMAX-3;j++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.09, ITR: 80, MVE: 2, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
72   p    2v       for (i=0;i<IMAX;i++){
73   p    2v        a[k][j][i] = a[k][j+1][i] + a[k][j+2][i] + a[k][j+3][i]
74                          + (b[k][j+2] / c[i][j]);
75   p    2v       }
76   p           }
77   p         }
```

```
#define IMAX 512
#define JMAX 512
#define KMAX 128

Array declaration
double
a[KMAX][JMAX][IMAX],
b[KMAX][JMAX],
c[IMAX][JMAX];
```

Low cache use efficiency since Array c has stride access pattern



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)
Before	0.00	3.50E+09	3.68E+09	1.05	99.73%	0.26%	0.00%	1.16E+08	0.03	59.11%

C/C++ Effect of Unroll-and-Jam (Optimization Control Line Tuning)

FUJITSU

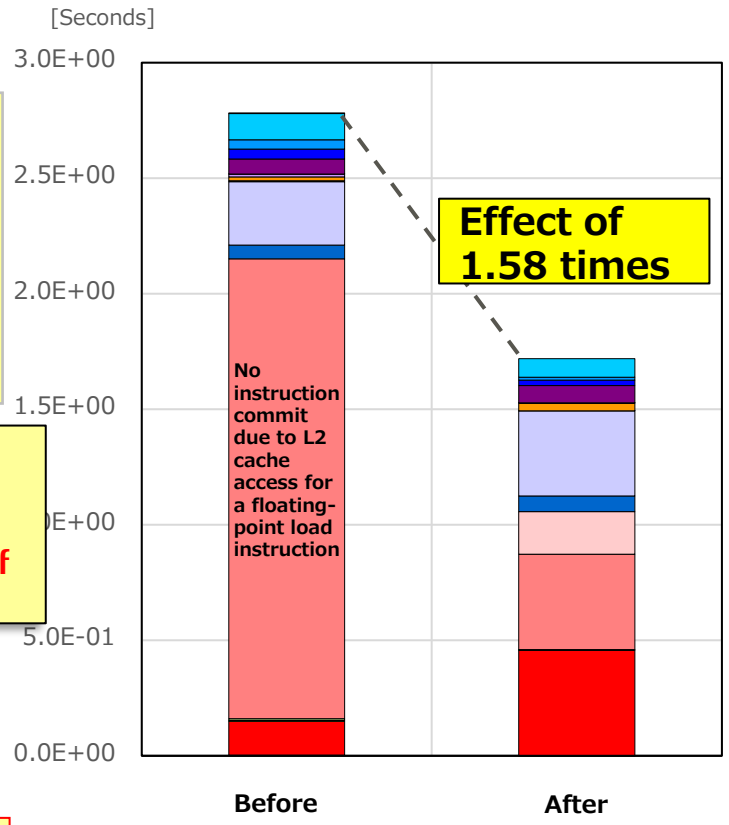
Unroll-and-jam increases cache use efficiency, reducing the number of the L1D misses. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement (Optimization Control Line Tuning)

```
74            for (k=0;k<KMAX;k++){
75          #pragma loop unroll_and_jam_force 8
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by com
          <<<      (unknown)
          <<< Loop-information  End >>>
76            for (j=0;j<JMAX-3;j++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SOFTWARE PIPELINING(IPC: 0.53,
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<<    PREFETCH(SOFT) : 32
          <<<     SEQUENTIAL : 32
          <<<      (unknown): 32
          <<<    SPILLS :
          <<<      GENERAL   : SPILL 0  FILL
          <<<      SIMD&FP   : SPILL 0  FILL
          <<<      SCALABLE  : SPILL 0  FILL 0
          <<<      PREDICATE : SPILL 0  FILL 0
          <<< Loop-information  End >>>
77        2v      for (i=0;i<IMAX;i++){
78        2v        a[k][j][i] = a[k][j+1][i] + a[k][j+2][i] + a[k][j+3][i]
79                          + (b[k][j+2] / c[i][j]);
80        2v      }
81            }
82          }
```

```
#define IMAX 512
#define JMAX 512
#define KMAX 128

Array declaration
double
a[KMAX][JMAX][IMAX],
b[KMAX][JMAX],
c[IMAX][JMAX];
```

Unrolling of an outer loop reduced instructions by eliminating the common expressions of Array a and increased the cache use efficiency of Arrays a and c.



L1D misses reduced significantly

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)
Before	0.00	3.50E+09	3.68E+09	1.05	99.73%	0.26%	0.00%	1.16E+08	0.03	59.11%
After	0.00	2.42E+09	6.24E+08	0.26	82.30%	3.11%	14.60%	1.15E+08	0.05	39.17%

FUJITSU

Data Access Wait Time (Hidden Latency)

- Indirect Access Prefetch
- Using Software Prefetch to Access Non-Sequential Data

FUJITSU

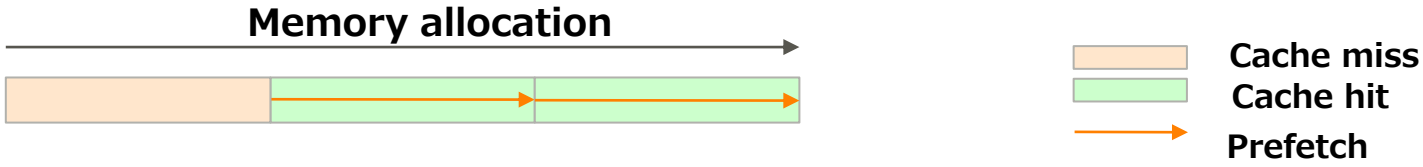
Indirect Access Prefetch

- What is Prefetch?
- Indirect Access Prefetch (Optimization Control Line)
- Effect of Indirect Access Prefetch (Compiler Option Tuning)
- Indirect Access Prefetch (Before Improvement)
- Effect of Indirect Access Prefetch (Optimization Control Line Tuning)

What is Prefetch?

FUJITSU

Prefetch is a mechanism that raises performance by loading data into the cache before the data is required by an executed instruction.

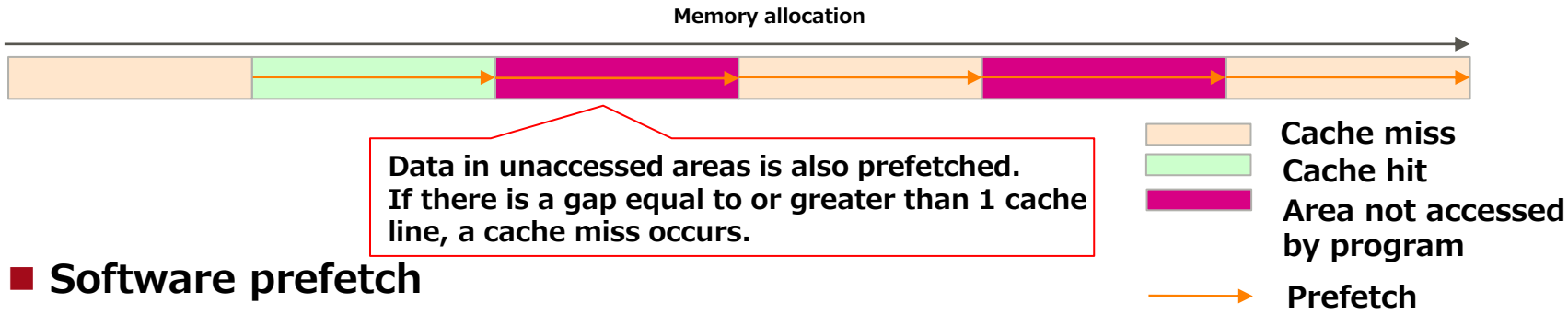


The two types of prefetch are hardware prefetch and software prefetch.

■ Hardware prefetch

Hardware prefetches data by predicting data access based on the regularity of memory access by programs.

The cache efficiency of a program may degrade significantly because data is also prefetched from areas not accessed by the program. In such cases, use software prefetch.



■ Software prefetch

Software (compiler) prefetches data by analyzing programs and generating a prefetch instruction.

Indirect Access Prefetch (Optimization Control Line)

FUJITSU

Specify the following optimization control line.

Optimization Specifier (Fortran)	Meaning	Optimization Control Line Specifiable?			
		By Program	By DO Loop	By Statement	By Array Assignment Statement
PREFETCH	Enables the automatic prefetch function of the compiler.	Yes	Yes	No	No

Optimization Specifier (C/C++)	Meaning	Optimization Control Line Specifiable?			
		global	procedure	loop	statement
prefetch	Enables the automatic prefetch function of the compiler.	Yes	Yes	Yes	No

◆**Supplementary information**

The prefetch optimization specifier is equivalent to specifying the following compiler option:

-Kprefetch_sequential,prefetch_stride,prefetch_indirect,prefetch_conditional, prefetch_cache_level=all

◆**Note**

Prefetch with the compiler option -Kprefetch_sequential, -Kprefetch_stride, -Kprefetch_indirect, or -Kprefetch_conditional enabled may degrade execution performance, depending on the cache efficiency of loops, whether they have any branches, and the complexity of subscripts.

Effect of Indirect Access Prefetch (Compiler Option Tuning)

FUJITSU

You can obtain an effect equivalent to that of optimization control line tuning by specifying the following compiler option.

Compiler Option	Functional Description
-Kprefetch_indirect	**Specifies whether or not to generate an object using a prefetch instruction for the array data that is used in a loop and accessed indirectly (list-accessed).** **This option is valid when the -O1 or higher option is enabled.** **The default is -Kprefetch_noindirect.**

■Use example (for source before improvement)

$ frtpx –Kfast,parallel sample.f90 -Kprefetch_indirect

$ fccpx –Kfast,parallel sample.c -Kprefetch_indirect

◆**Note**

Although data is prefetched, the intended effect may not be obtained depending on the cache efficiency of loops, whether they have any IF clauses, and the complexity of subscripts.

Indirect prefetch optimization is not available in Clang Mode.

Fortran Indirect Access Prefetch (Before Improvement)

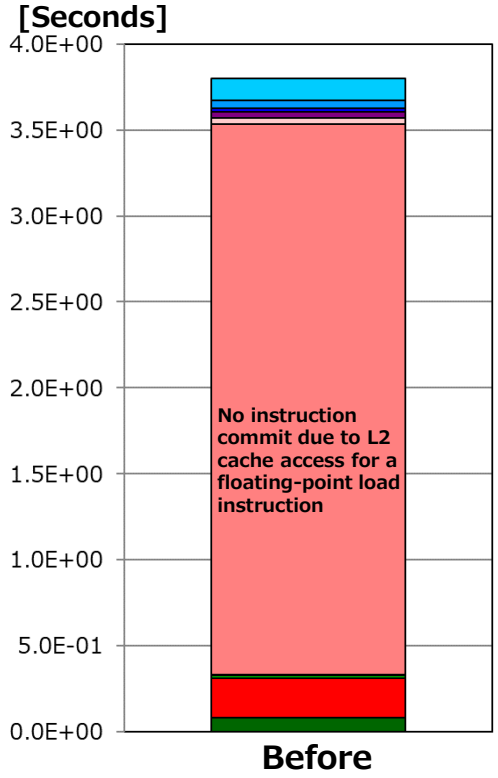
FUJITSU

Latency is apparent in memory access because the recommended option does not generate prefetch in cases of indirect access (list access). Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
        <<< Loop-information Start >>>
        <<<  [PARALLELIZATION]
        <<<    Standard iteration count: 572
        <<<  [OPTIMIZATION]
        <<<    SOFTWARE PIPELINING(IPC: 3.50, ITR: 18, MVE: 3, POL: S)
        <<<    PREFETCH(HARD) Expected by compiler :
        <<<      e, d, a
        <<< Loop-information  End >>>
52      1  pp   2            do i = 1 , n
53      1   p   2              a(i) = b(d(i)) + scalar * c(e(i))
54      1   p   2            enddo
```

Arrays b and c have indirect access patterns



The L1D and L2 miss dm rates are high, which means prefetch is not working. Performance may be higher because memory throughput is more than sufficient to raise performance.

	Memory throughput (GB/s)
Before	32.67

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	9.01E+09	3.77E+09	0.42	98.23%	1.77%	0.00%	4.29E+08	0.05	54.19%	75.86%	0.00%

Fortran Effect of Indirect Access Prefetch (Optimization Control Line Tuning)

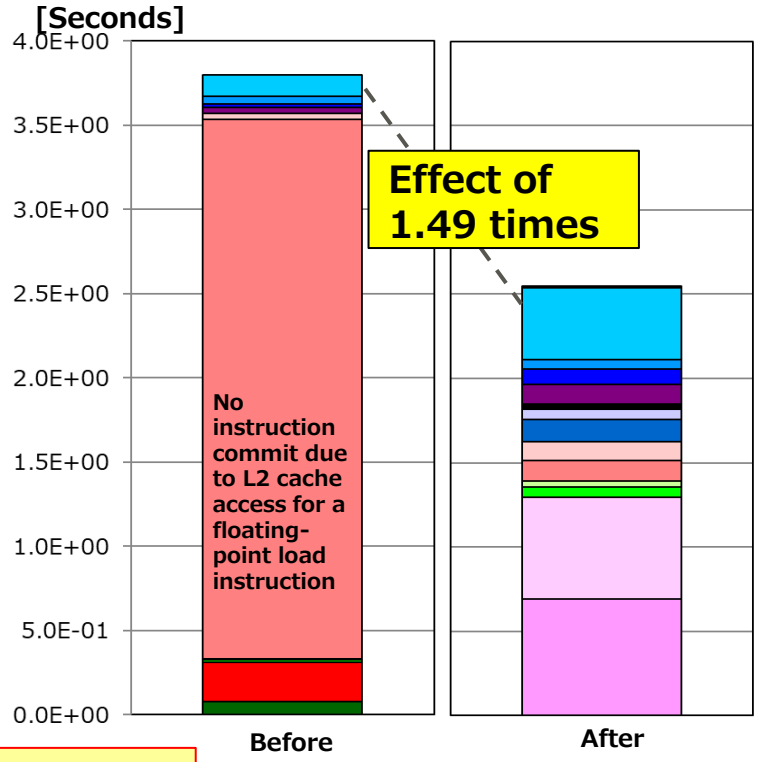
FUJITSU

Specify the prefetch specifier to generate prefetch for indirect access (list access). The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement

```
51              !ocl prefetch
          <<< Loop-information Start >>>
          <<<  [PARALLELIZATION]
          <<<    Standard iteration count: 572
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Ex
          <<<      e, d, a
          <<<    PREFETCH(SOFT) : 8
          <<<     INDIRECT : 8
          <<<      c: 4, b: 4
          <<< Loop-information  End >>>
52    1  pp   2            do i = 1 , n
53    1   p   2              a(i) = b(d(i)) + scalar * c(e(i))
54    1   p   2            enddo
```

Prefetch generated for indirect access (Arrays b and c)



The L1D and L2 miss dm rates were reduced by the generated prefetch instruction for indirect access (Arrays b and c).

	Memory throughput (GB/s)
Before	32.67
After	183.71

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	9.01E+09	3.77E+09	0.42	98.23%	1.77%	0.00%	4.29E+08	0.05	54.19%	75.86%	0.00%
After	0.00	1.66E+10	3.82E+09	0.23	2.94%	2.94%	94.12%	1.77E+09	0.11	2.02%	6.20%	91.78%

Indirect Access Prefetch (Before Improvement)

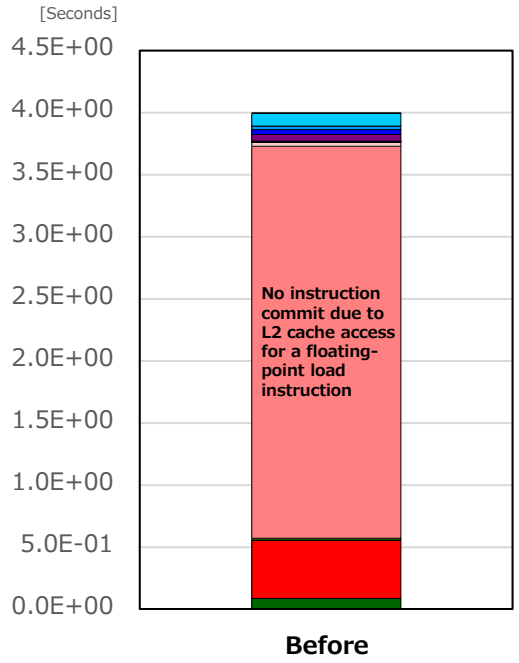
FUJITSU

Latency is apparent in memory access because the recommended option does not generate prefetch in cases of indirect access (list access). Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
53          void sub(double * restrict a, double * restrict b,
                     double * restrict c, int * restrict d, int * restrict e,
                     double scalar, int n){
54            int i;
56          #pragma omp parallel for
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SOFTWARE PIPELINING(IPC: 2.83, ITR: 12, MVE: 2, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
57   p    2    for (i = 0; i < n; i++){
58   p    2      a[i] = b[d[i]] + scalar * c[e[i]];
59   p    2    }
60          }
```

Arrays b and c have indirect access patterns



The L1D and L2 miss dm rates are high, which means prefetch is not working. Performance may be higher because memory throughput is more than sufficient to raise performance.

Statistics	Memory throughput (GB/s)
Before	31.24

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	8.65E+09	3.78E+09	0.44	98.01%	1.99%	0.00%	4.30E+08	0.05	47.96%	100.00%	0.00%

C/C++ Effect of Indirect Access Prefetch (Optimization Control Line Tuning)

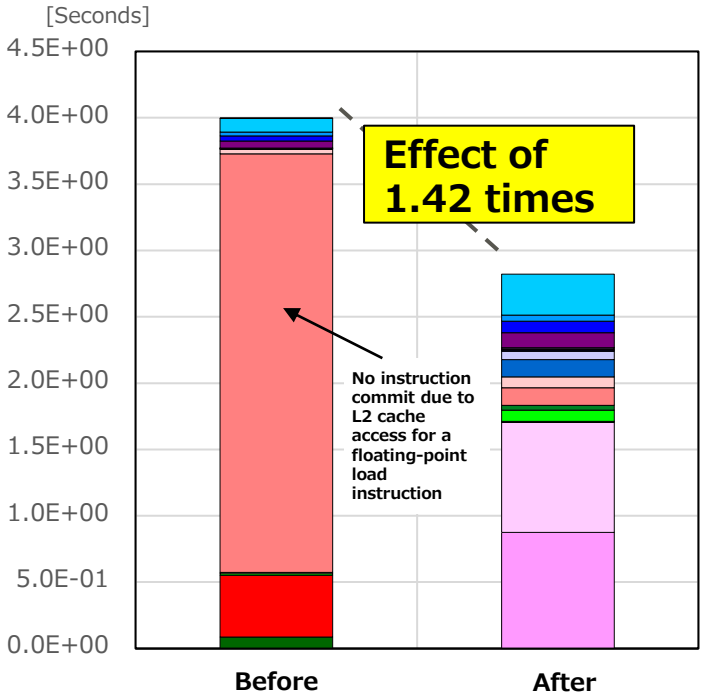
FUJITSU

Specify the prefetch specifier to generate prefetch for indirect access (list access). The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source Before Improvement

```
53          void sub(double * restrict a, double * restrict b,
                     double * restrict c, int * restrict d,
                     int * restrict e, double scalar, int n){
54            int i;
56          #pragma loop prefetch
57          #pragma omp parallel for
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<<    PREFETCH(SOFT) : 8
          <<<     INDIRECT : 8
          <<<      (unknown): 8
          <<< Loop-information  End >>>
58   p    2     for (i = 0; i < n; i++){
59   p    2       a[i] = b[d[i]] + scalar * c[e[i]];
60   p    2     }
61          }
```

Prefetch generated for indirect access (Arrays b and c)



The L1D and L2 miss dm rates were reduced by the generated prefetch instruction for indirect access (Arrays b and c).

Statistics	Memory throughput (GB/s)
Before	31.24
After	159.90

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	8.65E+09	3.78E+09	0.44	98.01%	1.99%	0.00%	4.30E+08	0.05	47.96%	100.00%	0.00%
After	0.00	1.70E+10	3.83E+09	0.22	2.99%	2.94%	94.08%	1.70E+09	0.10	1.80%	6.41%	91.79%

FUJITSU

Using Software Prefetch to Access Non-Sequential Data

- Using Software Prefetch to Access Non-Sequential Data
- Using Software Prefetch to Access Non-Sequential Data (Before Improvement)
- Using Software Prefetch to Access Non-Sequential Data (1) (Optimization Control Line Tuning)
- Using Software Prefetch to Access Non-Sequential Data (2) [Recommended] (Optimization Control Line Tuning)

Using Software Prefetch to Access Non-Sequential Data

FUJITSU

Cache misses occur easily with hardware prefetch in accessing non-sequential data or data with short sequential parts. In such cases, use software prefetch to raise performance.

To use software prefetch, use either the PREFETCH_READ and PREFETCH_WRITE specifiers, which are described later, or the following compiler options.

Compiler Option	Functional Description
-Kprefetch_sequential =auto	Sets the compiler to automatically select whether to use hardware prefetch or output a prefetch instruction for the array data used in a loop and accessed sequentially. This option is valid when the -O1 or higher option is enabled. If the -O2 or higher option is enabled, the default is -Kprefetch_sequential=auto.
-Kprefetch_sequential =soft	Outputs a prefetch instruction for the array data used in a loop and accessed sequentially, rather than using hardware prefetch. This option is valid when the -O1 or higher option is enabled.
-Kprefetch_nosequent ial	Generates an object, rather than using a prefetch instruction, for the array data used in a loop and accessed sequentially. If the -O0 or -O1 option is enabled, the default is -Kprefetch_nosequential.

Fortran Using Software Prefetch to Access Non-Sequential Data

FUJITSU

Specify the PREFETCH_READ and PREFETCH_WRITE specifiers as described below.

Optimization Specifier	Meaning	Optimization Control Line Specifiable?			
		By Program	By DO loop	By Statement	By Array Assignment Statement
PREFETCH_READ(name[,level={1|2}][,strong={0|1}])	Specifies that a prefetch instruction be generated for the referenced data. *name* is an array element name, *level* is the cache level used for prefetch, and *strong* is whether or not to perform strong prefetch.	Yes	No	Yes	No
PREFETCH_WRITE(name[,level={1|2}][,strong={0|1}])	Specifies that a prefetch instruction be generated for the defined data.	Yes	No	Yes	No

Before

```
do j=1,n
   do i=1,isize
      a(i,j) =  b(i,j) + scalar * c(i,j)
   enddo
enddo
```

After Improvement (1) (Software Prefetch Element Specified)

```
do j=1,n
   do i=1,isize
!OCL PREFETCH_WRITE(a(i,j+1),level=1)
!OCL PREFETCH_READ(b(i,j+1),level=1)
!OCL PREFETCH_READ(c(i,j+1),level=1)
      a(i,j) =  b(i,j) + scalar * c(i,j)
   enddo
enddo
```

After Improvement (2) (Software Prefetch Vector Specified)

Recommended

```
do j=1,n
!OCL PREFETCH_WRITE(a(1:isize,j+1),level=1)
!OCL PREFETCH_READ(b(1:isize,j+1),level=1)
!OCL PREFETCH_READ(c(1:isize,j+1),level=1)
   do i=1,isize
      a(i,j) =  b(i,j) + scalar * c(i,j)
   enddo
enddo
```

Array elements are specified in vector format. Since multiple prefetch instructions can be generated simultaneously, performance is even higher than when each element is individually specified.

C/C++ Using Built-in Prefetch to Access Non-Sequential Data

FUJITSU

Built-in prefetch function is used as described below. Refer to the GNU C/C++ compiler websites for specifications.

Before

```
for (j = 0; j < n; j++) {
  for (i = 0; i < isize; i++) {
    a[j][i] = b[j][i] + scalar * c[j][i];
  }
}
```

After (1) (Built-in prefetch Element Specified)

```
for (j = 0; j < n; j++) {
  for (i = 0; i < isize; i++) {
    __builtin_prefetch(&a[j+1][0], 1, 3);
    __builtin_prefetch(&b[j+1][0], 0, 3);
    __builtin_prefetch(&c[j+1][0], 0, 3);
    __builtin_prefetch(&a[j+1][32], 1, 3);
          :
    a[j][i] = b[j][i] + scalar * c[j][i];
  }
}
```

Performance is improved by reducing the number of prefetching.

After (2) (Built-in prefetch Vector Specified)

Recommended

```
for (j = 0; j < n; j++) {
  __builtin_prefetch(&a[j+1][0], 1, 3);
  __builtin_prefetch(&a[j+1][32], 1, 3);
  __builtin_prefetch(&a[j+1][64], 1, 3);
  __builtin_prefetch(&b[j+1][0], 0, 3);
  __builtin_prefetch(&b[j+1][32], 0, 3);
  __builtin_prefetch(&b[j+1][64], 0, 3);
  __builtin_prefetch(&c[j+1][0], 0, 3);
        :
  for (i = 0; i < isize; i++) {
    a[j][i] = b[j][i] + scalar * c[j][i];
  }
}
```

Fortran Using Software Prefetch to Access Non-Sequential Data (Before Improvement)

FUJITSU

Access is not sequential because the number of iterations of the innermost loop is small and the array size is larger than the number of iterations. The cost of prefetch startup is quite apparent in normal prefetch. Consequently, the "No instruction commit due to access for a floating-point load instruction" event occurs many times.

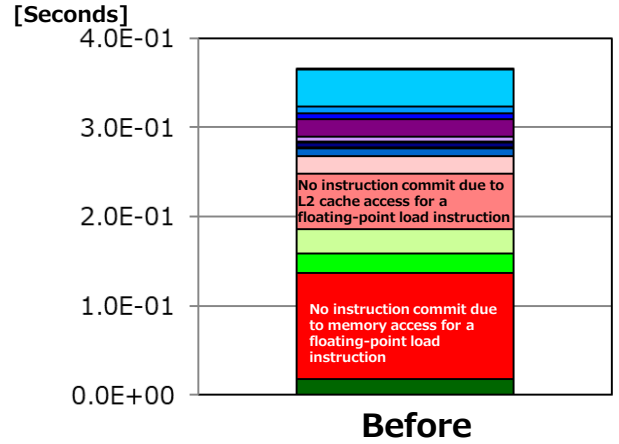
Source Before Improvement

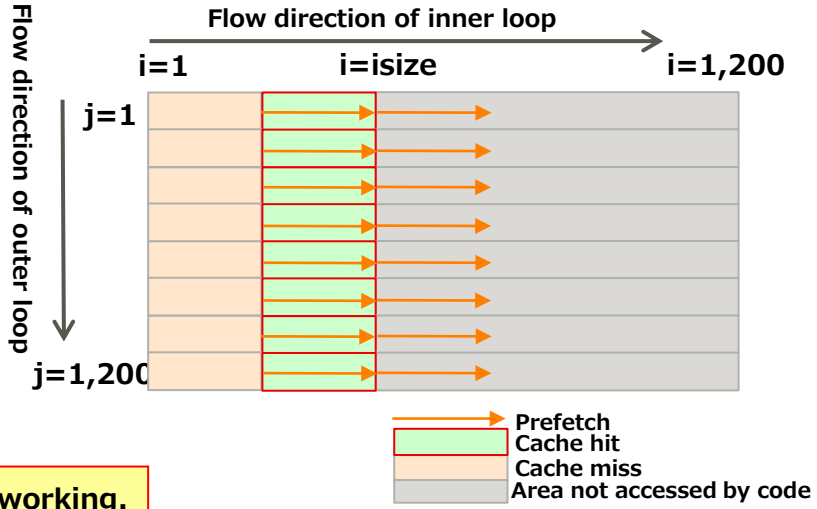
```
42                parameter(n=1200)
43                integer n
44                real*8  a(n,n),b(n,n),c(n,n),scalar
45                common /com/a,b,c
46
47                !$omp parallel do
                  <<< Loop-information
                  <<<  [OPTIMIZATION]
                  <<<    PREFETCH(HARD
                  <<<      c, b, a
                  <<< Loop-information
48    1   p           do j=1,n
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 3.40, ITR: 128, MVE: 3, POL: S)
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      c, b, a
                  <<< Loop-information  End >>>
49    2   p   2v          do i=1,isize
50    2   p   2v            a(i,j) =  b(i,j) + scalar * c(i,j)
51    2   p   2v          enddo
52    1   p           en
```

Number of elements in 1st dimension of array: 1,200

Number of loop iterations (isize): 128

Access continuity broken when outer loop j is incremented





The high L1D miss dm rate means prefetch is not working.

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	1.75E+09	2.64E+08	0.15	70.94%	29.46%	-0.40%	1.26E+08	0.07	39.28%	83.10%	0.00%

Fortran Using Software Prefetch to Access Non-Sequential Data (1) (Optimization Control Line Tuning)

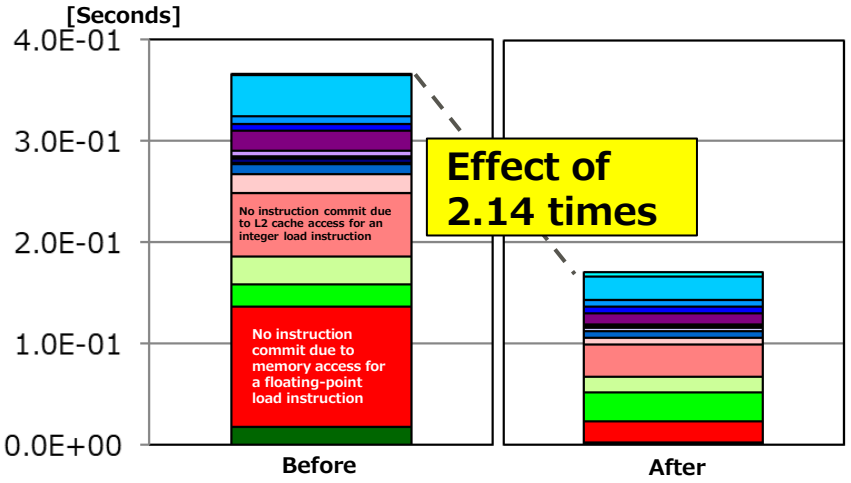
FUJITSU

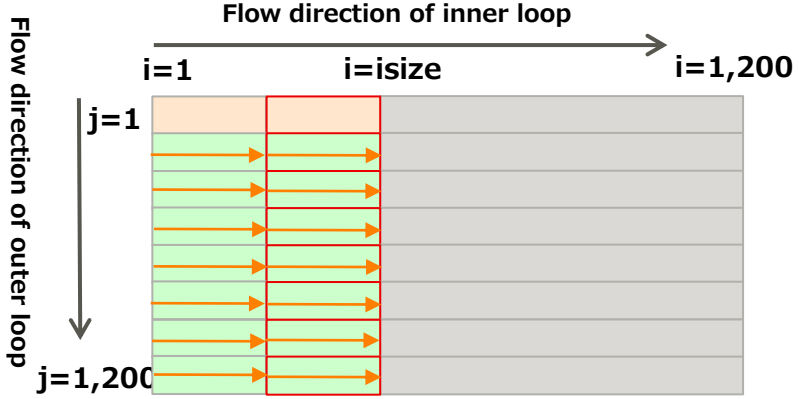
Use the PREFETCH_READ and PREFETCH_WRITE specifiers to generate prefetch for arrays in the outer loops to hide the prefetch startup cost. The result is improvement of the "No instruction commit due to L2 cache access for an integer load instruction" event.

Source After Improvement

```
42                parameter(n=1200)
43                integer n
44                real*8  a(n,n),b(n,n),c(n,n),scalar
45                common /com/a,b,c
46
47              !$omp parallel do
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    PREFETCH(HARD) Expected by compiler :
              <<<      c, b, a
              <<<    PREFETCH(SOFT) : 18
              <<<     SPECIFIED : 18
              <<<      a: 6, c: 6, b: 6
              <<< Loop-information  End >>>
48    1  p          do j=1,n
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    SIMD(VL: 8)
              <<<    SOFTWARE PIPELINING(IPC: 2.25, ITR: 88, MVE: 6, POL: S)
              <<<    PREFETCH(HARD) Expected by compiler :
              <<<      c, b, a
              <<<    PREFETCH(SOFT) : 18
              <<<     SPECIFIED : 18
              <<<      c: 6, b: 6, a: 6
              <<< Loop-information  End >>>
49    2  p  v          do i=1,isize
50    2  p           !OCL PREFETCH_WRITE(a(i,j+1),level=1)
51    2  p           !OCL PREFETCH_READ(b(i,j+1),level=1)
52    2  p           !OCL PREFETCH_READ(c(i,j+1),level=1)
53    2  p  v            a(i,j) =  b(i,j) + scalar * c(i,j)
54    2  p  v          enddo
55    1  p          enddo
```

Prefetch in array at 1 iteration ahead in outer loop





L1D miss dm rate reduced

Cache	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
Before	1.75E+09	2.64E+08	0.15	70.94%	29.46%	-0.40%
After	1.13E+09	1.90E+08	0.17	4.57%	0.44%	94.99%

Fortran — Using Software Prefetch to Access Non-Sequential Data (2) [Recommended] (Optimization Control Line Tuning)

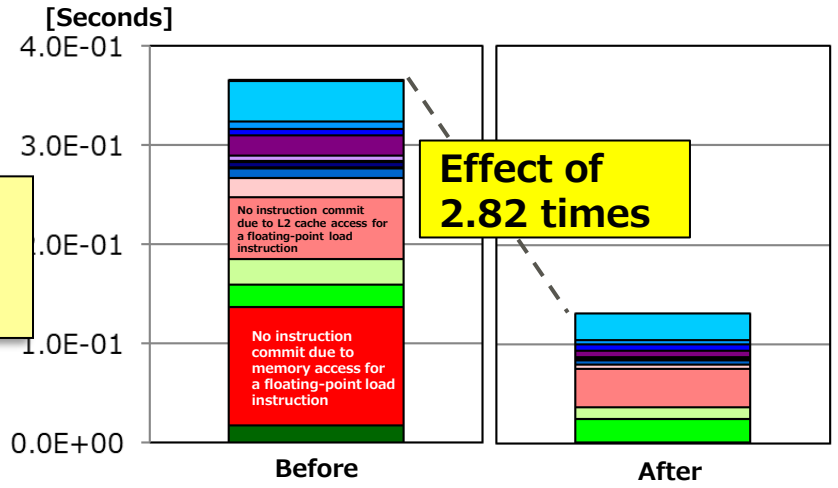
FUJITSU

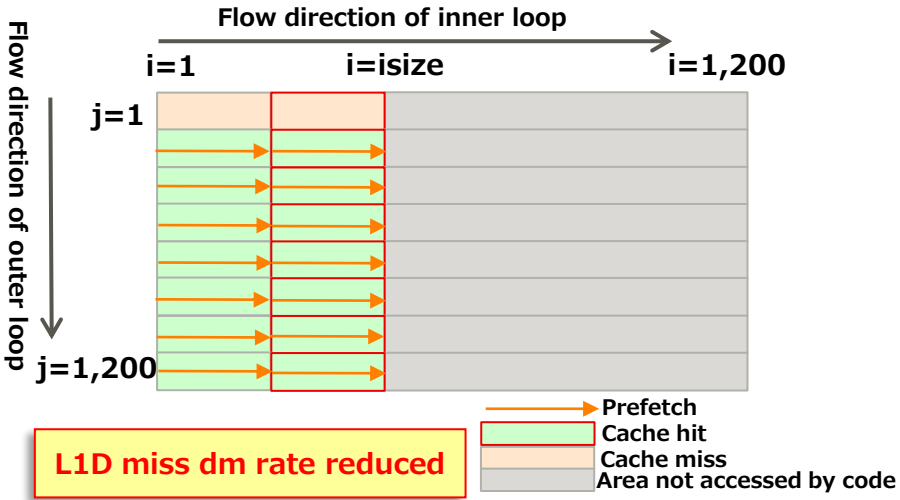
You can reduce the number of instructions of the innermost loop and raise performance by specifying the PREFETCH_READ and PREFETCH_WRITE specifiers for an array in an outer loop.

Source After Improvement

```
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, b, a
                <<<    PREFETCH(SOFT) : 3
                <<<     SPECIFIED : 3
                <<<      a: 1, c: 1, b: 1
                <<< Loop-information  End >>>
48    1  p           do j=1,n
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(SOFT) : 4
                <<<     SPECIFIED : 4
                <<<      a: 4
                <<< Loop-information  End >>>
49    1  p  4s     !OCL PREFETCH_WRITE(a(1:isize,j+1),level=1)
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(SOFT) : 4
                <<<     SPECIFIED : 4
                <<<      b: 4
                <<< Loop-information  End >>>
50    1  p  4s     !OCL PREFETCH_READ(b(1:isize,j+1),level=1)
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(SOFT) : 4
                <<<     SPECIFIED : 4
                <<<      c: 4
                <<< Loop-information  End >>>
51    1  p  4s     !OCL PREFETCH_READ(c(1:isize,j+1),level=1)
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.40, ITR: 128, MVE: 3, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, b, a
                <<< Loop-information  End >>>
52    2  p  2v          do i=1,isize
53    2  p  2v            a(i,j) =  b(i,j) + scalar * c(i,j)
54    2  p  2v          enddo
55    1  p           enddo
```

Prefetch of entire array at 1 iteration ahead in outer loop





L1D miss dm rate reduced

Cache	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1Dmiss)	L1D miss hardware prefetch rate (%) (/L1Dmiss)	L1D miss software prefetch rate (%) (/L1Dmiss)
Before	1.75E+09	2.64E+08	0.15	70.94%	29.46%	-0.40%
After	9.22E+08	1.93E+08	0.21	23.46%	1.65%	74.90%

C/C++ Using Built-in Prefetch to Access Non-Sequential Data (Before Improvement)

FUJITSU

Access is not sequential because the number of iterations of the innermost loop is small and the array size is larger than the number of iterations. The cost of prefetch startup is quite apparent in normal prefetch. Consequently, the "No instruction commit due to access for a floating-point load instruction" event occurs many times.

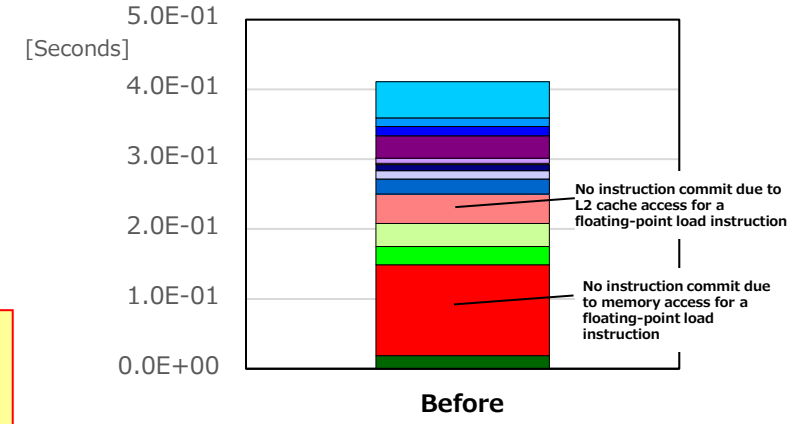
Source Before Improvement

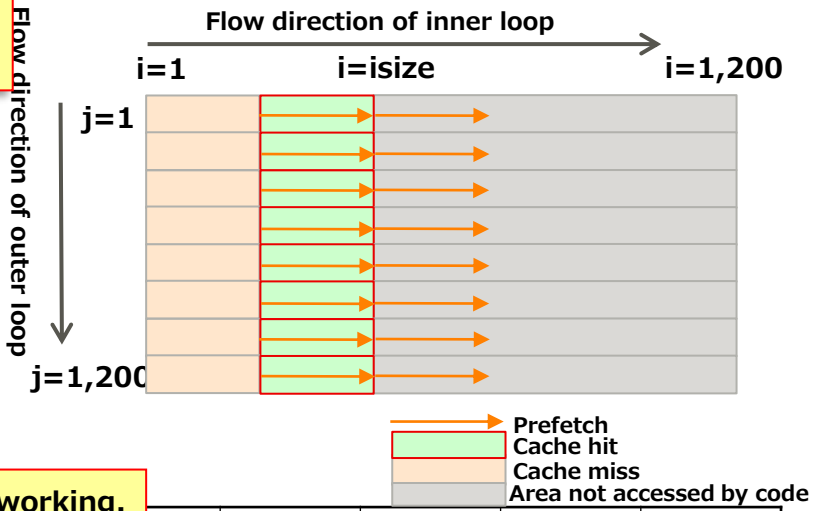
```
40          void sub(double scalar, int isize){
41           int i, j;
42
43          #pragma omp parallel for
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD)
          <<<      c, b, a
          <<< Loop-information  E
44   p        for (j = 0; j < n; j++)
          <<< Loop-information St
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 144, MVE: 4, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      c, b, a
          <<< Loop-information  End >>>
45   p    2v     for (i = 0; i < isize; i++){
46   p    2v       a[j][i] =  b[j][i] + scalar * c[j][i];
47   p    2v     }
48   p         }
49          }
```

Number of elements in 1st dimension of array: 1,200

Number of loop iterations (isize): 128

Access continuity broken when outer loop j is incremented





The high L1D miss dm rate means prefetch is not working.

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	hardware prefetch rate (%) (/L1D miss)	software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	2.18E+09	2.57E+08	0.12	73.77%	26.27%	-0.04%	6.67E+07	0.03	42.40%	82.13%	0.00%

Using Built-in Prefetch to Access Non-Sequential Data (1) (Optimization Control Line Tuning)

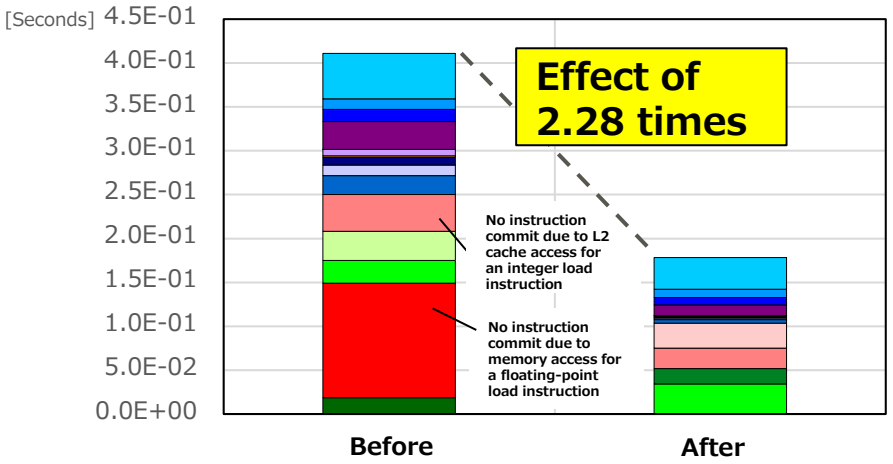
FUJITSU

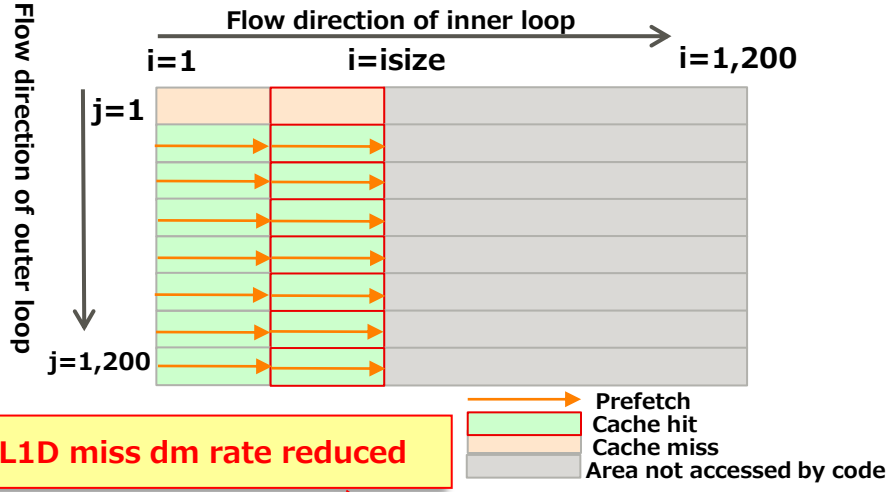
Use the built-in prefetch functions to generate prefetch for arrays in the outer loops to hide the prefetch startup cost. The result is improvement of the "No instruction commit due to L2 cache access for an integer load instruction" event.

Source After Improvement

```
41          void sub(double scalar, int isize){
42            int i, j;
43
44          #pragma omp parallel for
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      c, b, a
          <<<    PREFETCH(SOFT) : 48
          <<<     SPECIFIED : 48
          <<<      a: 16, c: 16, b: 16
          <<< Loop-information  End >>>
45  p         for (j = 0; j < n; j++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.62, ITR: 56, MVE: 4, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      c, a, b
          <<<    PREFETCH(SOFT) : 48
          <<<     SPECIFIED : 48
          <<<      b: 16, c: 16, a: 16
          <<< Loop-information  End >>>
46  p   v     for(i = 0; i < isize; i++){
47  p   v       __builtin_prefetch(&a[j+1][0], 1, 3);
48  p   v       __builtin_prefetch(&b[j+1][0], 0, 3);
49  p   v       __builtin_prefetch(&c[j+1][0], 0, 3);
50  p   v       __builtin_prefetch(&a[j+1][32], 1, 3);
51  p   v       __builtin_prefetch(&b[j+1][32], 0, 3);
52  p   v       __builtin_prefetch(&c[j+1][32], 0, 3);
53  p   v       __builtin_prefetch(&a[j+1][64], 1, 3);
54  p   v       __builtin_prefetch(&b[j+1][64], 0, 3);
55  p   v       __builtin_prefetch(&c[j+1][64], 0, 3);
56  p   v       __builtin_prefetch(&a[j+1][96], 1, 3);
57  p   v       __builtin_prefetch(&b[j+1][96], 0, 3);
58  p   v       __builtin_prefetch(&c[j+1][96], 0, 3);
59  p   v       a[j][i] =  b[j][i] + scalar * c[j][i];
60  p   v     }
61  p       }
62          }
```

Prefetch in array at 1 iteration ahead in outer loop





L1D miss dm rate reduced

Cache	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
Before	2.18E+09	2.57E+08	0.12	73.77%	26.27%	-0.04%
After	1.16E+09	1.93E+08	0.17	14.76%	1.20%	84.04%

C/C++ Using Built-in Prefetch to Access Non-Sequential Data (2) [Recommended] (Optimization Control Line Tuning)

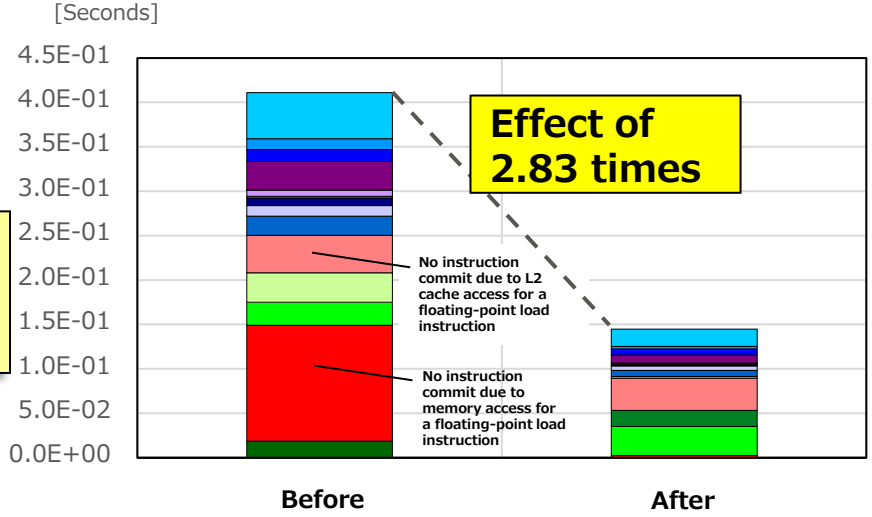
FUJITSU

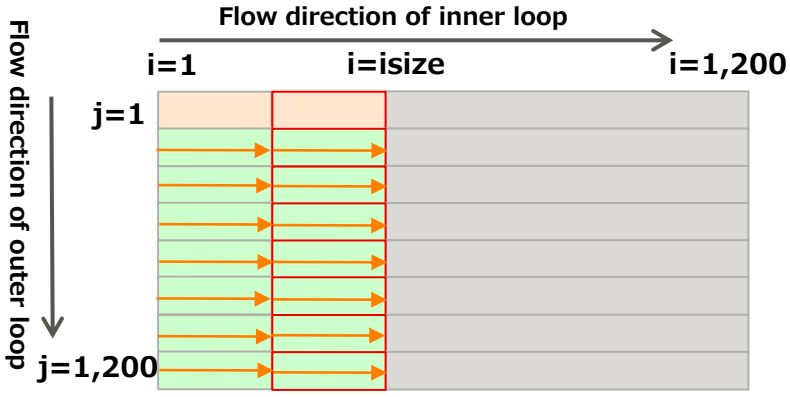
You can reduce the number of instructions of the innermost loop and raise performance by specifying the built-in prefetch functions for an array in an outer loop.

Source After Improvement

```
41          void sub(double scalar, int isize){
42            int i, j;
43
44          #pragma omp parallel for
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expect
           <<<      c, b, a
           <<<    PREFETCH(SOFT) : 12
           <<<     SPECIFIED : 12
           <<<      a: 4, b: 4, c: 4
           <<< Loop-information  End >>>
45  p         for (j = 0; j < n; j++){
46  p           __builtin_prefetch(&a[j+1][0], 1, 3);
47  p           __builtin_prefetch(&a[j+1][32], 1, 3);
48  p           __builtin_prefetch(&a[j+1][64], 1, 3);
49  p           __builtin_prefetch(&a[j+1][96], 1, 3);
50  p           __builtin_prefetch(&b[j+1][0], 0, 3);
51  p           __builtin_prefetch(&b[j+1][32], 0, 3);
52  p           __builtin_prefetch(&b[j+1][64], 0, 3);
53  p           __builtin_prefetch(&b[j+1][96], 0, 3);
54  p           __builtin_prefetch(&c[j+1][0], 0, 3);
55  p           __builtin_prefetch(&c[j+1][32], 0, 3);
56  p           __builtin_prefetch(&c[j+1][64], 0, 3);
57  p           __builtin_prefetch(&c[j+1][96], 0, 3);
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 14
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      c, b, a
           <<< Loop-information  End >>>
58  p   2v      for (i = 0; i < isize; i++){
59  p   2v        a[j][i] = b[j][i] + scalar * c[j][i];
60  p   2v      }
61  p         }
62          }
```

Prefetch of entire array at 1 iteration ahead in outer loop





L1D miss dm rate reduced

Cache	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
Before	2.18E+09	2.57E+08	0.12	73.77%	26.27%	-0.04%
After	1.17E+09	1.94E+08	0.17	23.67%	1.43%	74.90%

Data Access Wait Time (Reduced Access Amount)

- High-Speed Store (ZFILL)

FUJITSU

High-Speed Store (ZFILL)

- What is High-Speed Store (ZFILL)?
- ZFILL (Before Improvement)
- Effect of ZFILL (Optimization Control Line Tuning)

What is High-Speed Store (ZFILL)?

- What is high-speed store (ZFILL)?

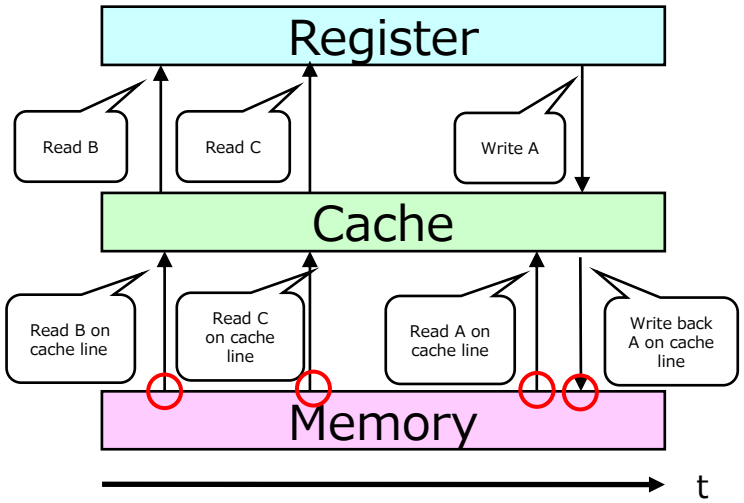
  ZFILL is a function that secures a cache line (containing undefined values) for writing in the cache. The function can reduce cache line read from memory to improve the performance of a program whose bottleneck is memory throughput.

- Operating conditions
  - The array to be stored has no dependency across iterations.
  - Arrays with definitions have no references.
  - Memory access is sequential.

Example:

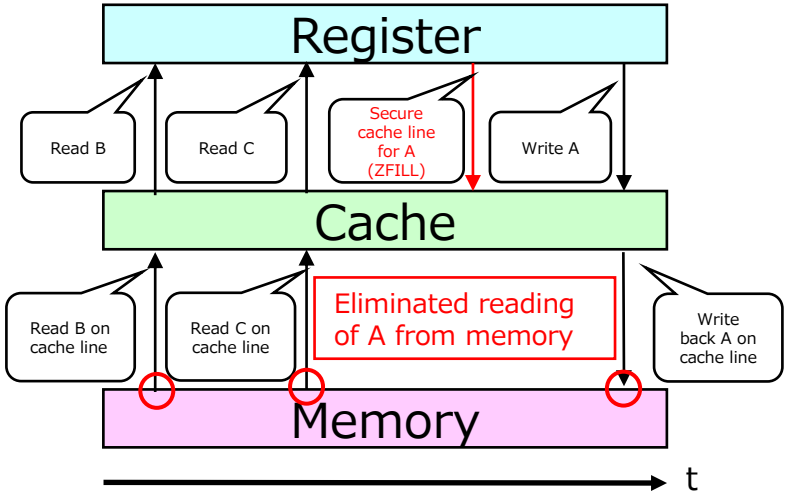
```
DO I = 1, N
      A(I) = B(I) + C(I)
END DO
```

ZFILL not used



Total number of memory accesses: 4

ZFILL used



Total number of memory accesses: 3

What is ZFILL? (1/2)

FUJITSU

Specify the following optimization control lines.

Optimization Specifier (Fortran)	Meaning	Optimization Control Line Specifiable?			
		By Program	By DO Loop	By Statement	By Array Assignment Statement
ZFILL[(*m1*)]	Specifies that a ZFILL instruction be generated. *m1* is a decimal number (1 to 100) representing the number of cache lines.	No	Yes	No	Yes
NOZFILL	Specifies that no ZFILL instruction be generated.	No	Yes	No	Yes

Optimization Specifier (C/C++)	Meaning	Optimization Control Line Specifiable?			
		global	procedure	loop	statement
zfill[(*m1*)]	Specifies that a ZFILL instruction be generated. *m1* is a decimal number (1 to 100) representing the number of cache lines.	No	No	Yes	No
nozfill	Specifies that no ZFILL instruction be generated.	No	No	Yes	No

- **Note**
- The ZFILL instruction is output for the array data stored in a loop. However, it is not output for arrays with references in the same loop, arrays that are not sequentially accessed, and arrays stored under IF statements.
- No prefetch instruction to the secondary cache is output when the ZFILL instruction is output.
- To definitely store the loop together with the cache line secured by the ZFILL instruction, the loop is transformed. Consequently, the following optimizations cannot be applied. This may degrade execution performance.
  - ✓ Loop unrolling
  - ✓ Loop striping
- Execution performance may also degrade in the following cases:
  - ✓ Loop with a small number of iterations
  - ✓ Data is in the primary or secondary cache

What is ZFILL? (2/2)

FUJITSU

You can obtain an effect equivalent to that of optimization control line tuning by specifying the following compiler option.

Compiler Option	Functional Description
-K{ zfill[=*N*] | nozfill } 1 ≦ *N* ≦ 100	Specifies that an instruction (ZFILL instruction) be generated for the array data written only in a loop in order to secure a cache line for writing in the cache rather than loading data from memory. Specify *N* to target the data located *N* cache lines ahead of the ZFILL instruction. You can specify *N* in a range from 1 to 100. If *N* is not specified, the compiler automatically decides a value. This option is valid when -O2 or higher option is enabled. The default is -Knozfill.

■**Use example (for source before improvement)**

```
$ frtpx -Kfast,parallel sample.f90 -Kzfill
$ fccpx -Kfast,parallel sample.f90 –Kzfill
```

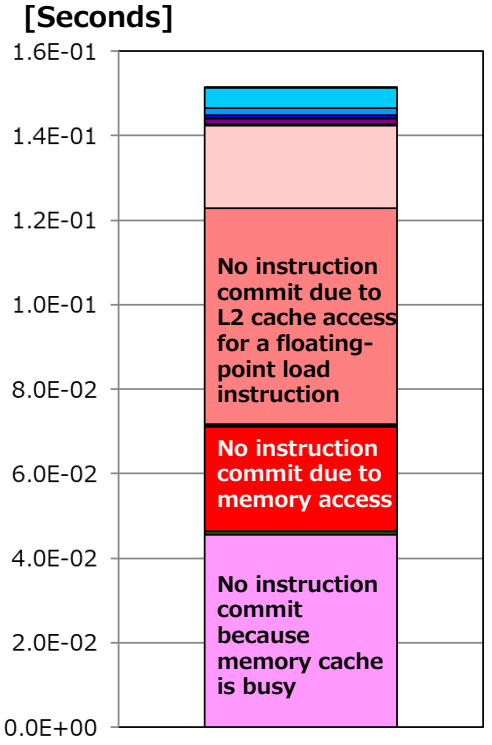
Fortran ZFILL (Before Improvement)

FUJITSU

Memory throughput is a bottleneck because the memory access load of the program is high. Consequently, the data access wait time is long.

Source Before Improvement

```
38                real*8 a(n),b(n),c(n),d
39
                  <<< Loop-information Start >>>
                  <<<  [PARALLELIZATION]
                  <<<    Standard iteration count: 1000
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 144,
                                                 MVE: 4, POL: S)
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      c, b, a
                  <<< Loop-information  End >>>
40     1  pp   2v      do i=1,n
41     1   p   2v        a(i) = b(i) + c(i)*d
42     1   p   2v      enddo
```



Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.90E+08	9.39E+07	0.24	27.78%	72.21%	0.01%	9.38E+07	0.24	12.72%	88.66%	0.00%

	Memory Throughput (GB/s)
Before	211.59

Memory throughput is bottleneck

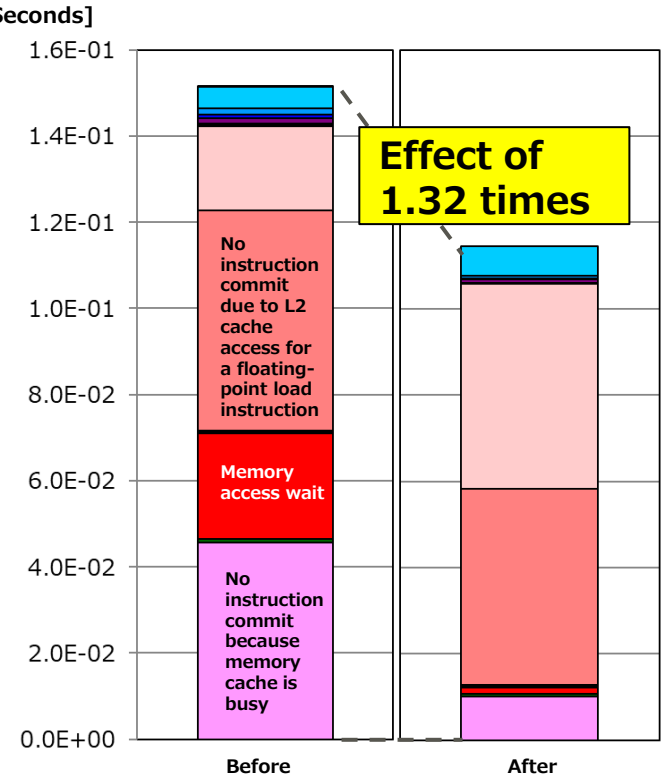
Fortran Effect of ZFILL (Optimization Control Line Tuning)

FUJITSU

Specify the ZFILL specifier to eliminate cache line reading from memory according to a store instruction and to reduce the number of L2 misses. This results in an improved data access wait time.

Source After Improvement (Optimization Control Line Tuning)

```
38                real*8 a(n),b(n),c(n),d
39                !ocl zfill
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 1000
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.55, ITR: 128,
                                           MVE: 2, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, b
                <<<    PREFETCH(SOFT) : 2
                <<<     SEQUENTIAL : 2
                <<<      a: 2
                <<<    ZFILL         :
                <<<      a
                <<< Loop-information  End >>>
40    1  pp   v      do i=1,n
41    1   p   v        a(i) = b(i) + c(i)*d
42    1   p   v      enddo
```



	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.90E+08	9.39E+07	0.24	27.78%	72.21%	0.01%	9.38E+07	0.24	12.72%	88.66%	0.00%
After	0.00	4.38E+08	9.39E+07	0.21	16.80%	49.98%	33.22%	6.25E+07	0.14	1.15%	98.98%	0.00%

	Memory Throughput (GB/s)
Before	211.59
After	209.96

Number of L2 misses reduced by 1/3, though memory throughput is still bottleneck even after improvement

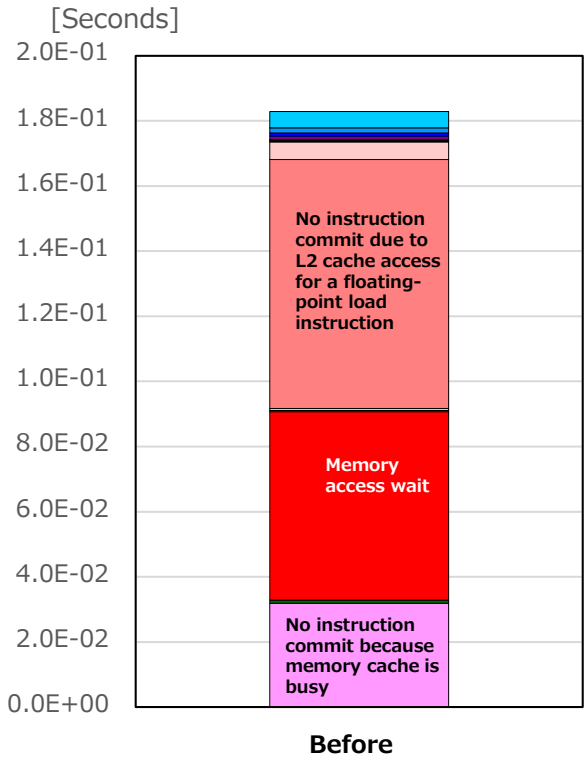
ZFILL (Before Improvement)

FUJITSU

The data access wait time is long because the memory access load of the program is high.

Source Before Improvement

```
38          void sub(double * restrict a, double * restrict b,
                     double * restrict c, double d, int n){
39            int i;
40
41          #pragma omp parallel for
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 144,
                        MVE: 4, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
42  p    2v    for (i = 0; i < n; i++){
43  p    2v      a[i] = b[i] + c[i]*d;
44  p    2v    }
45          }
```



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.95E+08	9.40E+07	0.24	41.24%	58.75%	0.01%	9.39E+07	0.24	19.15%	83.83%	0.00%

Statistics	Memory throughput (GB/s)
Before	175.68

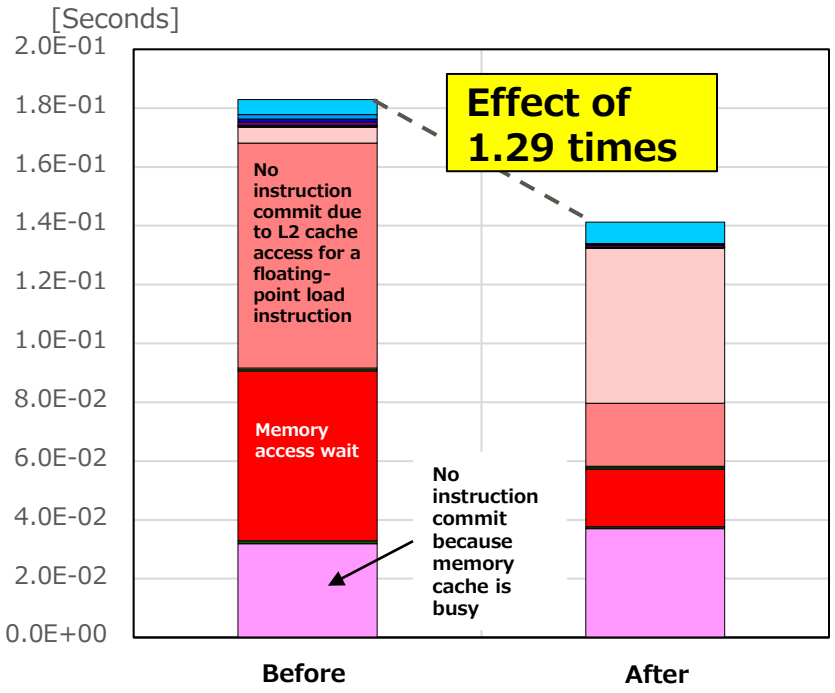
C/C++ Effect of ZFILL (Optimization Control Line Tuning)

FUJITSU

Specify the ZFILL specifier to eliminate cache line reading from memory according to a store instruction and to reduce the number of L2 misses. This results in an improved data access wait time.

Source After Improvement (Optimization Control Line Tuning)

```
37          void sub(double * restrict a, double * restrict b,
                     double * restrict c, double d, int n){
38            int i;
39
40          #pragma omp parallel for
41          #pragma loop zfill
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.55, ITR: 128,
                                     MVE: 2, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<<    PREFETCH(SOFT) : 2
          <<<     SEQUENTIAL : 2
          <<<      (unknown): 2
          <<<    ZFILL          :
          <<<      (unknown)
          <<< Loop-information  End >>>
42  p       v    for (i = 0; i < n; i++){
43  p       v      a[i] = b[i] + c[i]*d;
44  p       v    }
45            }
```



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.95E+08	9.40E+07	0.24	41.24%	58.75%	0.01%	9.39E+07	0.24	19.15%	83.83%	0.00%
After	0.00	4.31E+08	9.40E+07	0.22	30.94%	35.79%	33.27%	6.26E+07	0.15	4.59%	95.79%	0.00%

Statistics	Memory throughput (GB/s)
Before	175.68
After	170.27

Number of L2 misses reduced by 1/3

FUJITSU

Data Access Wait Time (Improved Thrashing)

- What is Cache Thrashing?
- Padding That Increases Array Elements in the First Dimension
- Padding That Increases Array Elements in the Second Dimension
- Padding With Dummy Arrays
- Padding With Dummy Arrays (Arrays of Different Sizes)
- Array Merge (Improved Thrashing)
- Loop Fission (Improved Thrashing)
- Padding Using the Large Page Environment Variable

What is Cache Thrashing?

Cache thrashing is a phenomenon where only data with specific indexes (location information) in the cache is frequently overwritten.
This phenomenon occurs easily when the array size is a power of 2 (multiple of 16 KB), since the size per way is 16 KB, and when the number of streams in a loop is large.

Note: A stream is a series of data referenced and defined in association with loop iterations.

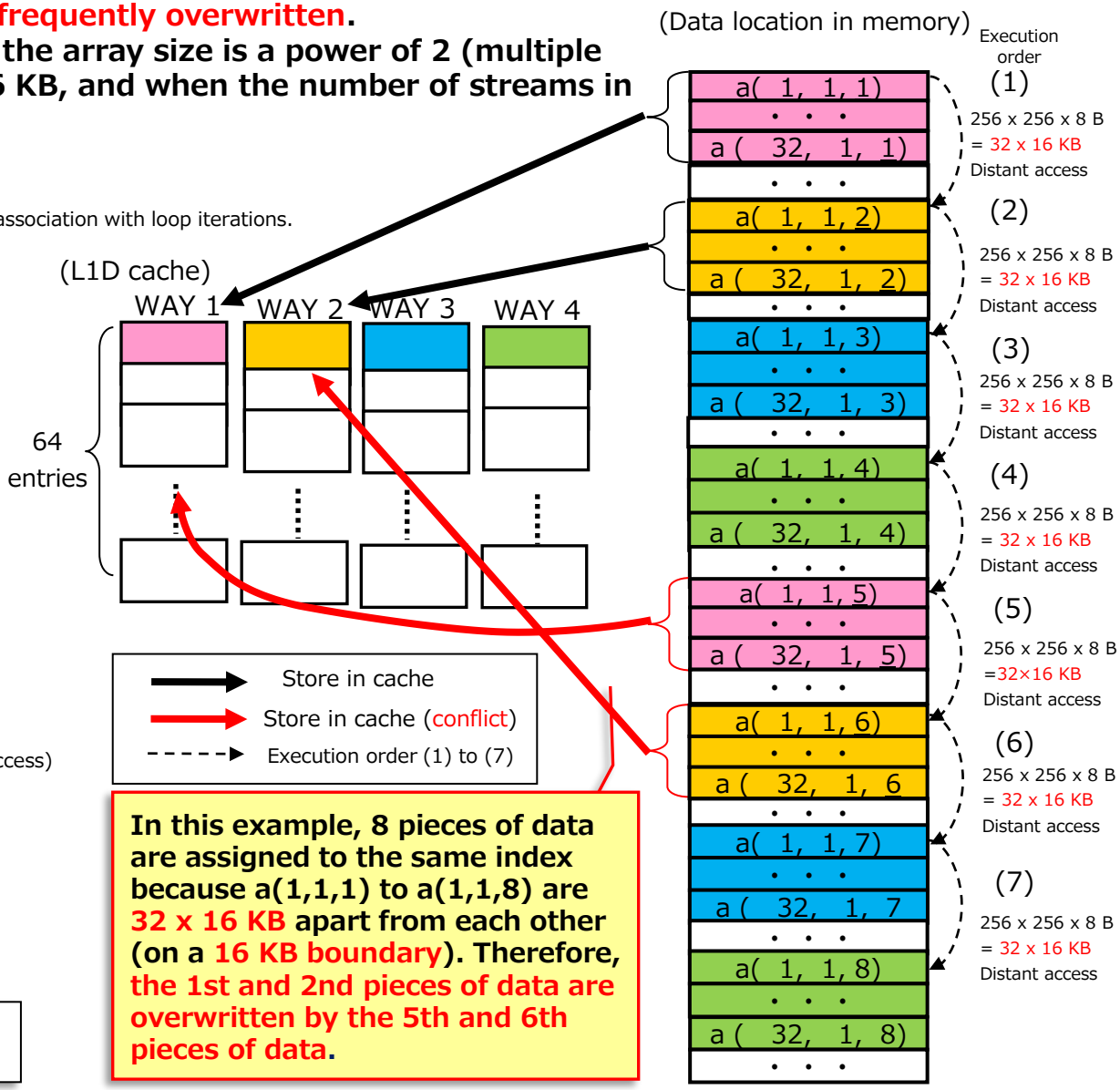
Example of Source

```
subroutine sub(a, n, m) *n=256, m=256
real*8  a(n,m,8)
do j= 1 , m
  do i= 1 , n
    a(i,j,8)=a(i,j,1)+a(i,j,2)+a(i,j,3)+a(i,j,4)+
             a(i,j,5)+a(i,j,6)+a(i,j,7)
  enddo
enddo
end
```

- L1D cache thrashing guideline
  (for 512-bit SVE access instructions and sequential access)

L1D miss rate (/Load-store instruction)
0.25 or higher

Single precision: 64/256
Double precision: 64/256



In this example, 8 pieces of data are assigned to the same index because a(1,1,1) to a(1,1,8) are 32 x 16 KB apart from each other (on a 16 KB boundary). Therefore, the 1st and 2nd pieces of data are overwritten by the 5th and 6th pieces of data.

FUJITSU

Padding

- What is Padding?
- Padding That Increases Array Elements in the First Dimension
- Padding That Increases Array Elements in the Second Dimension
- Padding With Dummy Arrays
- Padding With Dummy Arrays (Arrays of Different Sizes)

What is Padding?

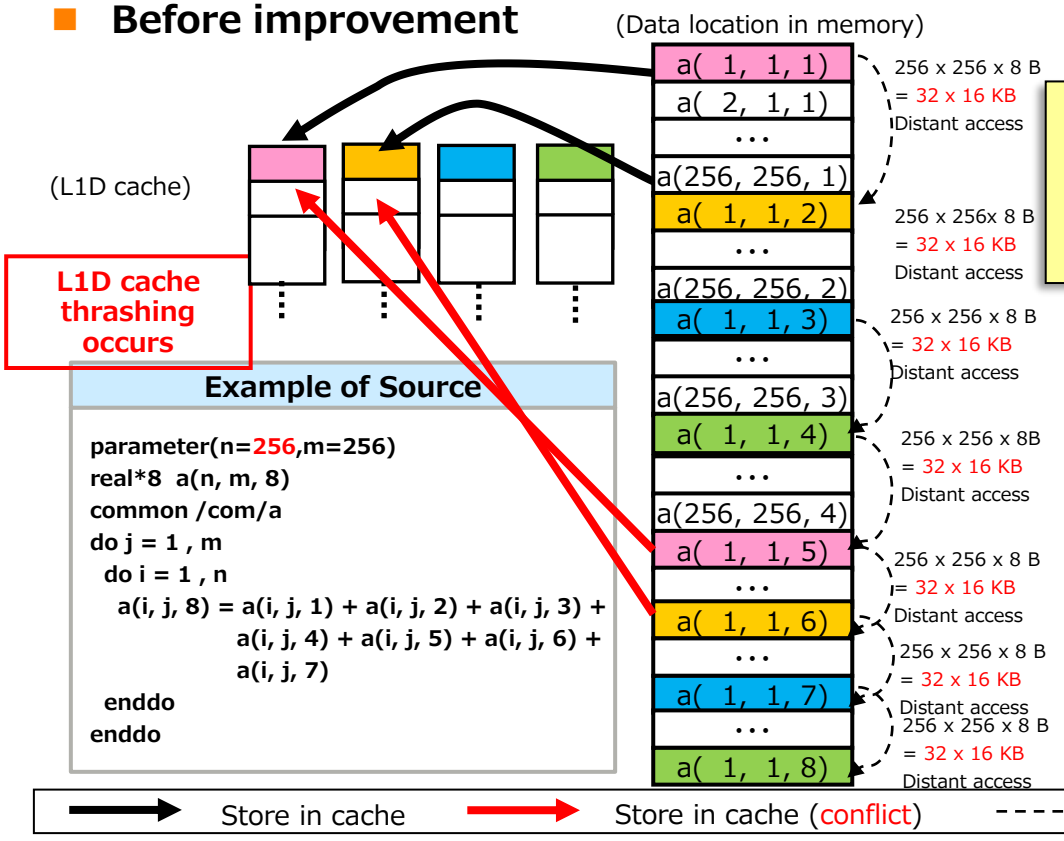
FUJITSU

Padding is a means to insert a dummy area between arrays or into an array.

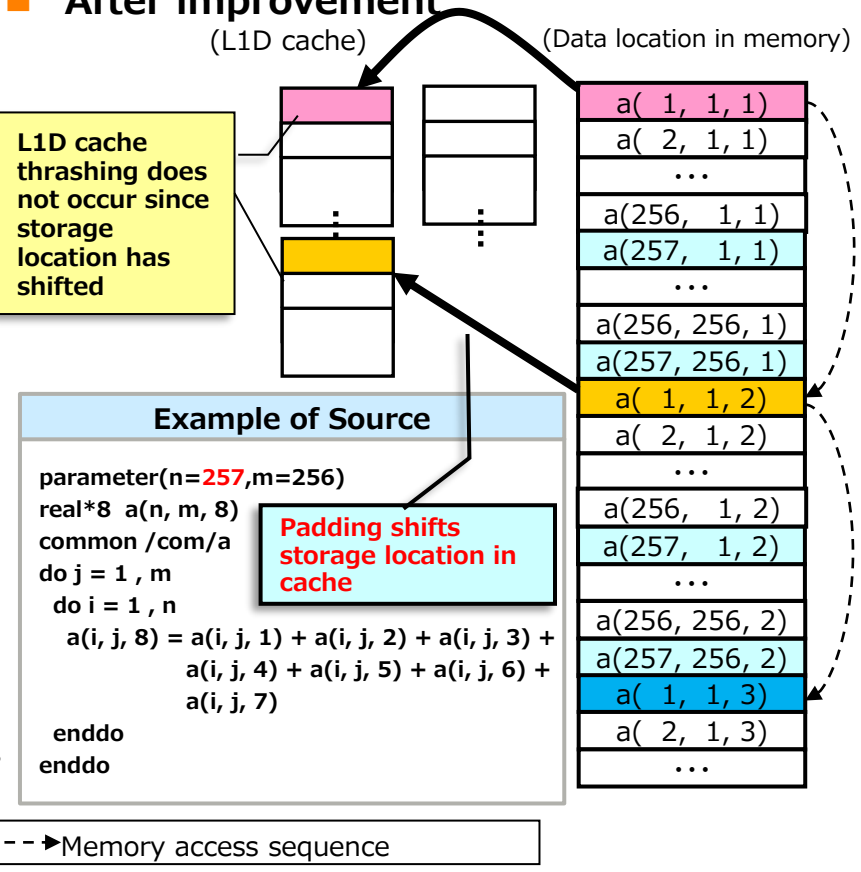
- Use conditions
  Multiple streams of the same array exist
  or
  multiple arrays exist.
- Purpose
  To create a temporary area to shift addresses
- Adverse effect
  The amount of padding must change every time that the scale of the problem changes.

Example where multiple streams of the same array exist

Before improvement



After improvement



Store in cache Store in cache (conflict) Memory access sequence

FUJITSU

Padding That Increases Array Elements in the First Dimension

- Padding That Increases Array Elements in the First Dimension (Before Improvement)
- Padding That Increases Array Elements in the First Dimension (After Improvement)
- Effect of Padding That Increases Array Elements in the First Dimension (Compiler Option Tuning)

Fortran Padding That Increases Array Elements in the First Dimension (Before Improvement)

FUJITSU

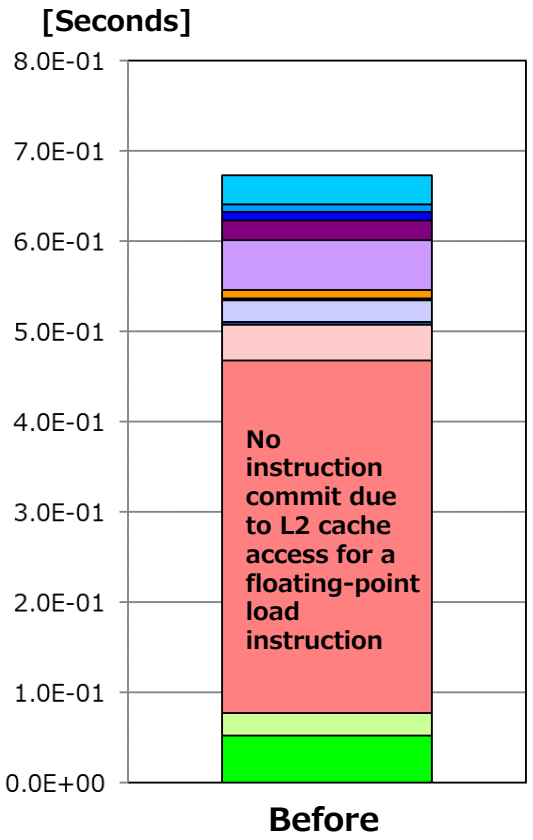
Each stream of Array a is on a 16 KB boundary. L1D cache thrashing occurs. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
42              parameter(n=256,m=256)
43              real*8  a(n, m, 8)
44              common /com/a
                <<< Loop-information Start
                <<<  [PARALLELIZATION]
                <<<    Standard iteration co
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104,
                                           MVE: 7, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a
                <<< Loop-information  End >>>
45    1  pp   v      do j = 1 , m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<< Loop-information  End >>>
46    2  p           do i = 1 , n
47    2  p    v       a(i, j, 8) = a(i, j, 1) + a(i, j, 2) + a(i, j, 3) + &
48    2                    a(i, j, 4) + a(i, j, 5) + a(i, j, 6) + a(i, j, 7)
49    2  p    v      enddo
50    1  p          enddo
```

Array size
256 x 256 x 8 B =
32 x 16 KB
(16 KB boundary)

Streams of same array



High L1D miss rates despite sequential array access
-> L1D cache thrashing occurs

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.08E+09	1.30E+09	0.42	67.73%	32.27%	0.00%	1.50E+04	0.00	37.70%	72.22%	0.00%

Fortran Padding That Increases Array Elements in the First Dimension (After Improvement)

FUJITSU

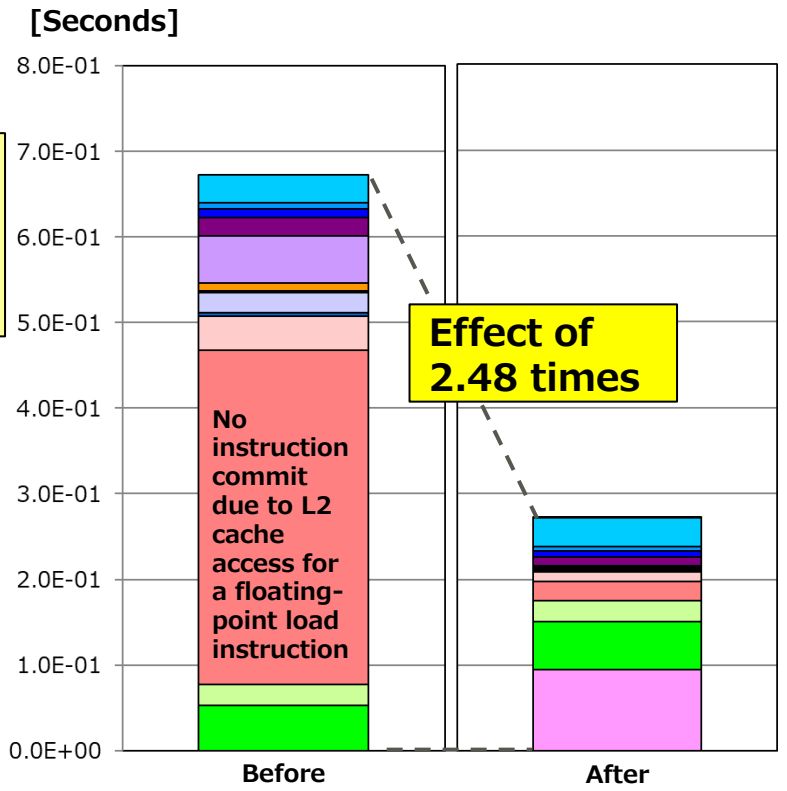
Add padding (+1) to the first dimension of each stream of Array a to prevent L1D cache thrashing. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement

```
42              parameter(n=257,m=256)
43              real*8  a(n, m, 8)
44              common /com/a
             <<< Loop-information Start
             <<<  [PARALLELIZATION]
             <<<    Standard iteration cou
             <<<  [OPTIMIZATION]
             <<<    COLLAPSED
             <<<    SIMD(VL: 8)
             <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104,
                                        MVE: 7, POL: S)
             <<<    PREFETCH(HARD) Expected by compiler :
             <<<      a
             <<< Loop-information  End >>>
45    1  pp   v     do j = 1 , m
             <<< Loop-information Start >>>
             <<<  [OPTIMIZATION]
             <<<    COLLAPSED
             <<< Loop-information  End >>>
46    2   p          do i = 1 , n
47    2   p   v        a(i, j, 8) = a(i, j, 1) + a(i, j, 2) + a(i, j, 3) + &
48    2                             a(i, j, 4) + a(i, j, 5) + a(i, j, 6) + a(i, j, 7)
49    2   p   v      enddo
50    1   p        enddo
```

1 added to n to shift data from 16 KB boundary

L1D misses reduced



Note
An overly large padding count may have a negative impact on data continuity, disabling hardware prefetch.

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.08E+09	1.30E+09	0.42	67.73%	32.27%	0.00%	1.50E+04	0.00	37.70%	72.22%	0.00%
After	0.00	2.62E+09	4.58E+08	0.17	8.20%	91.80%	0.00%	9.69E+03	0.00	46.19%	58.89%	0.00%

C/C++ Padding That Increases Array Elements in the Final Dimension (Before Improvement)

FUJITSU

Each stream of Array a is on a 16 KB boundary. L1D cache thrashing occurs. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

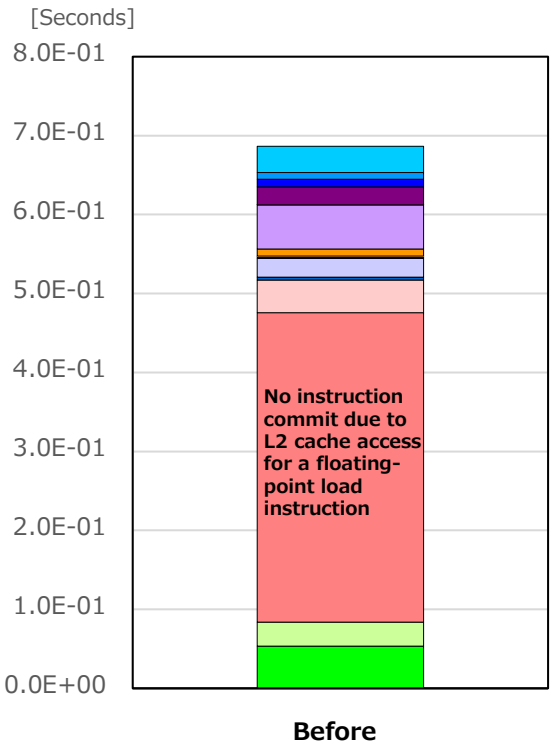
Source Before Improvement

```
30          void sub(void){
31            int i, j;
32
33          #pragma omp parallel for collapse(2)
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      a
            <<< Loop-information  End >>>
34  p     v    for (j = 0; j < m; j++){
35  p     v      for (i = 0; i < n; i++){
36  p     v        a[7][j][i] = a[0][j][i] + a[1][j][i] + a[2][j][i] + a[3][j][i] +
                               a[4][j][i] + a[5][j][i] + a[6][j][i];
37  p     v      }
38  p     v    }
39          }
```

Array declaration
double a[8][256][256];

Array a size
256×256×8B=
32×16 KB(16 KB boundary)

Streams of same array



High L1D miss rates despite sequential array access
-> L1D cache thrashing occurs

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.08E+09	1.33E+09	0.43	68.32%	31.68%	0.00%	2.65E+04	0.00	49.48%	60.25%	0.00%

C/C++ Padding That Increases Array Elements in the Final Dimension (After Improvement)

FUJITSU

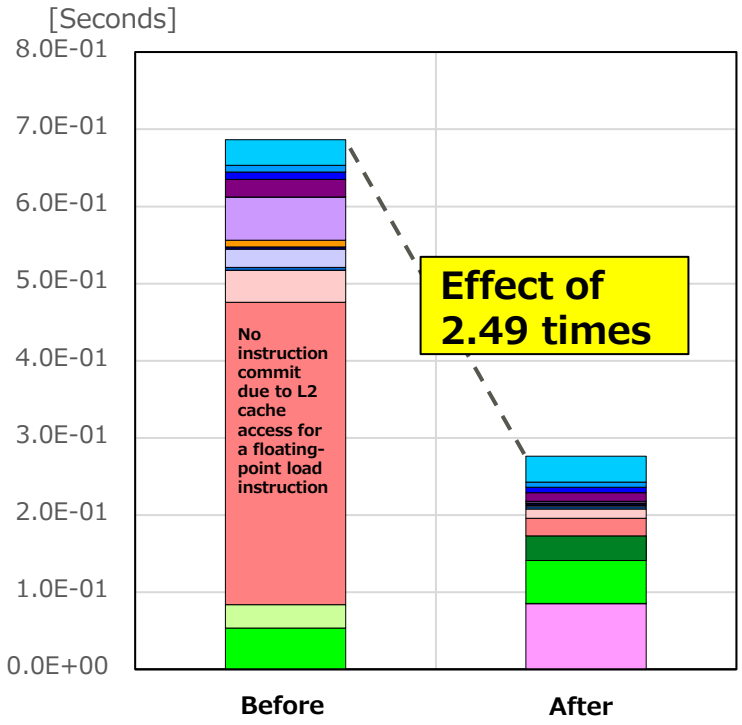
Add padding (+1) to the final dimension of each stream of Array a to prevent L1D cache thrashing. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement

```
30          void sub(void){
31            int i, j;
32
33          #pragma omp parallel for collaps
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      a
            <<< Loop-information  End >>>
34  p     v    for (j = 0; j < m; j++){
35  p     v      for (i = 0; i < n; i++){
36  p     v        a[7][j][i] = a[0][j][i] + a[1][j][i] + a[2][j][i] +
                               a[3][j][i] + a[4][j][i] + a[5][j][i] + a[6][j][i];
37  p     v      }
38  p     v    }
39          }
```

Array declaration
double a[8][256][257];

Shift from the 16 KB boundary by increasing (+1) the elements of the final dimension of array a.



L1D misses reduced

■ **Note**
An overly large padding count may have a negative impact on data continuity, disabling hardware prefetch.

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.08E+09	1.33E+09	0.43	68.32%	31.68%	0.00%	2.65E+04	0.00	49.48%	60.25%	0.00%
After	0.00	2.67E+09	4.60E+08	0.17	8.50%	91.50%	0.00%	2.26E+04	0.00	22.14%	83.91%	0.00%

Fortran Effect of Padding That Increases Array Elements in the First Dimension (Compiler Option Tuning)

FUJITSU

You can obtain an effect equivalent to that of source tuning by specifying the following compiler options (Fortran-specific).

Compiler Option	Functional Description
-Karraypad_const[=*N*] (1≦*N*≦2,147,483,647)	Pads *N* elements in arrays whose 1st dimension has an explicit upper/lower bound that has a constant upper/lower bound expression. If *N* is not specified, the compiler decides the amount of padding for each target array. The purpose of padding is to create a gap in an array.
-Karraypad_expr=*N* (1≦*N*≦2,147,483,647)	Pads *N* elements in arrays whose 1st dimension has an explicit upper/lower bound regardless of whether the upper/lower bound expression is a constant expression.

■**Use example (for source before improvement)**

```
$ frtpx -Kfast,parallel sample.f90 -Karraypad_expr=1
```

Padding is applied to the automatically selected target arrays.

- **Note**
  - **These options must be specified for all source code using a target array.**
  - **The effect of padding varies depending on the program.**
  - **If not used correctly, computational results may differ.**
  - **The -Karraypad_const [=N] and -Karray_expr=N options cannot be specified at the same time.**

FUJITSU

Padding That Increases Array Elements in the Second Dimension

- Case of No Improvement by Padding That Increases Array Elements in the First Dimension
- Padding That Increases Array Elements in the Second Dimension
- Padding That Increases Array Elements in the Second Dimension (Before Improvement)
- Effect of Padding That Increases Array Elements in the Second Dimension (Source Tuning)

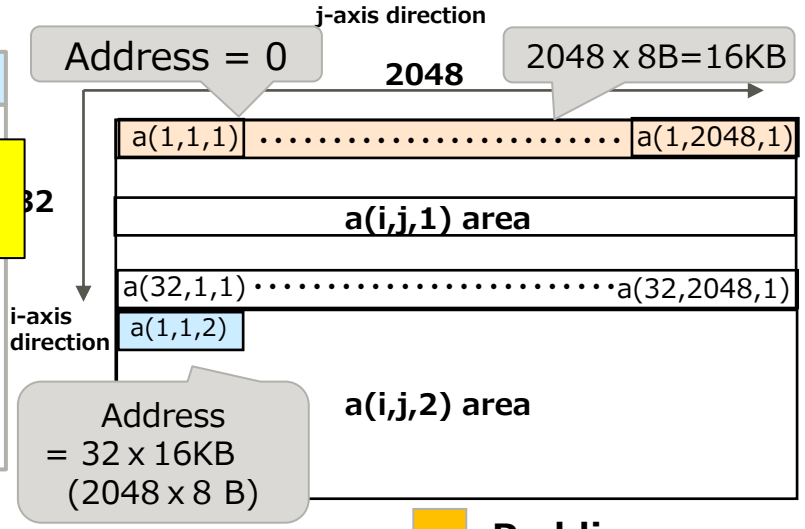
Case of No Improvement by Padding That Increases Array Elements in the First Dimension

Depending on the array size, adding padding (+1) to array elements in the first dimension may not result in improvement.

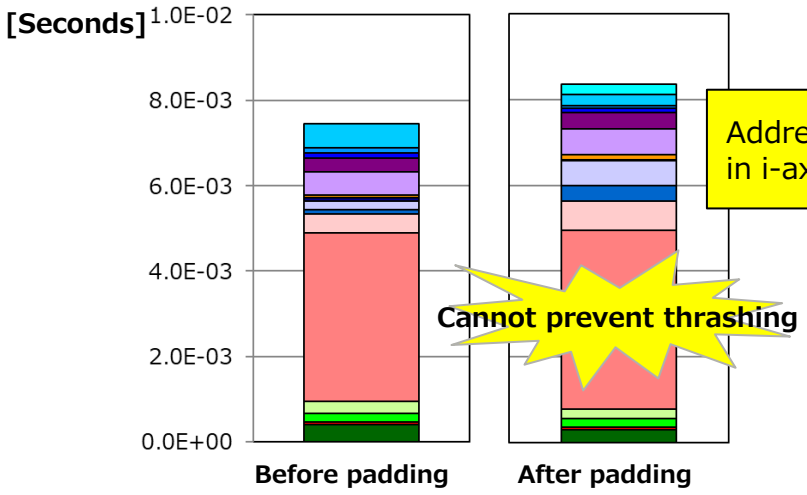
Source Before Improvement

```
parameter(k=32,l=2048)
real*8  a(k, l, 8)
common /com/a
do j = 1 , l
  do i = 1 , k
    a(i, j, 8) = a(i, j, 1) + a(i, j, 2) + a(i, j, 3) + &
                 a(i, j, 4) + a(i, j, 5) + a(i, j, 6) + a(i, j, 7)
  enddo
enddo
end
```

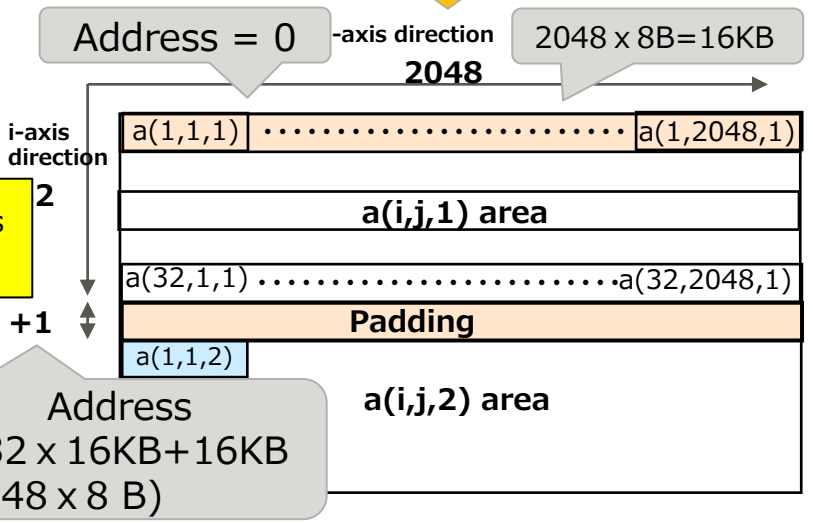
Address continuous in i-axis direction



	L1I miss rate (/Effective instruction)	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)
Before	0.00	1.04E+07	0.39	68.52
After	0.00	1.32E+07	0.52	75.20



Address continuous in i-axis direction



Thrashing occurs since Array a is still on 16 KB boundary

Padding That Increases Array Elements in the Second Dimension

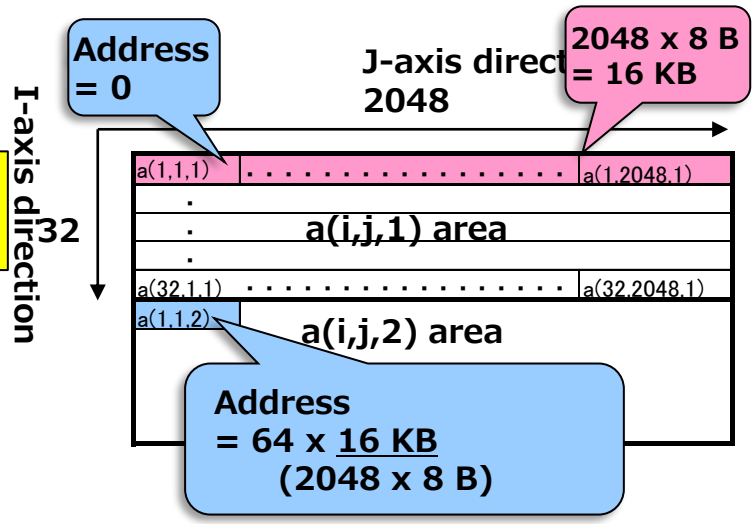
FUJITSU

L1D cache thrashing is prevented by adding padding (+1) to the second dimension to destroy 16 KB boundaries.

Source Before Improvement

```
33 parameter(k=32,l=2048)
34 real*8  a(k, l, 8)
35 common /com/a
36 do j = 1 , l
37   do i = 1 , k
38     a(i, j, 8) = a(i, j, 1) + a(i, j, 2) + a(i, j, 3) + a(i, j, 4) +
                    a(i, j, 5) + a(i, j, 6) + a(i, j, 7)
39   enddo
40 enddo
41 end
```

Address continuous in i-axis direction



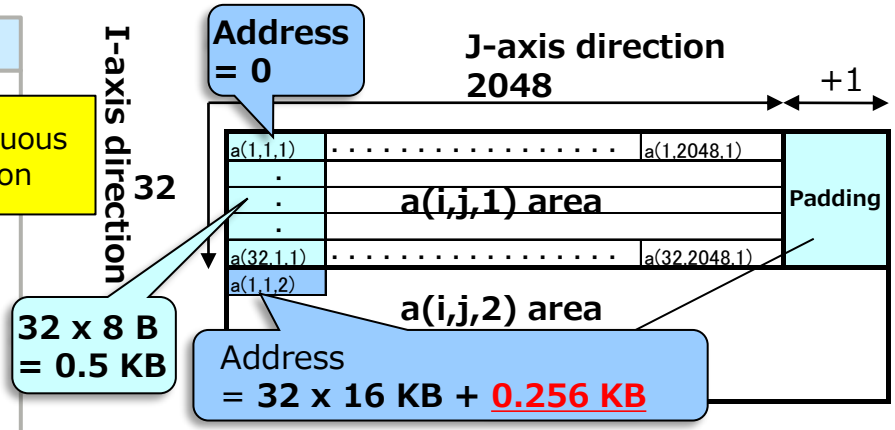
Thrashing occurs due to 16 KB boundary

Source After Improvement

```
33 parameter(k=32,l=2048)
34 real*8  a(k, l+1, 8)
35 common /com/a
36 do j = 1 , l
37   do i = 1 , k
38     a(i, j, 8) = a(i, j, 1) + a(i, j, 2) + a(i, j, 3) + a(i, j, 4) +
                    a(i, j, 5) + a(i, j, 6) + a(i, j, 7)
39   enddo
40 enddo
41 end
```

Address continuous in i-axis direction

Padding to 2nd dimension of Array a



Thrashing prevented because 16 KB boundary destroyed

Fortran Padding That Increases Array Elements in the Second Dimension (Before Improvement)

FUJITSU

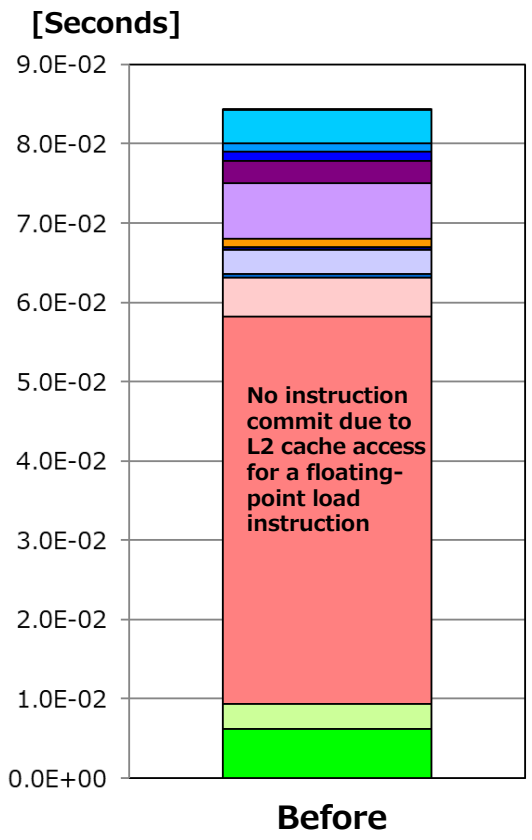
Each stream of Array a is on a 16 KB boundary. L1D cache thrashing occurs. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
38                parameter(n=32,m=2048)
39                real*8  a(n, m, 8)
40                common /com/a
                <<< Loop-information Sta
                <<<  [PARALLELIZATION]
                <<<    Standard iteration
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104,
                                           MVE: 7, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a
                <<< Loop-information  End >>>
41    1  pp   v      do j = 1 , m
                <<< Loop-information Sta
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<< Loop-information  End >>>
42    2  p           do i = 1 , n
43    2  p    v       a(i, j, 8) = a(i, j, 1) + a(i, j, 2) + a(i, j, 3) + &
44    2                    a(i, j, 4) + a(i, j, 5) + a(i, j, 6) + a(i, j, 7)
45    2  p    v      enddo
46    1  p          enddo
```

Array size
32 x 2048 x 8 B =
32 x 16 KB
(16 KB boundary)

Streams of same array



High L1D miss rates despite sequential array access
-> L1D cache thrashing occurs

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.88E+08	1.63E+08	0.42	67.76%	32.22%	0.01%	1.01E+04	0.00	71.17%	39.65%	0.00%

Fortran Effect of Padding That Increases Array Elements in the Second Dimension (Source Tuning)

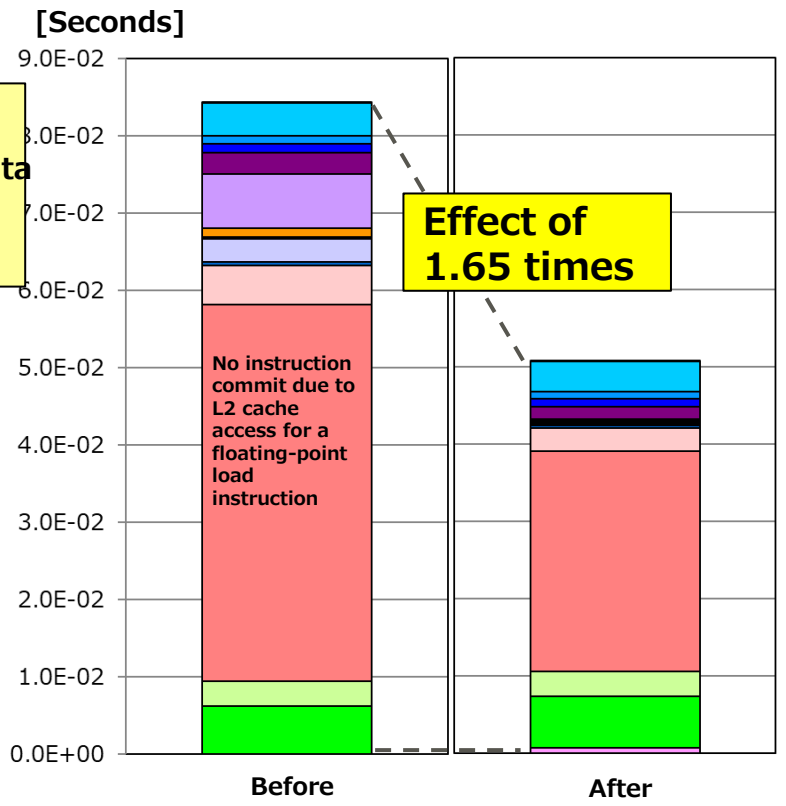
FUJITSU

Add padding (+1) to the second dimension of each stream of Array a to prevent L1D cache thrashing. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement

```
39              parameter(n=32,m=2048)
40              real*8  a(n, m+1, 8)
41              common /com/a
              <<< Loop-information Start >
              <<<  [PARALLELIZATION]
              <<<    Standard iteration cour
              <<<  [OPTIMIZATION]
              <<<    COLLAPSED
              <<<    SIMD(VL: 8)
              <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104,
                                         MVE: 7, POL: S)
              <<<    PREFETCH(HARD) Expected by compiler :
              <<<      a
              <<< Loop-information  End >>>
42    1  pp   v      do j = 1 , m
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    COLLAPSED
              <<< Loop-information  End >>>
43    2  p           do i = 1 , n
44    2  p    v       a(i, j, 8) = a(i, j, 1) + a(i, j, 2) + a(i, j, 3) + &
45    2                    a(i, j, 4) + a(i, j, 5) + a(i, j, 6) + a(i, j, 7)
46    2  p    v      enddo
47    1  p          enddo
```

1 added to m to shift data from 16 KB boundary



L1D misses reduced

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.88E+08	1.63E+08	0.42	67.76%	32.22%	0.01%	1.01E+04	0.00	71.17%	39.65%	0.00%
After	0.00	3.26E+08	1.07E+08	0.33	51.02%	48.98%	0.00%	9.13E+03	0.00	72.77%	33.09%	0.00%

C/C++ Padding That Increases Array Elements in the Second Dimension (Before Improvement)

FUJITSU

Each stream of Array a is on a 16 KB boundary. L1D cache thrashing occurs. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

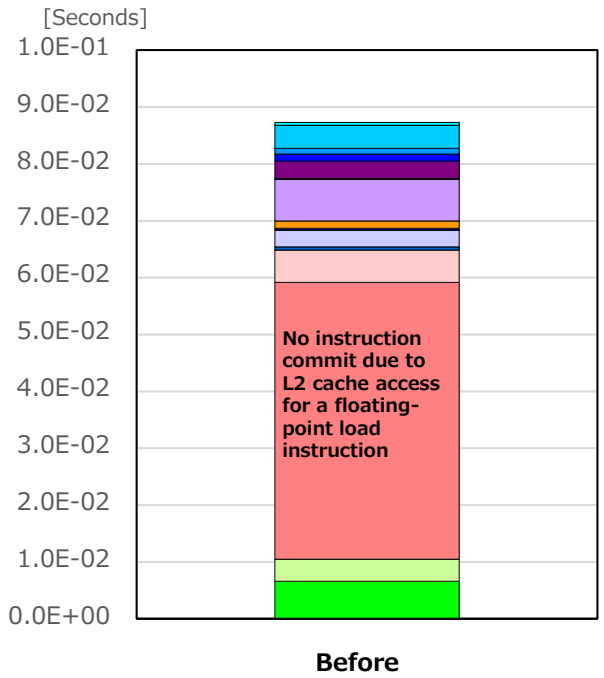
Source Before Improvement

```
30          void sub(void){
31            int i, j;
32
33          #pragma omp parallel for collapse(2)
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      a
           <<< Loop-information  End >>>
34  p     v    for (j = 0; j < m; j++){
35  p     v      for (i = 0; i < n; i++){
36  p     v        a[7][j][i] = a[0][j][i] + a[1][j][i] + a[2][j][i]
                             + a[3][j][i] + a[4][j][i] + a[5][j][i] + a[6][j][i];
37  p     v      }
38  p     v    }
39          }
```

Array declaration
double a[8][2048][32];

Array a size
32×2048×8B=
32×16 KB(16 KB boundary)

Streams of same array



High L1D miss rates despite sequential array access
-> L1D cache thrashing occurs

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.92E+08	1.66E+08	0.42	68.39%	31.61%	0.00%	2.04E+04	0.00	39.77%	72.33%	0.00%

C/C++ Effect of Padding That Increases Array Elements in the Second Dimension (Source Tuning)

FUJITSU

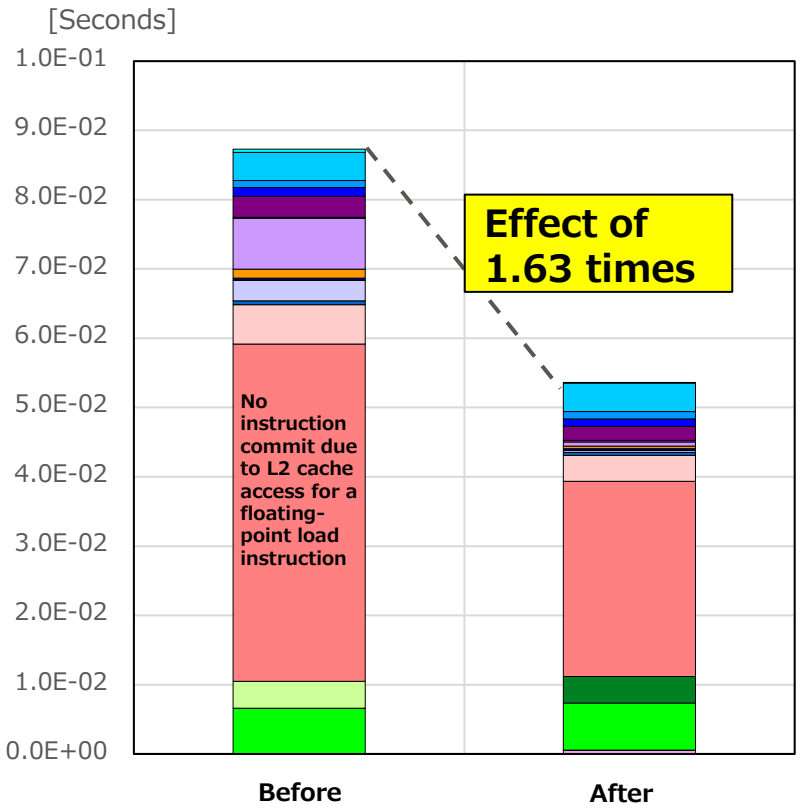
Add padding (+1) to the second dimension of each stream of Array a to prevent L1D cache thrashing. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement

```
30          void sub(void){
31            int i, j;
32
33          #pragma omp parallel for collapse(2)
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.(
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      a
            <<< Loop-information  End >>>
34  p     v    for (j = 0; j < m; j++){
35  p     v      for (i = 0; i < n; i++){
36  p     v        a[7][j][i] = a[0][j][i] + a[1][j][i] + a[2][j][i] + a[3][j][i]
                             + a[4][j][i] + a[5][j][i] + a[6][j][i];
37  p     v      }
38  p     v    }
39          }
```

Array declaration
double a[8][2049][32];

Shift from the 16 KB boundary by increasing (+1) the elements of the second dimension of array a.



L1D misses reduced

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.92E+08	1.66E+08	0.42	68.39%	31.61%	0.00%	2.04E+04	0.00	39.77%	72.33%	0.00%
After	0.00	3.44E+08	1.07E+08	0.31	51.12%	48.88%	0.00%	2.21E+04	0.00	19.24%	84.94%	0.00%

FUJITSU

Padding With Dummy Arrays

- Padding With Dummy Arrays (Before Improvement)
- Padding With Dummy Arrays (Source Tuning)
- Effect of Padding With Dummy Arrays (Compiler Option Tuning)

Fortran Padding With Dummy Arrays (Before Improvement)

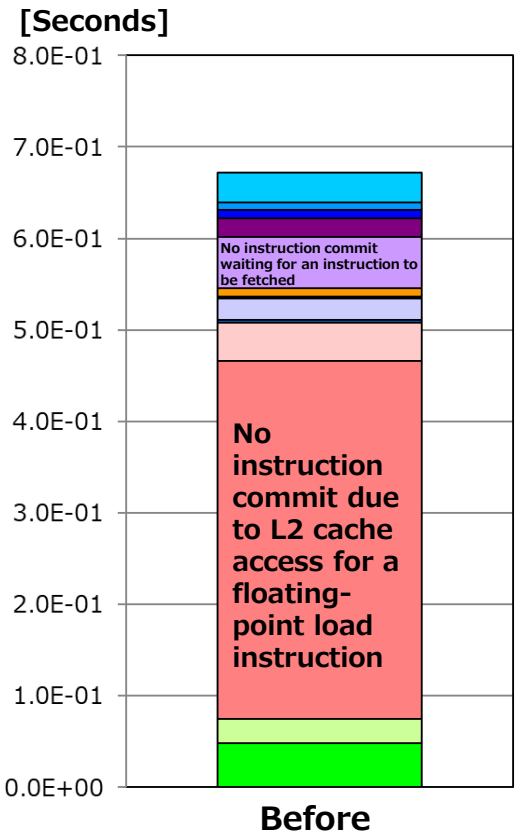
FUJITSU

Each array is on a 16 KB boundary. L1D cache thrashing occurs. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source After Improvement

```
1              parameter(n=256,m=256)
2              real*8  a(n, m),b(n,m),c(n,m),d(n,m),e(n,m),f(n,m),g(n,m),h(n,m)
3              character (1),parameter :: null0=z'00'
4              common /test/a,b,c,d,e,f,g,h
:
27   1  s   s       call sub()
:
34               subroutine sub()
35               parameter(n=256,m=256)
36               real*8  a(n, m),b(n,m),c(n,m),d(n,m),e(n,m),f(n,m),g(n,m),h(n,m)
37               common /test/a,b,c,d,e,f,g,h
              <<< Loop-information Start >>>
              <<<  [PARALLELIZATION]
              <<<    Standard iteration count: 433
              <<<  [OPTIMIZATION]
              <<<    COLLAPSED
              <<<    SIMD(VL: 8)
              <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7, POL: S)
              <<< Loop-information  End >>>
38   1  pp   v     do j = 1 , m
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    COLLAPSED
              <<< Loop-information  End >>>
39   2  p          do i = 1 , n
40   2  p   v        a(i, j) = b(i, j) + c(i, j) + d(i, j) + e(i, j) + f(i ,j) + g(i ,j) + h(i ,j)
41   2  p   v      enddo
42   1  p        enddo
```

Array size 256 x 256 x 8 B = 32 x 16 KB (16 KB boundary)



High L1D miss rates despite sequential array access -> L1D cache thrashing occurs

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.05E+09	1.30E+09	0.43	67.76%	32.24%	0.00%	8.35E+03	0.00	86.64%	19.76%	0.00%

Fortran Padding With Dummy Arrays (Source Tuning)

FUJITSU

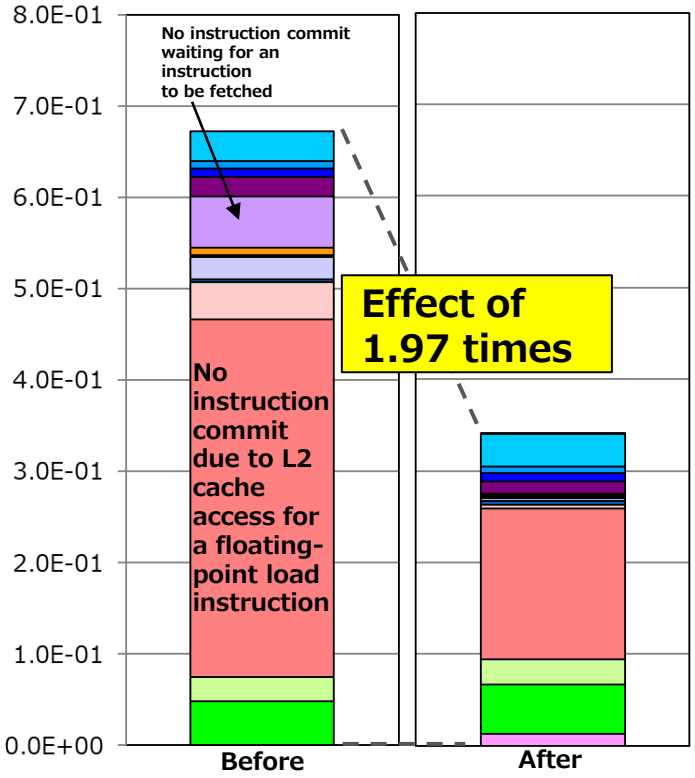
Shift arrays from 16 KB boundaries by adding dummy arrays between them to prevent L1D cache thrashing. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement

```
1                parameter(n=256,m=256)
2                real*8  a(n, m),dummy1(64),b(n,m),dummy2(64),&
                         c(n,m),dummy3(64),d(n,m),dummy4(64)
3                real*8  e(n, m),dummy5(64),f(n,m),dummy6(64),&
                         g(n,m),dummy7(64),h(n,m)
4                character (1),parameter :: null0='00'
:
28    1  s   s       call sub()
:
35               subroutine sub()
:
                 <<< Loop-information Start >>>
                 <<<  [PARALLELIZATION]
                 <<<    Standard iteration count: 433
                 <<<  [OPTIMIZATION]
                 <<<    COLLAPSED
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7)
                 <<< Loop-information  End >>>
40    1  pp   v     do j = 1 , m
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    COLLAPSED
                 <<< Loop-information  End >>>
41    2  p          do i = 1 , n
42    2  p   v        a(i, j) = b(i, j) + c(i, j) + d(i, j) + e(i, j) + f(i,j) + g(i,j) + h(i,j)
43    2  p   v      enddo
44    1  p        enddo
```

Arrays shifted from 16 KB boundaries by adding dummy arrays between them

[Seconds]



L1D miss and L1D miss dm rates improved

Cache	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)
Before	3.05E+09	1.30E+09	0.43	67.76%
After	2.79E+09	6.06E+08	0.22	31.03%

C/C++ Padding With Dummy Arrays (Before Improvement)

FUJITSU

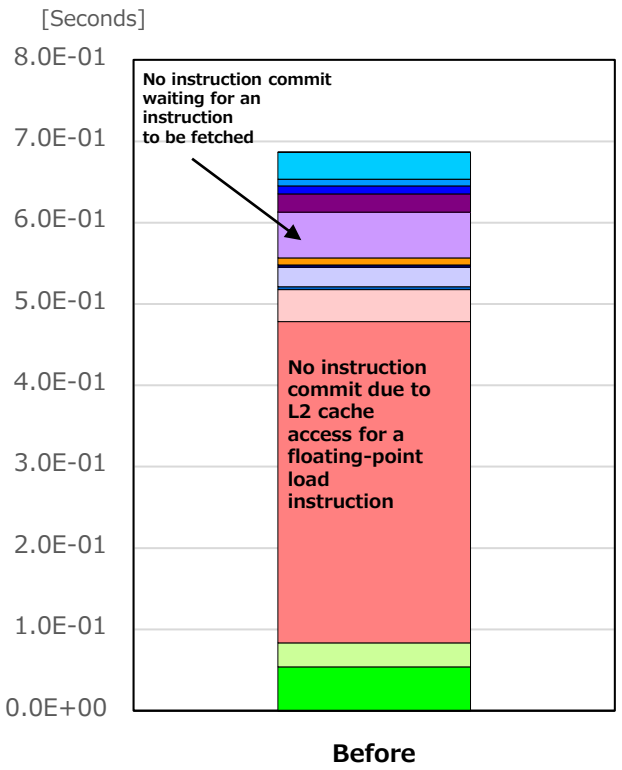
Each array is on a 16 KB boundary. L1D cache thrashing occurs. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source After Improvement

Array declaration
double a[256][256], b[256][256], c[256][256], d[256][256], e[256][256], f[256][256], g[256][256], h[256][256];

Array size 256×256×8B= 32×16 KB(16 KB boundary)

```
35          void sub(void){
36            int i, j;
37
38          #pragma omp parallel for collapse(2)
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      b, c, f, e, g, h, d, a
           <<< Loop-information  End >>>
39  p     v    for (j = 0; j < m; j++){
40  p     v      for (i = 0; i < n; i++){
41  p     v        a[j][i] = b[j][i] + c[j][i] + d[j][i] + e[j][i] + f[j][i] + g[j][i] + h[j][i];
42  p     v      }
43  p     v    }
44          }
```



High L1D miss rates despite sequential array access
-> L1D cache thrashing occurs

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.12E+09	1.33E+09	0.43	68.30%	31.70%	0.00%	2.34E+04	0.00	46.43%	66.98%	0.00%

Padding With Dummy Arrays (Source Tuning)

FUJITSU

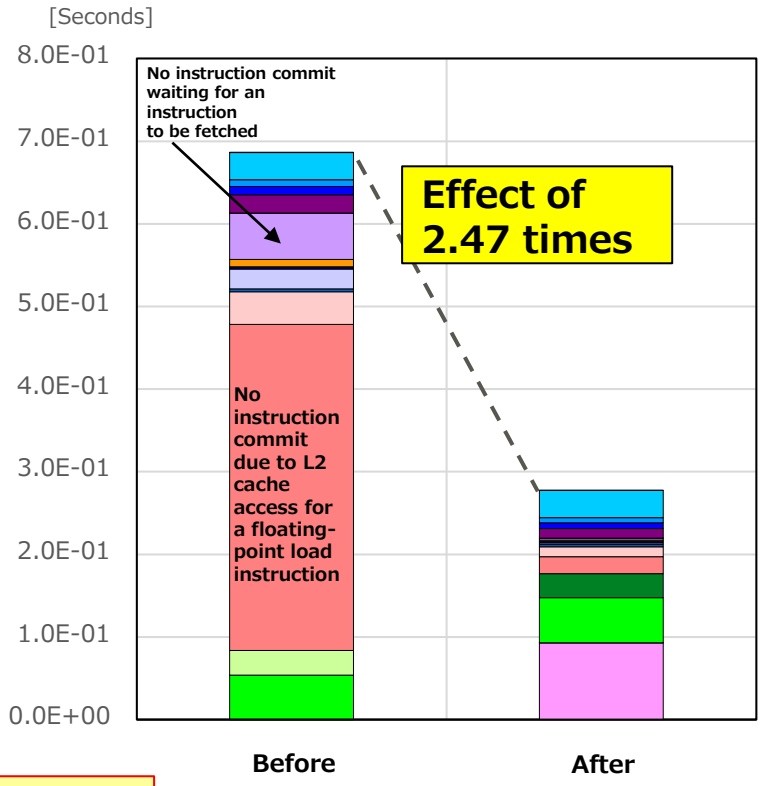
Shift arrays from 16 KB boundaries by adding dummy arrays between them to prevent L1D cache thrashing. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement

```
36          void sub(void){
37            int i, j;
38
39          #pragma omp parallel for colla
           <<< Loop-information Start >>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(
           <<<    PREFETCH(HARD) Expec
           <<<      b, c, f, e, g, h, d, a
           <<< Loop-information  End >>
40  p     v    for (j = 0; j < m; j++){
41  p     v      for (i = 0; i < n; i++){
42  p     v        a[j][i] = b[j][i] + c[j][i] + d[j][i] + e[j][i] + f[j][i] + g[j][i] + h[j][i];
43  p     v      }
44  p     v    }
```

Shift from 16 KB boundary by adding a dummy array declaration of double type with 64 elements (e.g., dummy[64]) between each of arrays a, b, c, d, e, f, g, h.

Array declaration
double a[256][256], dummy1[64], b[256][256], dummy2[64], c[256][256], dummy3[64], d[256][256], dummy4[64], e[256][256], dummy5[64], f[256][256], dummy6[64], g[256][256], dummy7[64], h[256][256];



L1D miss and L1D miss dm rates improved

Cache	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)
Before	3.12E+09	1.33E+09	0.43	68.30%
After	2.67E+09	4.56E+08	0.17	8.37%

Fortran Effect of Padding With Dummy Arrays (Compiler Option Tuning)

FUJITSU

You can obtain an effect equivalent to that of source tuning by specifying the following compiler option (Fortran-specific).

Compiler Option	Functional Description
-Kcommonpad[=*N*] (4≦*N*≦2,147,483,644)	Specifies that a gap be placed between areas of variables in the common block to improve data cache use efficiency. If *N* is not specified, the compiler automatically decides the best value.

■Use example (for source before improvement)

```
$ frtpx -Kfast,parallel sample.f90 -Kcommonpad=512
```

- Automatically selecting target arrays -> Applying padding
- Note
  - To compile separately when you specify the compiler option -Kcommonpad for a file containing a common block, you need to also specify it for other files containing the common block of the same name.
  - To compile with the compiler option -Kcommonpad=*N* specified for multiple files, the value of *N* must be the same.
  - If you use the same common block name but change its elements when specifying the compiler option -Kcommonpad, the program may not run properly.

FUJITSU

Padding With Dummy Arrays (Arrays of Different Sizes)

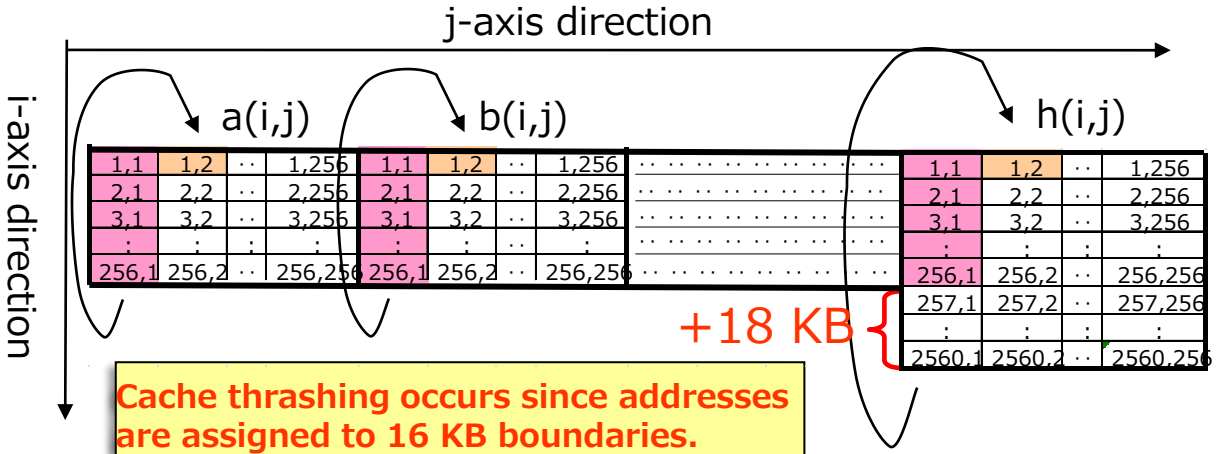
- Conflict Between Arrays of Different Sizes
- Padding With Dummy Arrays (Arrays of Different Sizes: Before Improvement)
- Padding With Dummy Arrays (Arrays of Different Sizes: Source Tuning)

Conflict Between Arrays of Different Sizes (1/2)

In general, cache thrashing does not regularly occur with arrays of different sizes.

Example of Source

```
parameter(n=256,m=256)
parameter(k=2560,l=256)

real*8  a(n,m), b(n,m), c(n,m),
        d(n,m), e(n,m), f(n,m),
        g(n,m), h(k,l)

common /test/a,b,c,d,e,f,g,h
do j = 1 , m
  do i = 1 , n
    a(i, j) = b(i, j) + c(i, j) + d(i, j) +
              e(i, j) + f(i, j) + g(i, j) +
              h(i, j)
  enddo
enddo
```



Cache thrashing occurs since addresses are assigned to 16 KB boundaries.

If address of Array a(1,1) is 0, then:
address of Array a(1,1) is 0 (16 KB x 0)
address of Array b(1,1) is 256 x 256 x 8 (16 KB x 32)
: : :
address of Array h(1,1) is 256 x 256 x 8 x 7 (16 KB x 224)

Counting up in second dimension:
address of Array a(1,2) is address of a(1,1) + 256 x 8 B
address of Array b(1,2) is address of b(1,1) + 256 x 8 B
: : :
address of Array h(1,2) is address of h(1,1) + 2560 x 8 B

256 x 8 B + 18 KB

In general, cache thrashing does not regularly occur with arrays of different sizes.

Arrays a, b, c, d, e, f, and g remain on 16 KB boundaries, but the address of Array h is not on a 16 KB boundary.

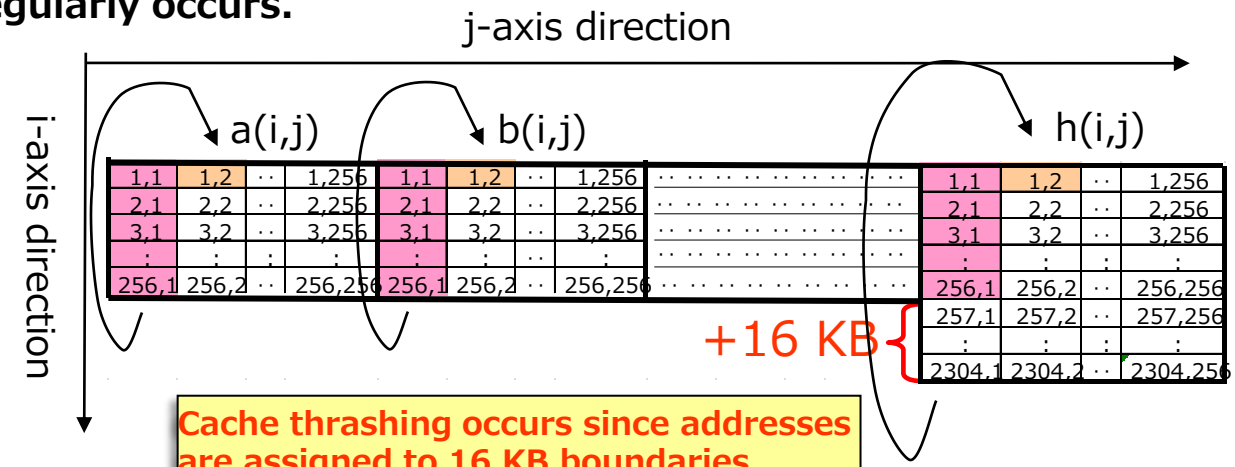
Conflict Between Arrays of Different Sizes (2/2)

FUJITSU

Even arrays of different sizes may remain on 16 KB boundaries, depending on the array size. In that case, cache thrashing regularly occurs.

Example of Source

```
parameter(n=256,m=256)
parameter(k=2304,l=256)

real*8  a(n,m), b(n,m), c(n,m),
        d(n,m), e(n,m), f(n,m),
        g(n,m), h(k,l)

common /test/a,b,c,d,e,f,g,h
do j = 1 , m
  do i = 1 , n
    a(i, j) = b(i, j) + c(i, j) + d(i, j) +
              e(i, j) + f(i, j) + g(i, j) +
              h(i, j)
  enddo
enddo
```



Cache thrashing occurs since addresses are assigned to 16 KB boundaries.

If address of Array a(1,1) is 0, then:
address of Array a(1,1) is 0 (16 KB x 0)
address of Array b(1,1) is 256 x 256 x 8 (16 KB x 32)
: : :
address of Array h(1,1) is 256 x 256 x 8 x 7 (16 KB x 224)

Counting up in second dimension:
address of Array a(1,2) is address of a(1,1) + 256 x 8 B
address of Array b(1,2) is address of b(1,1) + 256 x 8 B
: : :
address of Array h(1,2) is address of h(1,1) + 2304 x 8 B

256 x 8 B + 16 KB

Measures against thrashing are required for arrays of all sizes, including Array h.

Arrays a, b, c, d, e, f, g, and h all remain on 16 KB boundaries.

Fortran Padding With Dummy Arrays (Arrays of Different Sizes: Before Improvement)

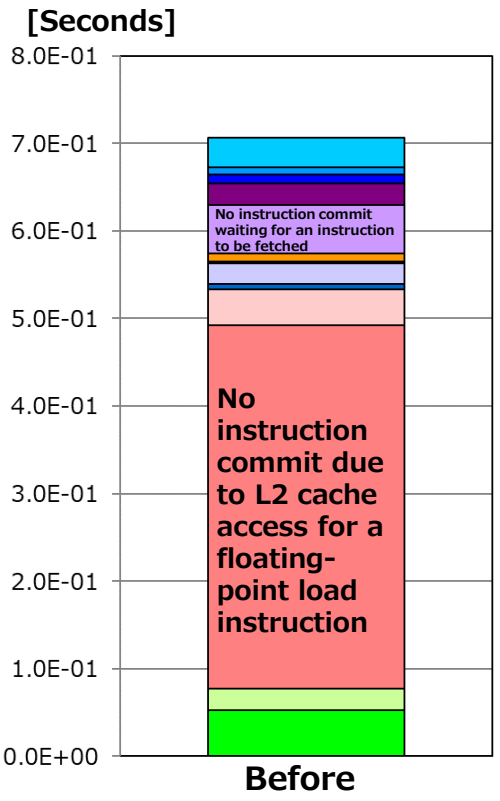
FUJITSU

Each array is on a 16 KB boundary. L1D cache thrashing occurs. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
52                integer k,l,n,m
53                parameter(n=256,m=256)
54                parameter(k=2304,l=256)
55
56                real*8  a(n,m), b(n,m), c(n,m), d(n,m), &
                  e(n,m), f(n,m), g(n,m), h(k,l)

57                common /test/a,b,c,d,e,
                <<< Loop-information Start
                <<<  [PARALLELIZATION]
                <<<    Standard iteration co
                <<< Loop-information  End >>>
58    1  pp           do j = 1 , m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.83, ITR: 112,
                                                MVE: 13, POL: S)
                <<< Loop-information  End >>>
59    2  p    v        do i = 1 , n
60    2  p    v          a(i, j) = b(i, j) + c(i, j) + d(i, j) + e(i, j) +
                              f(i, j) + g(i, j) + h(i, j)
61    2  p    v        enddo
62    1  p           enddo
```

Interval between arrays remains 16 KB even after counting up in second dimension



High L1D miss rates despite sequential array access -> L1D cache thrashing occurs

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.00E+09	1.39E+09	0.46	70.14%	29.86%	0.00%	3.34E+04	0.00	34.30%	77.94%	0.00%

Fortran Padding With Dummy Arrays (Arrays of Different Sizes: Source Tuning)

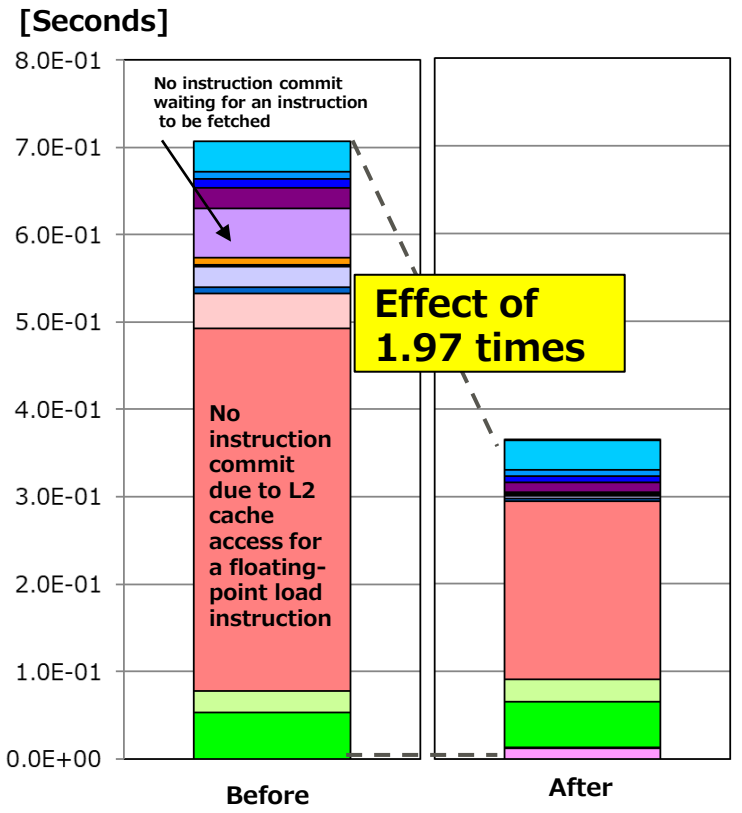
FUJITSU

Shift arrays from 16 KB boundaries by adding dummy arrays them to prevent L1D cache thrashing. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement

```
52              integer k,l,n,m
53              parameter(n=256,m=256)
54              parameter(k=2304,l=256)
55
56              real*8  a(n,m),dummy1(64),b(n,m),dummy2(64), &
57                    c(n,m),dummy3(64),d(n,m),dummy4(64), &
58                    e(n,m),dummy5(64),f(n,m),dummy6(64), &
59                    g(n,m),dummy7(64),h(k,l)
60              common /test/a,dummy1,b,dummy2,c,dummy3,d,dummy4,e,
                  dummy5,f,dummy6,g,dummy7,h
61
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 2
                <<< Loop-information  End >>>
62    1  pp         do j = 1 , m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.83, ITR: 112, MVE: 13, POL: S)
                <<< Loop-information  End >>>
63    2  p   v       do i = 1 , n
64    2  p   v         a(i, j) = b(i, j) + c(i, j) + d(i, j) + e(i, j) + f(i, j) + g(i, j) + h(i, j)
65    2  p   v       enddo
66    1  p         enddo
```

Dummy arrays placed between arrays



L1D miss and L1D miss dm rates improved

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.00E+09	1.39E+09	0.46	70.14%	29.86%	0.00%	3.34E+04	0.00	34.30%	77.94%	0.00%
After	0.00	2.57E+09	6.90E+08	0.27	38.70%	61.31%	0.00%	2.54E+04	0.00	29.53%	80.62%	0.00%

C/C++ Padding With Dummy Arrays (Arrays of Different Sizes: Before Improvement)

FUJITSU

Each array is on a 16 KB boundary. L1D cache thrashing occurs. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

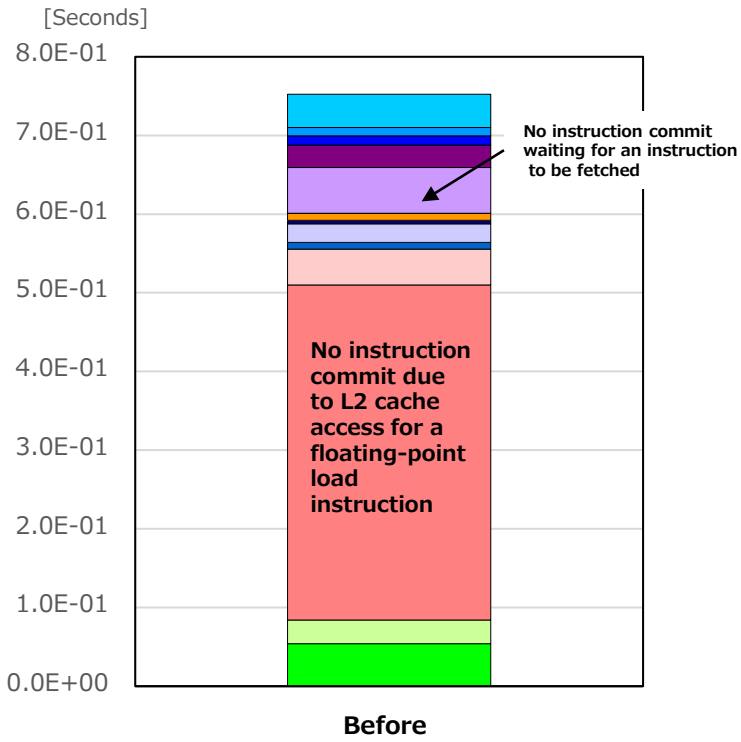
Source Before Improvement

```
24          void sub(void){
25            int i, j;
26
27          #pragma omp parallel for
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      b, c, f, e, g, h, d, a
            <<< Loop-information  End >>>
28  p         for (j = 0; j < m; j++){
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 13, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      b, c, f, e, g, h, d, a
            <<< Loop-information  End >>>
29  p     v     for (i = 0; i < n; i++){
30  p     v       a[j][i] = b[j][i] + c[j][i] + d[j][i] + e[j][i] + f[j][i] + g[j][i] + h[j][i];
31  p     v     }
32  p         }
33          }
```

Array declaration
double a[256][256], b[256][256], c[256][256], d[256][256], e[256][256], f[256][256], g[256][256], h[256][2304];

Array a size: 256×256×8B= 32×16 KB(16 KB boundary)

Interval between arrays remains 16 KB even after counting up in second dimension



High L1D miss rates despite sequential array access -> L1D cache thrashing occurs

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.62E+09	1.38E+09	0.38	71.21%	28.78%	0.01%	1.04E+05	0.00	91.27%	10.77%	0.00%

C/C++ Padding With Dummy Arrays (Arrays of Different Sizes: Source Tuning)

FUJITSU

Shift arrays from 16 KB boundaries by adding dummy arrays them to prevent L1D cache thrashing. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

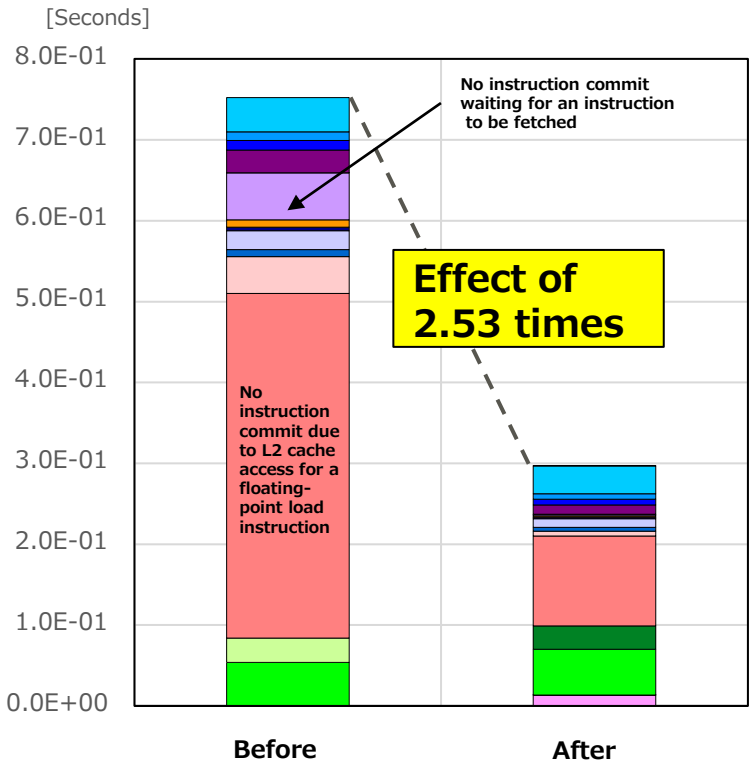
Source After Improvement

```
25          void sub(void){
26            int i, j;
27
28          #pragma omp parallel for
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expected
            <<<      b, c, f, e, g, h, d, a
            <<< Loop-information  End >>>
29  p         for (j = 0; j < m; j++){
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC
            <<<    PREFETCH(HARD) Expected
            <<<      b, c, f, e, g, h, d, a
            <<< Loop-information  End >>>
30  p     v     for (i = 0; i < n; i++){
31  p     v       a[j][i] = b[j][i] + c[j][i] + d[j][i] + e[j][i] + f[j][i] + g[j][i] + h[j][i];
32  p     v     }
33  p         }
34          }
```

Shift from 16 KB boundary by adding a dummy array declaration of double type with 64 elements (e.g., dummy[64]) between each of arrays a, b, c, d, e, f, g, h.

配列宣言

```
double a[256][256],
dummy1[64], b[256][256],
dummy2[64], c[256][256],
dummy3[64], d[256][256],
dummy4[64], e[256][256],
dummy5[64], f[256][256],
dummy6[64], g[256][256],
dummy7[64], h[256][2304];
```



L1D miss and L1D miss dm rates improved

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.62E+09	1.38E+09	0.38	71.21%	28.78%	0.01%	1.04E+05	0.00	91.27%	10.77%	0.00%
After	0.00	2.69E+09	4.81E+08	0.18	17.67%	82.33%	0.00%	1.20E+05	0.00	54.33%	62.57%	0.00%

FUJITSU

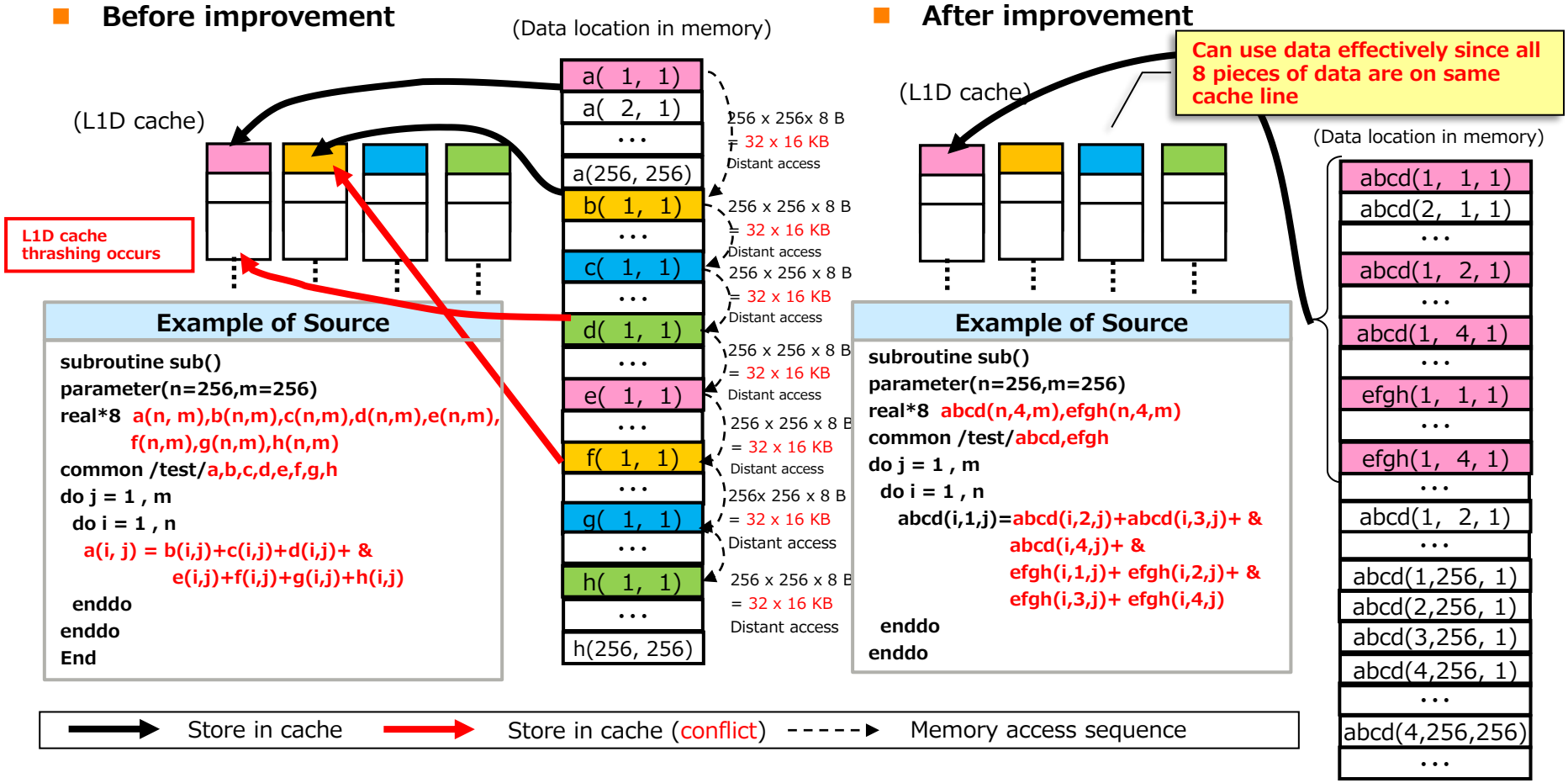
Array Merge (Improved Thrashing)

- What is Array Merge?
- Array Merge (Improved Thrashing) (Before Improvement)
- Array Merge (Improved Thrashing) (Source Tuning)
- Array Merge (Compiler Option Tuning)

What is Array Merge?

Array merge is a tuning method that merges multiple arrays into one.

As shown in the following example, you can store data on the same cache line by reducing the number of arrays.



Fortran Array Merge (Improved Thrashing) (Before Improvement)

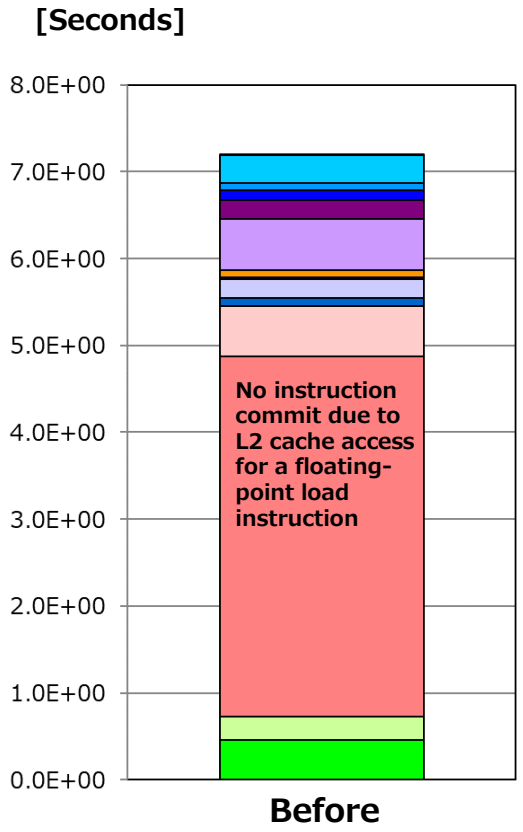
FUJITSU

Each array is on a 16 KB boundary. L1D cache thrashing occurs. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
40              parameter(n=256,m=256)
41              real*8  a(n, m),b(n,m),c(n,m),d(n,m),&
                        e(n,m),f(n,m),g(n,m),h(n,m)
42              common /test/a,b,c,d,e,f,g,h
              <<< Loop-information Start >>>
              <<<  [PARALLELIZATION]
              <<<    Standard iteration count: 433
              <<<  [OPTIMIZATION]
              <<<    COLLAPSED
              <<<    SIMD(VL: 8)
              <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104,
                                         MVE: 7, POL: S)
              <<<    PREFETCH(HARD) Expected by compiler :
              <<<      b, c, f, e, g, h, d, a
              <<< Loop-information  End >>>
43    1  pp   v     do j = 1 , m
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    COLLAPSED
              <<< Loop-information  End >>>
44    2  p          do i = 1 , n
45    2  p   v        a(i,j)=b(i,j)+c(i,j)+d(i,j)+e(i,j)+f(i,j)+g(i,j)+h(i,j)
46    2  p   v      enddo
47    1  p        enddo
```

Array size
256 x 256 x 8 B =
32 x 16 KB
(16 KB boundary)



High L1D miss rates despite sequential array access
-> L1D cache thrashing occurs

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.40E+10	1.33E+10	0.39	68.21%	31.79%	0.00%	1.12E+04	0.00	72.56%	37.92%	0.00%

Fortran Array Merge (Improved Thrashing) (Source Tuning)

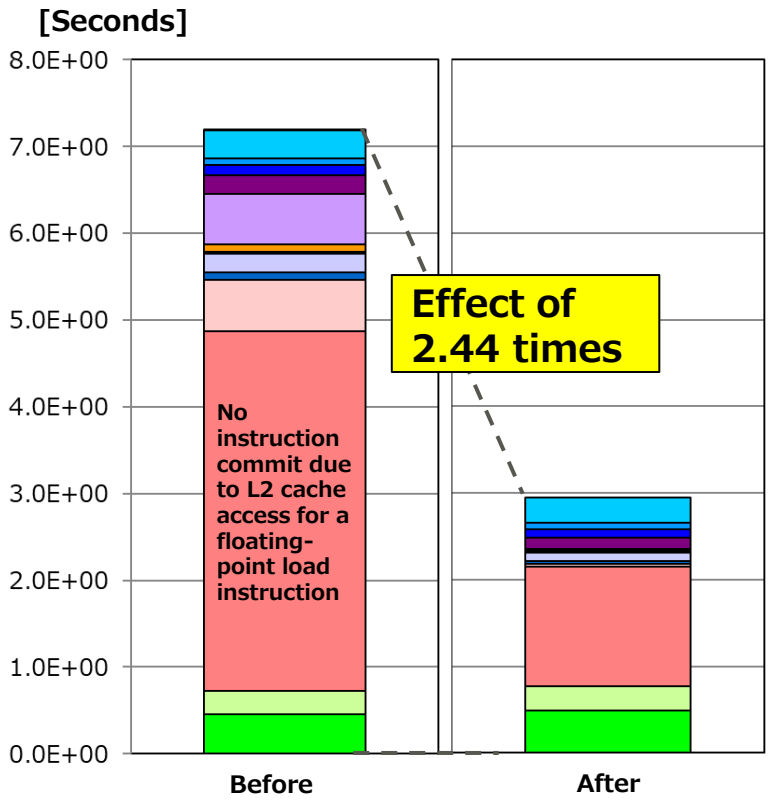
FUJITSU

Array merge reduces the number of streams from 8 to 2, preventing L1D cache thrashing. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement (Source Tuning)

```
44                parameter(n=256,m=256)
45                real*8  abcd(n,4,m),efgh(n,4,m)
46                common /test/abcd,efgh
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 2
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      efgh, abcd
                <<< Loop-information  End >>>
47    1  pp         do j = 1,m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.28, ITR: 112,
                                           MVE: 13, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      efgh, abcd
                <<< Loop-information  End >>>
48    2  p    v       do i = 1,n
49    2  p    v         abcd(i,1,j)=abcd(i,2,j)+abcd(i,3,j)+abcd(i,4,j)+&
50    2                      efgh(i,1,j)+efgh(i,2,j)+efgh(i,3,j)+efgh(i,4,j)
51    2  p    v       enddo
52    1  p         enddo
```

8 arrays merged into 2 (4 arrays each)



L1D misses reduced

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.40E+10	1.33E+10	0.39	68.21%	31.79%	0.00%	1.12E+04	0.00	72.56%	37.92%	0.00%
After	0.00	2.54E+10	4.40E+09	0.17	84.98%	15.03%	0.00%	1.70E+04	0.00	69.93%	46.61%	0.00%

C/C++ Array Merge (Improved Thrashing) (Before Improvement)

FUJITSU

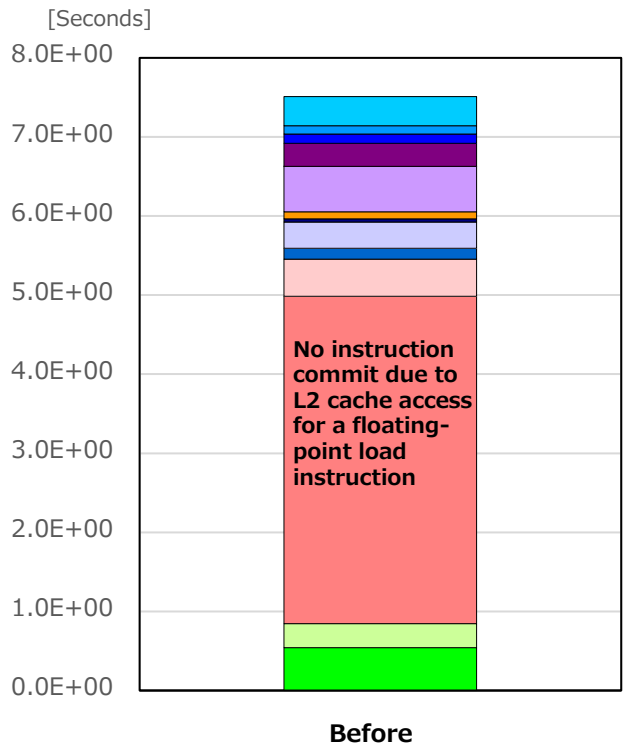
Each array is on a 16 KB boundary. L1D cache thrashing occurs. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
31          void sub(void)
32          {
33             int i,j;
34          #pragma omp parallel for
        <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected b
          <<<      b, c, f, e, g, h, d, a
          <<< Loop-information  End >>>
35  p           for(j=0;j<m;j++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      b, c, f, e, g, h, d, a
          <<< Loop-information  End >>>
36  p    v       for(i=0;i<n;i++){
37  p    v         a[j][i]=b[j][i]+c[j][i]+d[j][i]+e[j][i]+f[j][i]+g[j][i]+h[j][i];
38  p    v       }
39  p          }
40          }
```

Array declaration
double a[256][256], b[256][256], c[256][256], d[256][256], e[256][256], f[256][256], g[256][256], h[256][256];

Array a size: 256×256×8B=
32×16 KB(16 KB boundary)



High L1D miss rates despite sequential array access
-> L1D cache thrashing occurs

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.32E+10	1.36E+10	0.41	69.48%	30.52%	0.00%	1.16E+05	0.00	86.19%	26.53%	0.00%

C/C++ Array Merge (Improved Thrashing) (Source Tuning)

FUJITSU

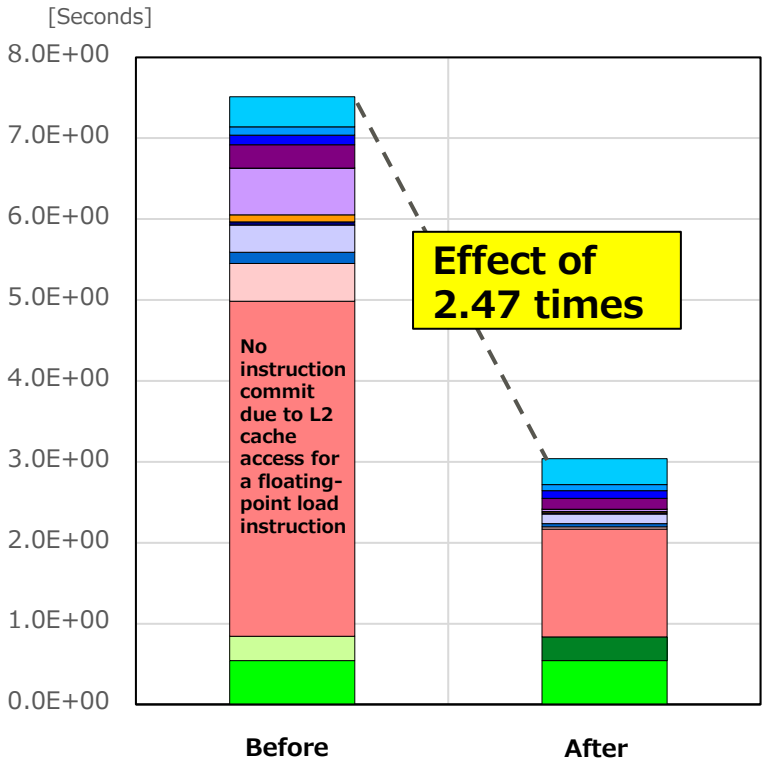
Array merge reduces the number of streams from 8 to 2, preventing L1D cache thrashing. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement (Source Tuning)

```
31          void sub(void)
32          {
33            int i,j;
34          #pragma omp parallel for
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      efgh, abcd
          <<< Loop-information  End >>>
35  p         for(j=0;j<m;j++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.28, ITR: 112, MVE: 8, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      efgh, abcd
          <<< Loop-information  End >>>
36  p    v       for(i=0;i<n;i++){
37  p    v         abcd[j][0][i]=abcd[j][1][i]+abcd[j][2][i]+abcd[j][3][i]+
                   efgh[j][0][i]+efgh[j][1][i]+efgh[j][2][i]+efgh[j][3][i];
38  p    v       }
39  p          }
40          }
```

Array declaration
double abcd[256][4][256], efgh[256][4][256];

8 arrays merged into 2 (4 arrays each)



L1D misses reduced

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.32E+10	1.36E+10	0.41	69.48%	30.52%	0.00%	1.16E+05	0.00	86.19%	26.53%	0.00%
After	0.00	2.63E+10	4.40E+09	0.17	90.03%	9.98%	-0.01%	4.45E+04	0.00	80.52%	35.47%	0.00%

Array Merge (Compiler Option Tuning)

FUJITSU

You can obtain an effect equivalent to that of source tuning by specifying the following compiler options (Fortran-specific).

Compiler Option	Functional Description
-Karray_merge_common [=name]	Specifies the merging of multiple arrays in a common block. You can specify the common block name in *name*. If *name* is not specified, the arrays in all the common blocks with names are subject to this option.
-Karray_merge_local	Specifies the merging of multiple local arrays. -Karray_merge_local_size=1000000 is also enabled at the same time.
-Karray_merge_local_size=N (2≦*N*≦2,147,483,647)	Specifies *N* or more bytes as the size of local arrays merged. This option is valid when the -Karray_merge_local option is enabled.
-Karray_merge	This option is equivalent to specifying the -Karray_merge_local and -Karray_merge_common options.

- Use example (for source before improvement)

$frtpx –Kfast,parallel sample.f90 –Karray_merge_common

- Note
  - These options must be specified in all source code using a target array.
  - The effect of the merge varies depending on the program.
  - If not used correctly, computational results may vary.
  - They cannot be used with a debug option (-g or -H).

FUJITSU

Loop Fission (Improved Thrashing)

- What is Loop Fission?
- Loop Fission (Improved Thrashing) (Before Improvement)
- Effect of Loop Fission (Improved Thrashing) (Source Tuning)
- Effect of Loop Fission (Optimization Control Line Tuning)

What is Loop Fission?

FUJITSU

- **Loop fission is a means to split a loop into multiple smaller loops mainly for the following purposes:**
  - To facilitate software pipelining
  - To improve cache memory use efficiency
  - To eliminate a register shortage

Loop fission reduces the number of arrays accessed in a loop, and thus may be able to facilitate software pipelining and prevent cache thrashing.

However, note that efficient use of data in the cache may no longer be possible, depending on how the loop is split.

Source Before Improvement

```
parameter(n=65536)
real*8  a(n),b(n),c(n),d(n),e(n),f(n),
g(n),h(n)
common /com/a,b,c,d,e,f,g,h
do i=1,n
   a(i) =  s / b(i)
   c(i) =  s / d(i)
   e(i) =  s / f(i)
   g(i) =  s / h(i)
enddo
```

Cache thrashing occur

Source After Improvement

```
parameter(n=65536)
real*8  a(n),b(n),c(n),d(n),e(n),f(n),
g(n),h(n)
common /com/a,b,c,d,e,f,g,h
!OCL LOOP_NOFUSION
do i=1,n
   a(i) =  s / b(i)
   c(i) =  s / d(i)
enddo
do i=1,n
   e(i) =  s / f(i)
   g(i) =  s / h(i)
enddo
```

Loop fusion suppressed

Loop fission
Suppress cache thrashing

Fortran Loop Fission (Improved Thrashing) (Before Improvement)

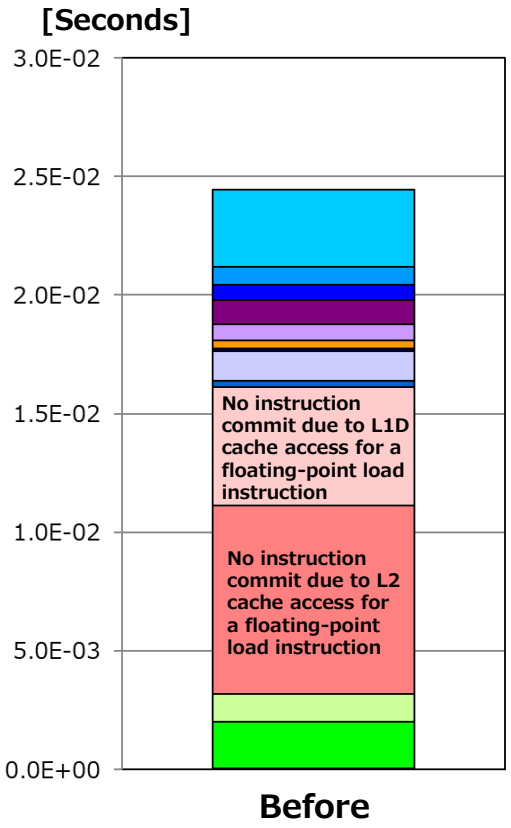
FUJITSU

Each array is on a 16 KB boundary. L1D cache thrashing occurs. Consequently, the "No instruction commit due to access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
46                parameter(n=65536)
47                real*8  a(n),b(n),c(n),d(n),e(n),f(n),g(n),h(n)
48                common /com/a,b,c,d,e,f,g,h
49
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count:
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 1.90, ITR: 56,
                                           MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      f, d, b, h, g, e, c, a
                <<< Loop-information  End >>>
50    1  pp   v       do i=1,n
51    1   p   v         a(i) =  s / b(i)
52    1   p   v         c(i) =  s / d(i)
53    1   p   v         e(i) =  s / f(i)
54    1   p   v         g(i) =  s / h(i)
55    1   p   v       enddo
```

Array size 65536 x 8 B = 32 x 16 KB (16 KB boundary)



High L1D miss and L1 miss dm rates despite sequential array access
-> L1D cache thrashing occurs

Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	prefetch rate (%) (/L1D miss)	L2 miss	(/Load-store instruction)	demand rate (%) (/L2 miss)	prefetch rate (%) (/L2 miss)	prefetch rate (%) (/L2 miss)
Before	0.00	1.25E+08	3.85E+07	0.31	57.48%	42.52%	0.01%	1.84E+04	0.00	31.49%	72.74%	0.00%

Fortran Effect of Loop Fission (Improved Thrashing) (Source Tuning)

FUJITSU

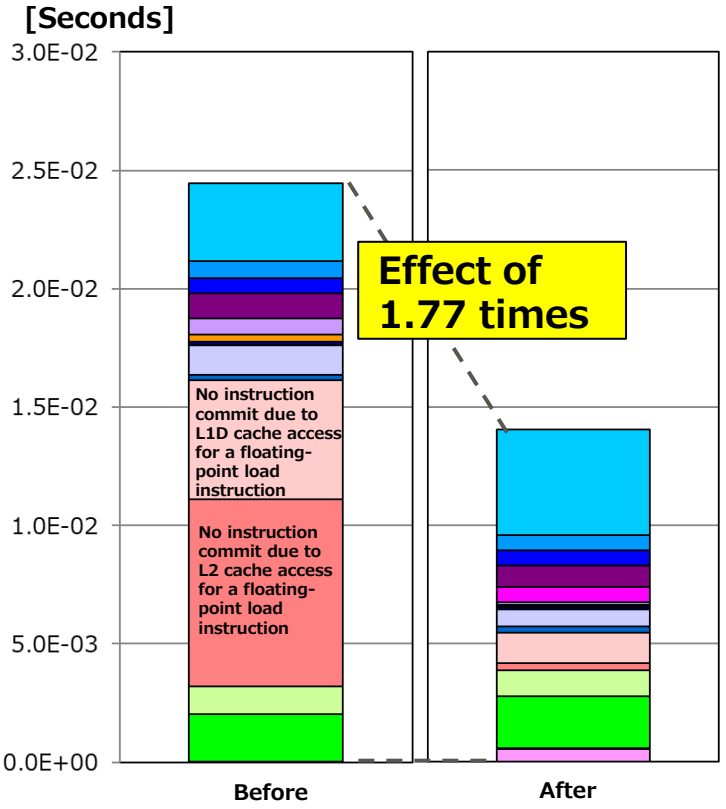
Loop fission reduces the number of streams from 8 to 4, preventing L1D cache thrashing. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement

```
46                parameter(n=65536)
47                real*8  a(n),b(n),c(n),d(n),e(n),f(n),g(n),h(n)
48                common /com/a,b,c,d,e,f,g,h
49
50              !OCL LOOP_NOFUSION
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration cou
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.36, ITR: 80,
                                           MVE: 2, POL: S)
                <<< Loop-information  End >>>
51    1  pp  2v       do i=1,n
52    1   p  2v         a(i) =  s / b(i)
53    1   p  2v         c(i) =  s / d(i)
54    1   p  2v       enddo
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 411
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.45, ITR: 80,
                                           MVE: 2, POL: S)
                <<< Loop-information  End >>>
55    1  pp  2v       do i=1,n
56    1   p  2v         e(i) =  s / f(i)
57    1   p  2v         g(i) =  s / h(i)
58    1   p  2v       enddo
```

Loop fusion suppressed

Loop fission



L1D miss and L1D miss dm rates reduced

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
Before	0.00	1.25E+08	3.85E+07	0.31	57.48%	42.52%	0.01%
After	0.00	1.10E+08	1.78E+07	0.16	8.37%	91.63%	0.00%

C/C++ Loop Fission (Improved Thrashing) (Before Improvement)

FUJITSU

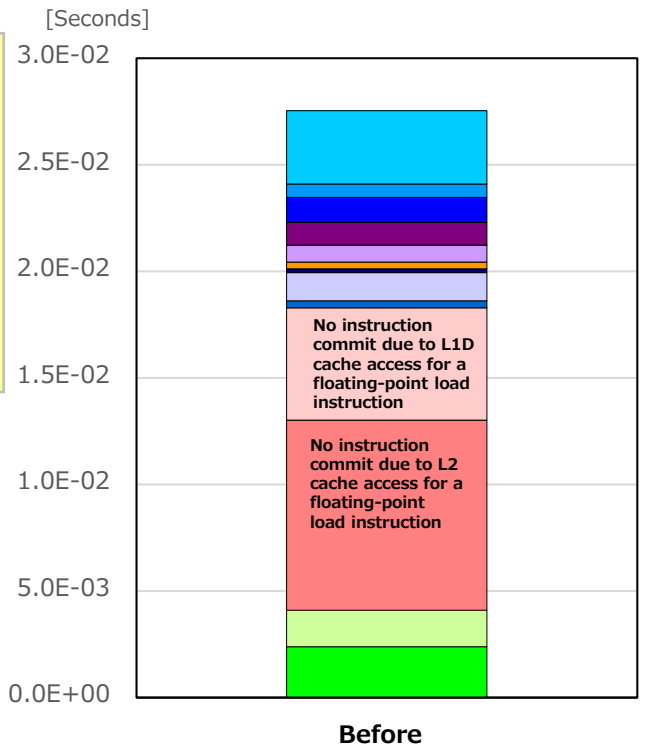
Each array is on a 16 KB boundary. L1D cache thrashing occurs. Consequently, the "No instruction commit due to access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
45          void sub(double s)
46          {
47            int i;
48
49          #pragma omp parallel for
          <<< Loop-information Start >
        <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 1.90, ITR: 56,
                                MVE: 4, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      f, d, b, h, g, e, c, a
          <<< Loop-information  End >>>
50  p     v    for(i=0;i<n; i++)
51  p     v    {
52  p     v      a[i] =  s / b[i];
53  p     v      c[i] =  s / d[i];
54  p     v      e[i] =  s / f[i];
55  p     v      g[i] =  s / h[i];
56  p     v    }
57
58            return;
59          }
```

Array declaration
double a[65536], b[65536], c[65536], d[65536], e[65536], f[65536], g[65536], h[65536];

Array a,b,c,d,e,f,g,h size: 65536 x 8B= 32x16 KB(16 KB boundary)



High L1D miss and L1 miss dm rates despite sequential array access
-> L1D cache thrashing occurs

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	1.17E+08	3.93E+07	0.34	58.44%	41.56%	0.00%	1.94E+04	0.00	16.93%	87.92%	0.00%

Effect of Loop Fission (Improved Thrashing) (Source Tuning)

FUJITSU

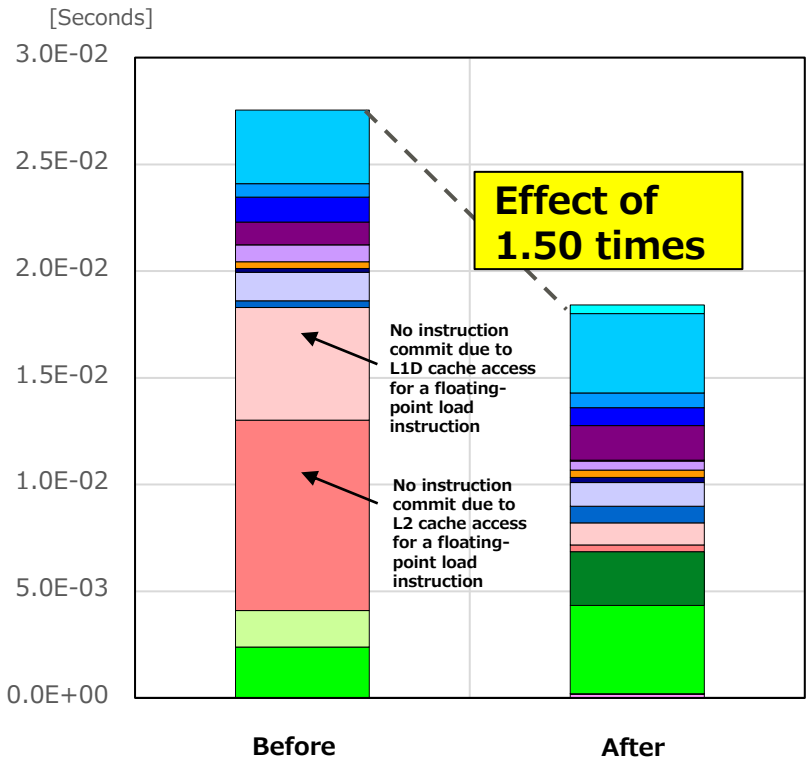
Loop fission reduces the number of streams from 8 to 4, preventing L1D cache thrashing. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement

```
45          void sub(double s)
46          {
47            int i;
48
49          #pragma omp parallel for
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
        <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.09, ITR: 64,
                        MVE: 2, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      d, b, c, a
          <<< Loop-information  End >>>
50  p   2v   for(i=0;i<n; i++)
51  p   2v   {
52  p   2v     a[i] =  s / b[i];
53  p   2v     c[i] =  s / d[i];
54  p   2v   }
55
56          #pragma omp parallel for
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.09, ITR: 64,
                        MVE: 2, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      h, f, g, e
          <<< Loop-information  End >>>
57  p   2v   for(i=0;i<n; i++)
58  p   2v   {
59  p   2v     e[i] =  s / f[i];
60  p   2v     g[i] =  s / h[i];
61  p   2v   }
62            return;
63          }
```

Array declaration
double a[65536], b[65536], c[65536], d[65536], e[65536], f[65536], g[65536], h[65536];

Loop fission



L1D miss and L1D miss dm rates reduced

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
Before	0.00	1.17E+08	3.93E+07	0.34	58.44%	41.56%	0.00%
After	0.00	1.51E+08	1.86E+07	0.12	12.30%	87.72%	-0.02%

Effect of Loop Fission (Optimization Control Line Tuning)

FUJITSU

You can obtain an effect equivalent to that of source tuning by specifying the following optimization control line option.

Optimization Specifier (Fortran)	Meaning	Optimization Control Line Specifiable?			
		By Program	By DO Loop	By Statement	By Array Assignment Statement
FISSION_POINT[(n1)] (n1 is a decimal number from 1 to 6.)	Specifies fission of a loop at the specified point in the loop. The *n1*-fold nested multiloop is looped. (*n1* is counted from the innermost loop.)	No	No	Yes	No

Optimization Specifier (C/C++: Trad Mode)	Meaning	Optimization Control Line Specifiable?			
		global	procedure	loop	statement
fission_point[(n1)] (n1 is a decimal number from 1 to 6.)	Specifies fission of a loop at the specified point in the loop. The *n1*-fold nested multiloop is looped. (*n1* is counted from the innermost loop.)	No	No	No	Yes

Source After Improvement (Optimization Control Line Tuning)

```
46                parameter(n=65536)
47                real*8  a(n),b(n),c(n),d(n),e(n),f(n),g(n),h(n)
48                common /com/a,b,c,d,e,f,g,h
49
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 411
                <<<  [OPTIMIZATION]
                <<<    FISSION(num: 2)
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.45, ITR: 80, MVE: 2, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, d, c, a, f, h, g, e
                <<< Loop-information  End >>>
50    1  pp   2v       do i=1,n
51    1   p   2v         a(i) =  s / b(i)
52    1   p   2v         c(i) =  s / d(i)
53    1             !ocl fission_point(1)
54    1   p   2v         e(i) =  s / f(i)
55    1   p   2v         g(i) =  s / h(i)
56    1   p   2v       enddo
57                  end subroutine sub
58
 Diagnostic messages: program name(sub)
   jwd8212o-i  "a.f90", line 50: Loop was split into two.
```

FUJITSU

Padding Using the Large Page Environment Variable

- XOS_MMM_L_FORCE_MMAP_THRESHOLD
- Padding Using the Large Page Environment Variable (Before Improvement)
- Padding Using the Large Page Environment Variable (After Improvement)

XOS_MMM_L_FORCE_MMAP_THRESHOLD

FUJITSU

Environment Variable Name	Specified Value (_ indicates default)	Description
XOS_MMM_L_FORCE_MMAP_THRESHOLD	0 | 1	Sets whether or not to give priority to mmap(2) when acquiring memory with a size equal to or greater than MALLOC_MMAP_THRESHOLD_ (default: 128 MiB). "0" means priority is not given to mmap(2). First, the heap area is searched for space. If there is space, the free memory of the heap area is returned. mmap(2) is used to acquire memory only when space is not found in the heap area. "1" means priority is given to mmap(2). mmap(2) is used to acquire memory without searching the heap area for space (even when there is space).

Fortran Padding Using the Large Page Environment Variable (Before Improvement)

FUJITSU

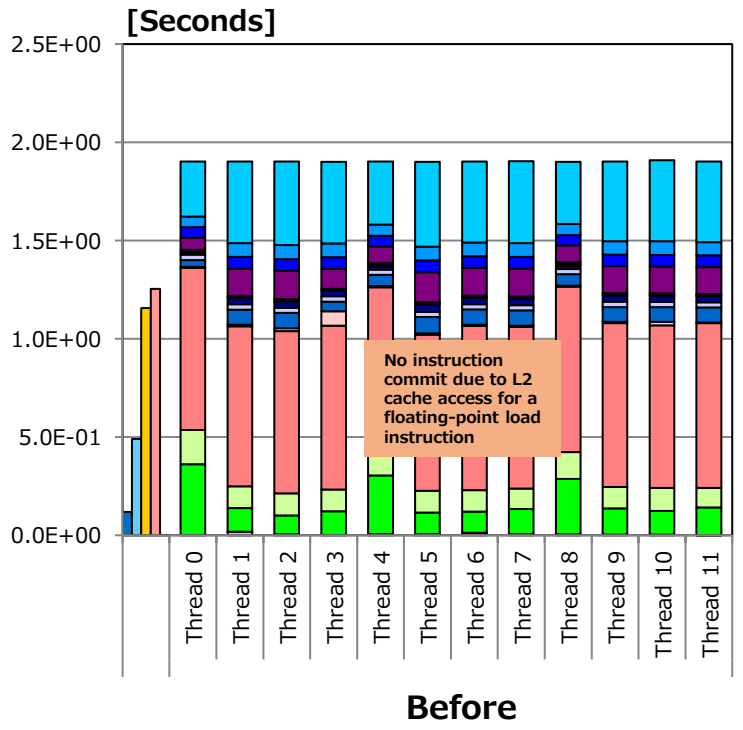
Each stream of the dynamic array is on a 16 KB boundary. L1D cache thrashing occurs. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
3                parameter(n=256,m=256)
4                real*8,allocatable ::  a(:,:),b(:,:),c(:,:),d(:,:),
                                        e(:,:),f(:,:),g(:,:),h(:,:)
6                allocate(a(n,m))
7                allocate(b(n,m))
8                allocate(c(n,m))
9                allocate(d(n,m))
10                allocate(e(n,m))
11                allocate(f(n,m))
12                allocate(g(n,m))
13                allocate(h(n,m))
:                  ......
36    1   p         do j = 1 , m
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELI                          MVE: 2,
POL: S)
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      c, b, d, e, f, g, h, a
                 <<< Loop-information  End >>>
37    2   p   v       do i = 1 , n
38    2   p   v           a(i, j) = b(i, j) + c(i, j) + d(i, j) + e(i, j)
                                  + f(i ,j) + g(i ,j) + h(i ,j)
39    2   p   v       enddo
40    1   p         enddo
```

Array size
256 x 256 x 8 B =
32 x 16 KB (16 KB boundary)

Streams of same size



Before

High L1D miss rates
-> L1D cache thrashing occurs

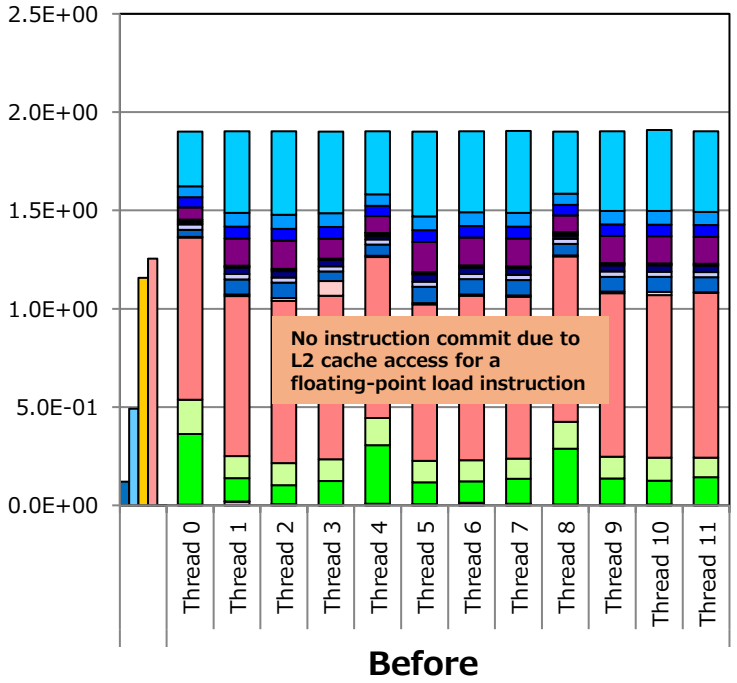
Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	1.34.E+10	3.44.E+09	0.26	51.52%	48.28%	0.20%	1.61.E+03	0.00	73.10%	53.55%	0.00%

Fortran Padding Using the Large Page Environment Variable (After Improvement)

FUJITSU

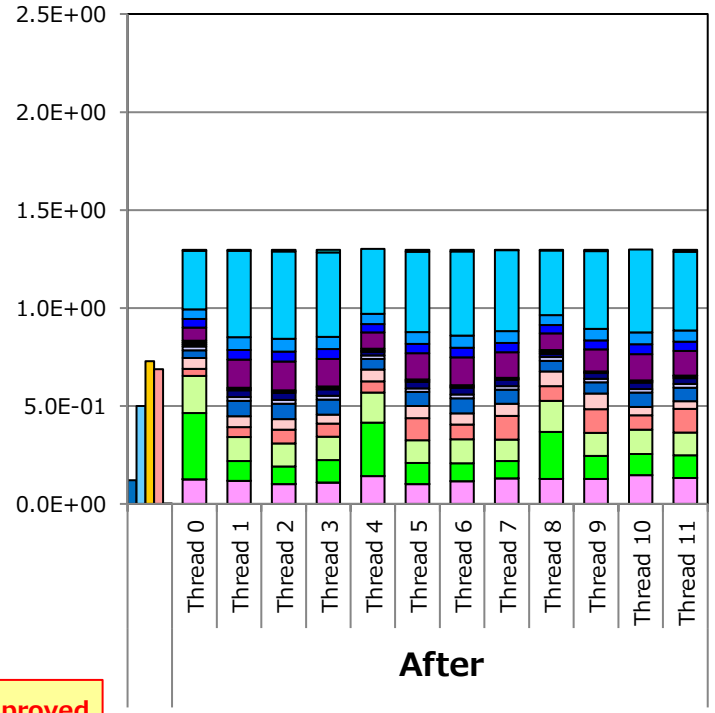
Specify the MALLOC_MMAP_THRESHOLD_=204800 environment variable to change the address alignment of each array to prevent L1D cache thrashing. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

[Seconds]



Before

[Seconds]



After

L1D miss and L1D miss dm rates improved

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	1.34.E+10	3.44.E+09	0.26	51.52%	48.28%	0.20%	1.61.E+03	0.00	73.10%	53.55%	0.00%
After	0.00	1.35.E+10	1.81.E+09	0.13	9.47%	90.51%	0.02%	2.43.E+03	0.00	68.31%	58.01%	0.00%

C/C++ Padding Using the Large Page Environment Variable (Before Improvement)

FUJITSU

Each stream of the dynamic array is on a 16 KB boundary. L1D cache thrashing occurs. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

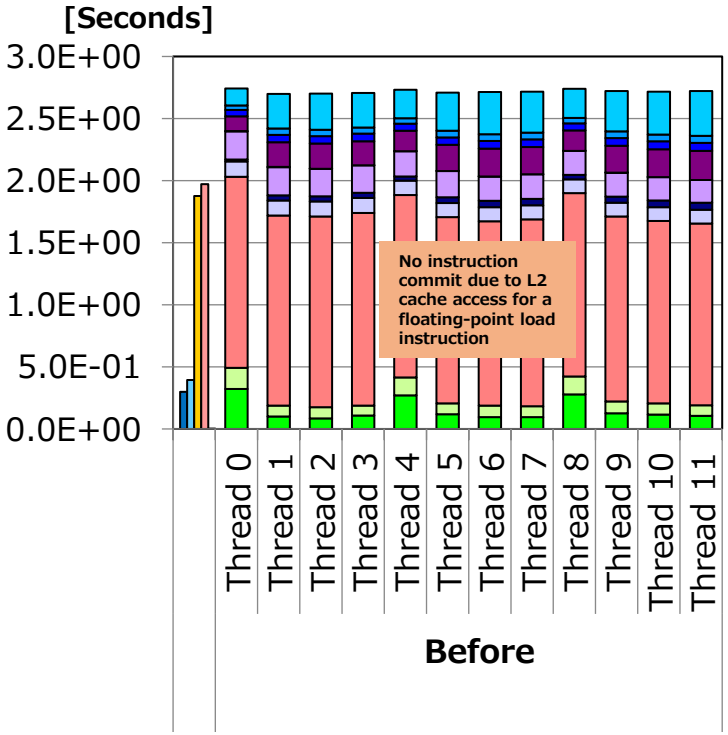
Source Before Improvement

```
62          void sub(int n, int m, double (* restrict a)[n], double (* restrict b)[n],
              double (* restrict c)[n], double (* restrict d)[n], double (* restrict e)[n],
              double (* restrict f)[n], double (* restrict g)[n], double (* restrict h)[n])
63          {
64            int i,j;
65          #pragma omp parallel for private(i)
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by
          <<<      (unknown)
          <<< Loop-information  End >>>
66  p         for(j = 0; j<m; j++)
67  p         {
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler
          <<<      (unknown)
          <<< Loop-information  End >>>
68  p   v     for(i = 0; i<n; i++)
69  p   v     {
70  p   v       a[j][i] = b[j][i] + c[j][i] + d[j][i] + e[j][i] + f[j][i] + g[j][i] + h[j][i];
71  p   v     }
72  p         }
73            return;
74          }
```

Array declaration
double a[256][256], b[256][256], c[256][256], d[256][256], e[256][256], f[256][256], g[256][256], h[256][256];

Array a size: 256×256×8B= 32×16 KB(16 KB boundary)

Streams of same size



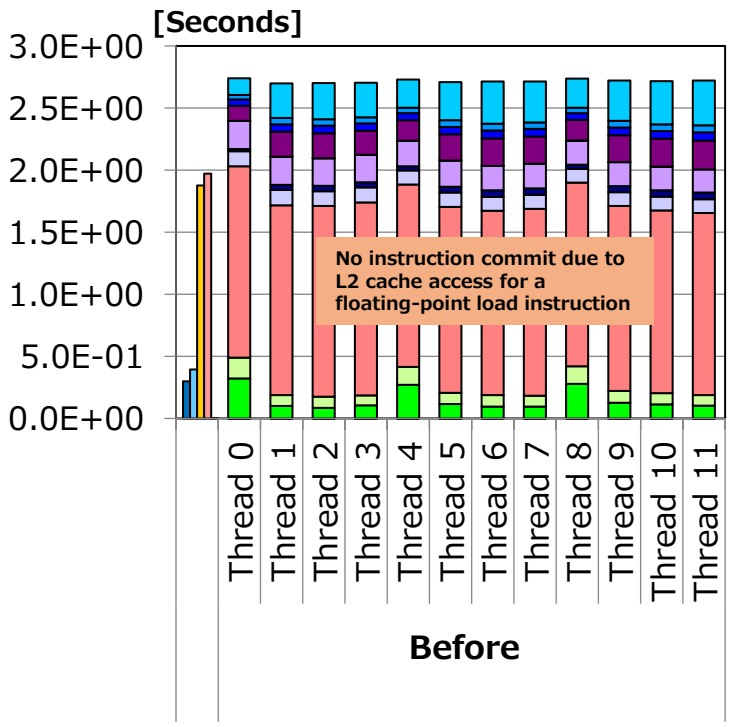
High L1D miss rates
-> L1D cache thrashing occurs

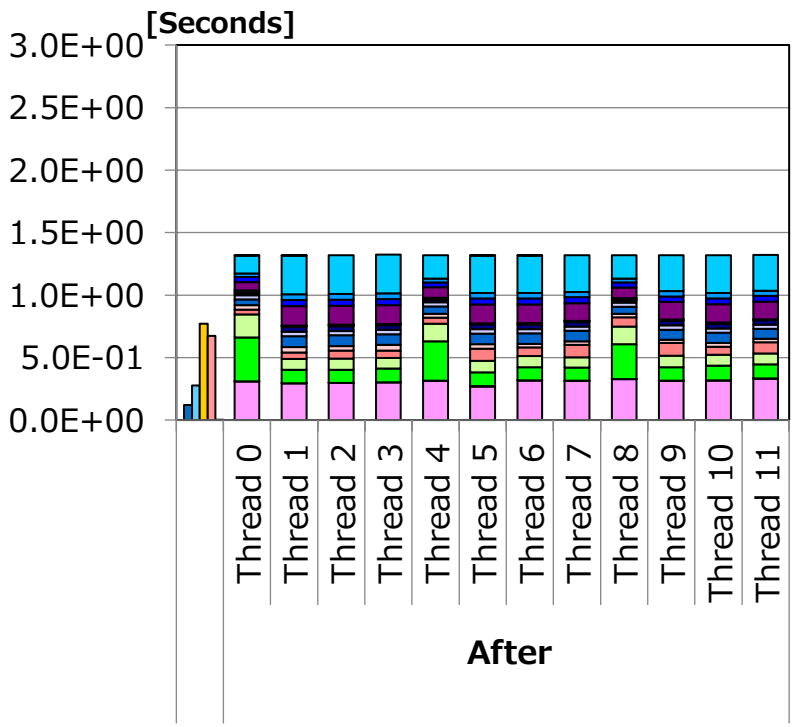
Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	1.39E+10	5.24E+09	0.38	69.38%	30.62%	0.01%	7.49E+04	0.00	68.31%	44.20%	0.00%

C/C++ Padding Using the Large Page Environment Variable (After Improvement)

FUJITSU

Specify the MALLOC_MMAP_THRESHOLD_=204800 environment variable to change the address alignment of each array to prevent L1D cache thrashing. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.





L1D miss and L1D miss dm rates improved

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	1.39E+10	5.24E+09	0.38	69.38%	30.62%	0.01%	7.49E+04	0.00	68.31%	44.20%	0.00%
After	0.00	1.25E+10	1.80E+09	0.14	9.31%	90.69%	0.00%	2.98E+04	0.00	49.45%	57.41%	0.00%

FUJITSU

Operation Wait (Facilitation of SIMDization)

- Loop Peeling
- Loops With an Unclear Defining Relationship
- Loops Containing Pointer Variables

FUJITSU

Loop Peeling

- What is Loop Peeling?
- Loop Peeling (Before Improvement)
- Loop Peeling (Source Tuning)

What is Loop Peeling?

FUJITSU

Loop peeling is a means to separate part of processing from a loop.

SIMDization or effective software pipelining may not be performed when a loop contains data dependency.

As shown below, you can address the problem of dependency in a loop by peeling the loop to separate only the dependency part from the loop.

Source Before Improvement

```
do j = 1, n
   a[j] = a[j] + a[1]
enddo
```

Source After Improvement

```
a[1] = a[1] + a[1]
do j = 2, n
   a[j] = a[j] + a[1]
enddo
```

The example on the left shows partial dependency, because a[1] defined when j=1 is used when $2 \leq j \leq n$.

The example on the right shows that the dependency in the loop can be removed by peeling the loop only in cases where j=1.

Fortran Loop Peeling (Before Improvement)

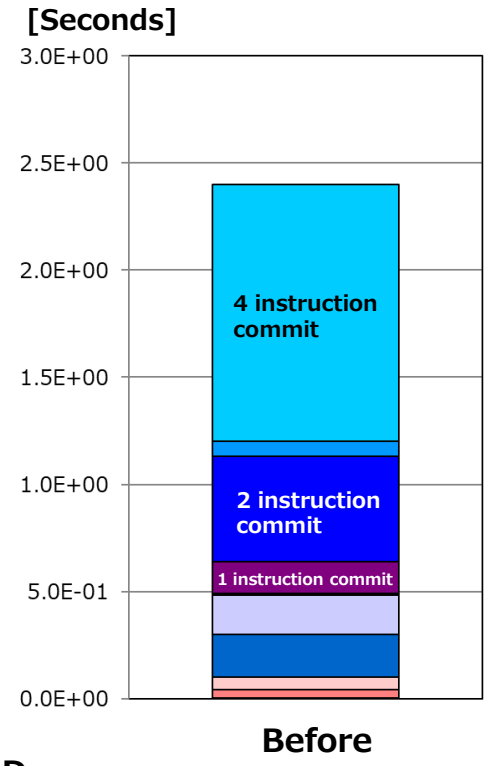
SIMDization is not performed effectively because Array a has a data dependency. The dependency is that what is defined at i=1 is referenced at i=2 or higher.

Source Before Improvement

```
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expe
                <<<      b, (unknown)
                <<< Loop-information  End >
 8    2   p             do j = 1, m
                <<< Loop-information Start >
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 1.38, ITR: 144,
                                           MVE: 5, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a, b, (unknown)
                <<< Loop-information  End >>>
 9    3   p  2m            do i = 1, n
10    3   p  2m             a(i,j) = c0 + a(1,j)*(c1 + b(i,j)*(c2 + b(i,j)*(c3 + b(i,j)*
11    3              &          (c4 + b(I,j)*(c5 + b(I,j)*(c6 + b(I,j)*(c7 + b(I,j)*
12    3              &          (c8 + b(i,j)*c9)))))))
13    3   p   v          end do
14    2   p            end do
```

SIMDization only partially done

Array a has a dependency between the load side and store side when i=1.



SIMD

	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
Before	1.49E+11	14.61%

Fortran Loop Peeling (Source Tuning)

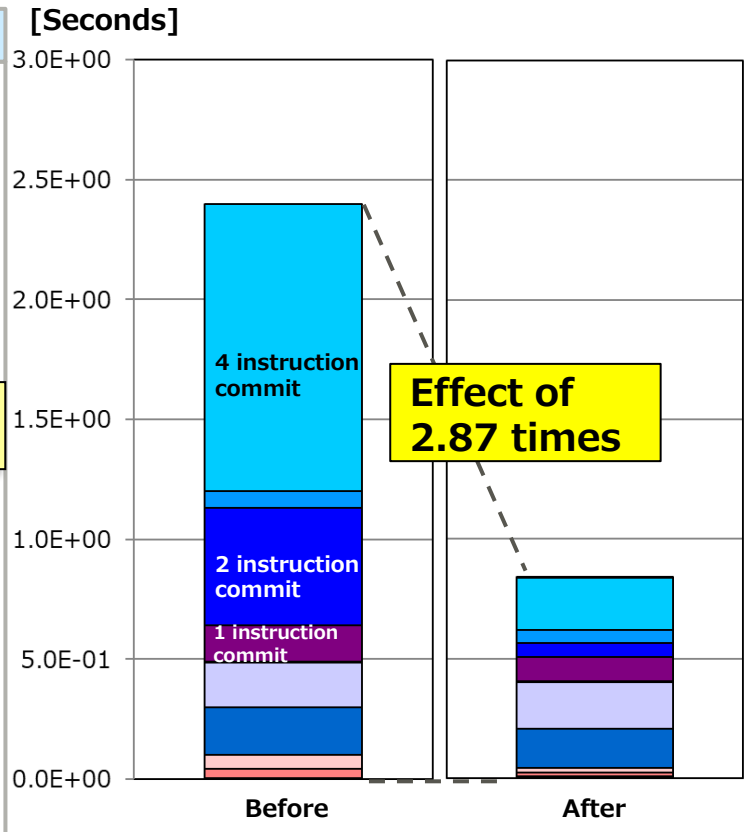
SIMDization is facilitated by loop peeling at one iteration. This results in a reduced total number of effective instructions, reduced instruction commits, and higher performance.

Source After Improvement (Source Tuning)

```
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
 8    2  p              do j = 1, m
 9    2  p                i = 1
10    2  p                a(i,j) = c0 + a(1,j)*(c1 + b(i,j)*(c2 + b(i,j)*(c3 + b(i,j)*
11    2                &          (c4 + b(i,j)*(c5 + b(i,j)*(c6 + b(i,j)*(c7 + b(i,j)*
12    2                &          (c8 + b(i,j)*c9))))))))
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 1.23, ITR: 128,
                                           MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
13    3  p  2v           do i = 2, n
14    3  p  2v             a(i,j) = c0 + a(1,j)*(c1 + b(i,j)*(c2 + b(i,j)*(c3 + b(i,j)*
15    3                &          (c4 + b(i,j)*(c5 + b(i,j)*(c6 + b(i,j)*(c7 + b(i,j)*
16    3                &          (c8 + b(i,j)*c9)))))))))
17    3  p  2v           end do
18    2  p              end do
```

Peeling of dependency part

SIMDization facilitated since dependency was removed



SIMD	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
Before	1.49E+11	14.61%
After	2.96E+10	81.26%

Loop Peeling (Before Improvement)

FUJITSU

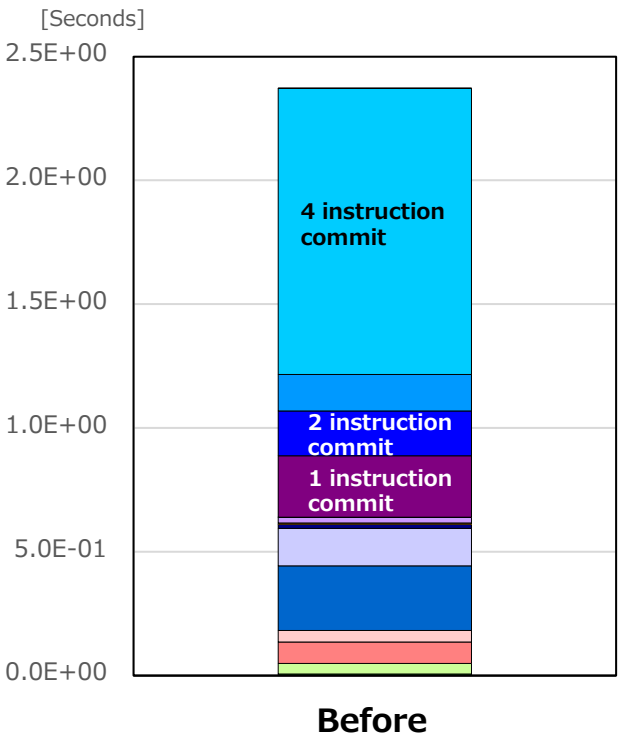
SIMDization is not performed effectively because Array a has a data dependency. The dependency is that what is defined at i=0 is referenced at i=1 or higher.

Source Before Improvement

```
40            #pragma omp parallel
41            for(k=0;k<1000000;k++)
42            #pragma omp for nowait
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
43  p           for(j=0;j<m;j++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 1.27, ITR: 144, MVE: 5, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
44  p    2m       for(i=0;i<n;i++){
45  p    2m         a[j][i] = c0 + a[j][0]*(c1 + b[j][i]*(c2 + b[j][i]*(c3 + b[j][i]*
46                      (c4 + b[j][i]*(c5 + b[j][i]*(c6 + b[j][i]*(c7 + b[j][i]*
47                      (c8 + b[j][i]*c9))))))));
48  p    v      }
49  p         }
50          }
51        }
```

SIMDization only partially done

Array a has a dependency between the load side and store side when i=0.



Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
Before	1.33E+11	16.32%

Loop Peeling (Source Tuning)

FUJITSU

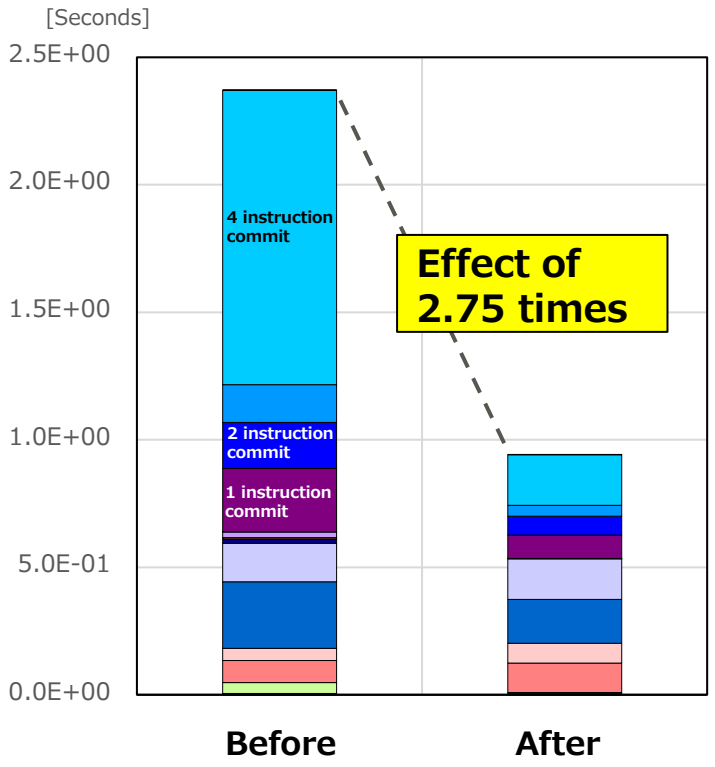
SIMDization is facilitated by loop peeling at one iteration. This results in a reduced total number of effective instructions, reduced instruction commits, and higher performance.

Source After Improvement (Source Tuning)

```
40          #pragma omp parallel
41          for(k=0;k<1000000;k++){
42          #pragma omp for nowait
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
43  p         for(j=0;j<m;j++){
44  p           i=0;
45  p           a[j][i] = c0 + a[j][0]*(c1 + b[j][i]*(c2 + b[j][i]*(c3 + b[j][i]*
46                    (c4 + b[j][i]*(c5 + b[j][
47                    (c8 + b[j][i]*c9)))))));
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 1.23, ITR: 128, MVE: 5, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
48  p   2v      for(i=1;i<n;i++){
49  p   2v        a[j][i] = c0 + a[j][0]*(c1 + b[j][i]*(c2 + b[j][i]*(c3 + b[j][i]*
50                    (c4 + b[j][i]*(c5 + b[j][i]*(c6 + b[j][i]*(c7 + b[j][i]*
51                    (c8 + b[j][i]*c9)))))));
52  p   2v      }
53  p         }
54          }
55        }
```

Peeling of dependency part

SIMDization facilitated since dependency was removed



Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
Before	1.33E+11	16.32%
After	2.79E+10	86.10%

FUJITSU

Loops With an Unclear Defining Relationship

- Loops With an Unclear Defining Relationship (Before Improvement)
- Loops With an Unclear Defining Relationship (Optimization Control Line Tuning)
- Loops With an Unclear Defining Relationship (Optimization Control Line)

Loops With an Unclear Defining Relationship (Optimization Control Line)

FUJITSU

Specify the following optimization control line.

Optimization Specifier (Fortran)	Meaning	Optimization Control Line Specifiable?			
		By Program	By DO Loop	By Statement	By Array Assignment Statement
NORECURRENCE [(*array1*[,*array2*]...)]	Notifies the main processing system that the elements of arrays targeted by operations in a DO loop are not defined and cited across iterations. (Loops can be sliced for the specified arrays.) *array1*, *array2,* and so on are array names.	Yes	Yes	No	Yes

Optimization Specifier (C/C++)	Meaning	Optimization Control Line Specifiable?			
		global	procedure	loop	statement
norecurrence [(*array1*[,*array2*]...)]	Notifies the main processing system that the elements of arrays targeted by operations in a loop are not defined and cited across iterations. (Loops can be sliced for the specified arrays.) *array1*, *array2,* and so on are array names.	Yes	Yes	No	Yes

Fortran Loops With an Unclear Defining Relationship (Before Improvement)

FUJITSU

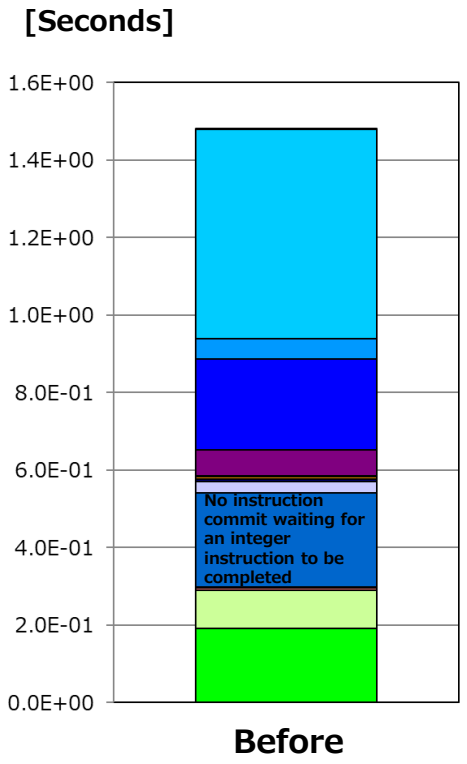
SIMDization and software pipelining are not performed effectively because data dependency is unclear in Array a. Consequently, the "No instruction commit waiting for an integer instruction to be completed" event occurs many times.

Source Before Improvement

```
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<
                <<<
                <<<
                <<<
                <<< Loop-information  End >>>
53    1  pp             do j=1,n1
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SOFTWARE PIPELINING(IPC: 0.56, ITR: 3,
                                           MVE: 2, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      l, b, x
                <<< Loop-information  End >>>
54    2  p    s             do i=1,n2
55    2  p    m                 a(l(i),j)=a(x(i),j)/b(i,j)
56    2  p    v             end do
57    1  p                end do
```

Software pipelining is not performed effectively because data dependency across iterations in Array a is unclear.

Dependency between the load side and store side of Array a is unclear.



SIMD	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
Before	7.10E+10	0.00%

Fortran Loops With an Unclear Defining Relationship (Optimization Control Line Tuning)

FUJITSU

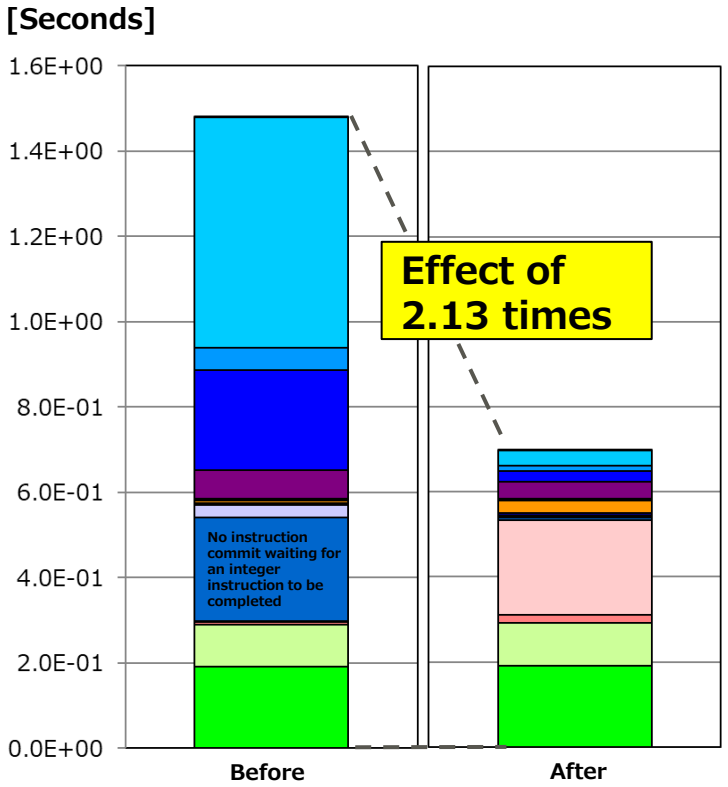
Use the NORECURRENCE specifier to explicitly specify no data dependency in order to facilitate SIMDization and software pipelining. The result is significant improvement of the "No instruction commit waiting for an integer instruction to be completed" event.

Source After Improvement (Optimization Control Line Tuning)

```
53                !ocl norecurrence(a)
                  <<< Loop-information
                  <<<  [PARALLELIZAT
                  <<<    Standard iterat
                  <<<  [OPTIMIZATION
                  <<<    PREFETCH(HAR
                  <<<      x, b, l
                  <<< Loop-information  End >>>
54    1  pp             do j=1,n1
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 16)
                  <<<    SOFTWARE PIPELINING(IPC: 2.72, ITR: 288,
                                                MVE: 3, POL: S)
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      x, b, l
                  <<< Loop-information  End >>>
55    2  p  2v           do i=1,n2
56    2  p  2v              a(l(i),j)=a(x(i),j)/b(i,j)
57    2  p  2v           end do
58    1  p             end do
```

Compiler notified about no data dependency between a(l(i) , j) and a(x(i) , j)

SIMDization and software pipelining done effectively



SIMD	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
Before	7.10E+10	0.00%
After	1.91E+10	10.08%

C/C++ Loops With an Unclear Defining Relationship (Before Improvement)

FUJITSU

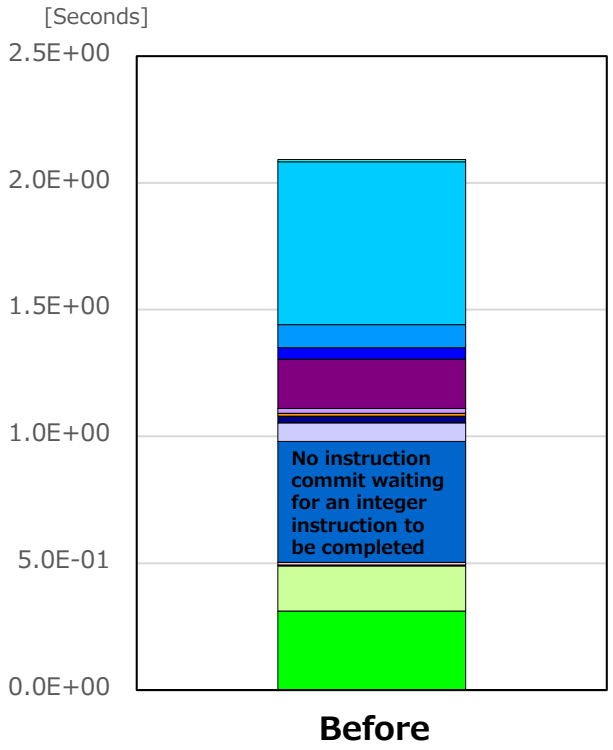
SIMDization and software pipelining are not performed effectively because data dependency is unclear in Array a. Consequently, the "No instruction commit waiting for an integer instruction to be completed" event occurs many times.

Source Before Improvement

```
48          #pragma omp parallel
49           {
50             #pragma omp for nowait
            <<< Loop-information Start >>>
            <<< [OPTIMIZATION]
            <<<   SOFTWARE PIPELINING(IPC: 0.39, ITR: 6,
                                      MVE: 2, POL: S)
            <<<   PREFETCH(HARD) Expected by compiler :
            <<<     (unknown)
            <<< Loop-information  End >>>
51   p        for(j=1; j<n1; j++)
52   p        {
            <<< Loop-information Start >>>
            <<< [OPTIMIZATION]
            <<<   SOFTWARE PIPELINING(IPC: 0.39, ITR: 6,
                                      MVE: 2, POL: S)
            <<<   PREFETCH(HARD) Expected by compiler :
            <<<     (unknown)
            <<< Loop-information  End >>>
53   p   2s      for(i=0; i<n2; i++)
54   p   2v      {
55   p   2m        a[j][l[i]]=a[j][x[i]]/b[j][i];
56   p   2v      }
57   p         }
58           }
60           return;
61         }
```

Software pipelining is not performed effectively because data dependency across iterations in Array a is unclear.

Dependency between the load side and store side of Array a is unclear.



Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
Before	8.49E+10	0.00%

Loops With an Unclear Defining Relationship (Optimization Control Line Tuning)

FUJITSU

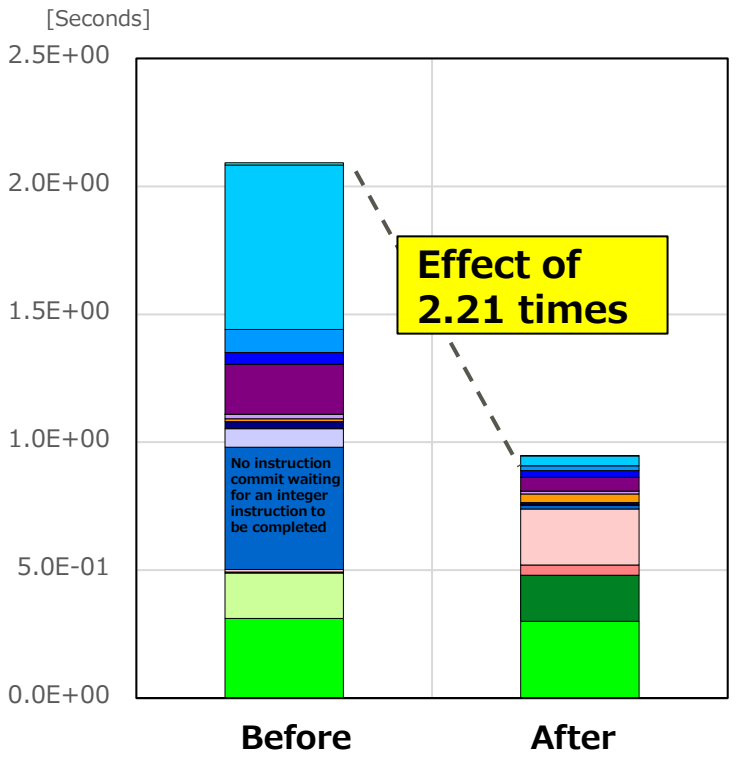
Use the norecurrence specifier to explicitly specify no data dependency in order to facilitate SIMDization and software pipelining. The result is significant improvement of the "No instruction commit waiting for an integer instruction to be completed" event.

Source After Improvement (Optimization Control Line Tuning)

```
48          #pragma omp parallel
49            {
50              #pragma omp for nowait
51              #pragma loop norecurrence
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) E
            <<<      (unknown)
            <<< Loop-information  End >>>
52  p           for(j=1; j<n1; j++)
53  p           {
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 16)
            <<<    SOFTWARE PIPELINING(IPC: 2.27,ITR: 256,
                                       MVE: 3, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
54  p   2v      for(i=0; i<n2; i++)
55  p   2v      {
56  p   2v        a[j][l[i]]=a[j][x[i]]/b[j][i];
57  p   2v      }
58  p         }
59            }
61            ret
62          }
```

Compiler notified about no data dependency between a[j, l[i]] and a[j, x[i]]

SIMDization and software pipelining done effectively



Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
Before	8.49E+10	0.00%
After	2.74E+10	5.49%

FUJITSU

Loops Containing Pointer Variables

- What is a Loop Containing a Pointer Variable?
- Loops Containing Pointer Variables (Before Improvement)
- Loops Containing Pointer Variables (Optimization Control Line Tuning)
- Loops Containing Pointer Variables (contiguous Attribute Specified)

What is a Loop Containing a Pointer Variable?

FUJITSU

Optimization may not be facilitated for a loop containing a pointer variable since which storage area part is occupied by the pointer variable is determined at execution.

Example of Source Code

```
real,dimension(100),target::x
real,dimension(:),pointer::a,b
a=>x(1:10)
b=>x(11:20)
do i=1,10000
    a(i)=2.0/b(i)+1.0
end do
```

Example of Source Code

```
      real,dimension(100),target::x
      real,dimension(:),pointer::a,b
!ocl noalias
      a=>x(1:10)
      b=>x(11:20)
      do i=1,10000
          a(i)=2.0/b(i)+1.0
      end do
```

By specifying the NOALIAS specifier, you can judge at the compile time that different pointer variables do not point to the same storage area. This facilitates optimization related to pointer variables.

However, if the combined state of pointer variables changes within the loop, optimization may not be facilitated even though the optimization specifier is specified.

Optimization Specifier (Fortran)	Meaning	Optimization Control Line Specifiable?			
		By Program	By DO Loop	By Statement	By Array Assignment Statement
NOALIAS	Specifies that a pointer variable not share a storage area with other variables.	Yes	Yes	No	Yes

Optimization Specifier (C/C++)	Meaning	Optimization Control Line Specifiable?			
		global	procedure	loop	statement
noalias	Specifies that a pointer variable not share a storage area with other variables.	Yes	Yes	Yes	No

Fortran What is a Loop Containing a Pointer Variable?

FUJITSU

You can obtain an effect equivalent to that of optimization control line tuning by specifying the following compiler option.

Compiler Option	Functional Description
-Knoalias [=spec]	Specifies optimization that assumes no pointer variable or pointer component combines a storage area with another variable. You can specify s in *spec*. If S is specified, the compiler performs optimization that assumes no entity with the Fortran pointer attribute is combined with another variable. In the following contexts, entities with the pointer attribute may combine with other variables: - Pointer assignment statement - Derived-type assignment statement with a pointer component - ALLOCATE statement where SOURCE=specifier shows a derived type with a pointer component - Dummy argument with a pointer attribute or component - Initial setting for variables with a pointer attribute or component

■**Use example (for source before improvement)**

```
$ frtpx -Kfast,parallel sample.f90 -Knoalias
```

Fortran Loops Containing Pointer Variables (Before Improvement)

FUJITSU

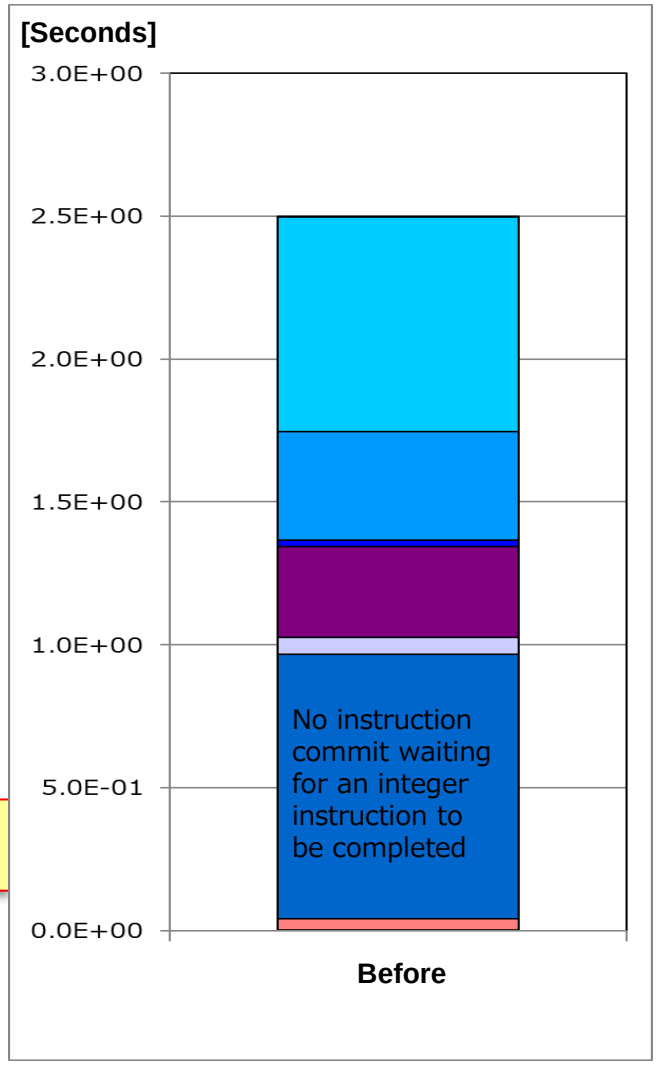
SIMDization is not facilitated because it is not clear that Pointer Variables a and b point to different storage areas. Consequently, the "No instruction commit waiting for an integer instruction to be completed" event occurs many times.

Source Before Improvement

```
 3                 real,dimension(100000),target::x
 4                 integer :: kmax
 5                 real,dimension(:),pointer::a,b
 :
 9                 a=>x(1:10000)
10                  b=>x(10001:20000)
11                  kmax = 1000000
12               !$omp parallel
13   1                do k=1,kmax
14   1             !$omp do
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SOFTWARE PIPELINING(IPC: 0.19, ITR: 8, MVE: 2, POL: L)
                <<< Loop-information  End >>>
15   2  p  2s           do i=1,10000
16   2  p  2s             a(i)=2.0/b(i)+1.0
17   2  p  2s           end do
18   1             !$omp enddo nowait
19   1                end do
20               !$omp end parallel
```

No SIMDization

Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
Before	1.10E+11	0.00%



Fortran Loops Containing Pointer Variables (Optimization Control Line Tuning)

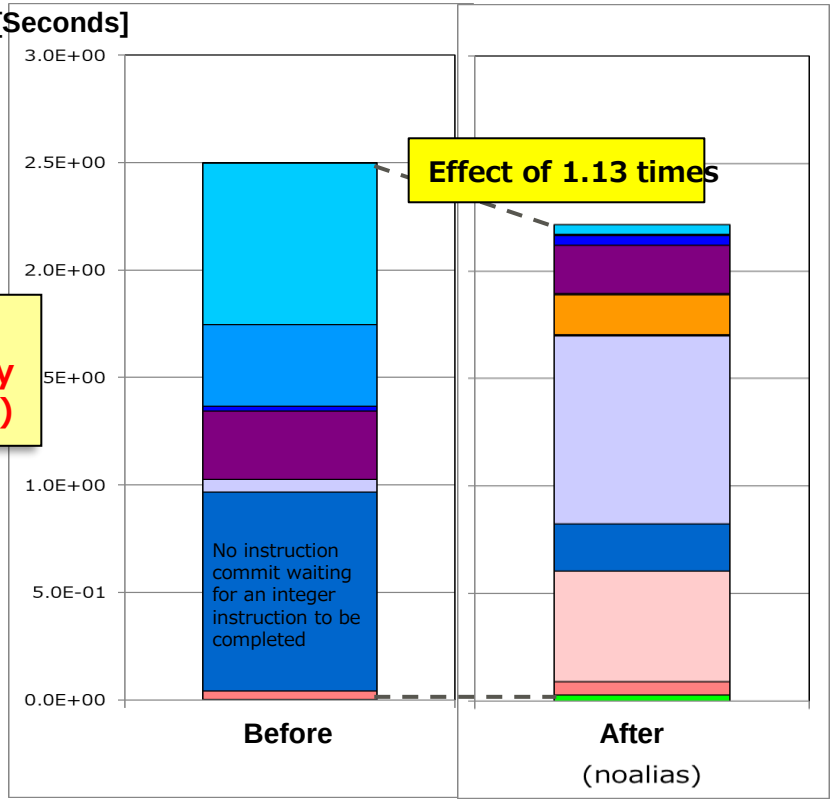
FUJITSU

Use the NOALIAS specifier to explicitly specify no data dependency in order to facilitate SIMDization and software pipelining. The result is significant improvement of the "No instruction commit waiting for an integer instruction to be completed" event and other events.

Source After Improvement (Optimization Control Line Tuning)

```
 3                    real,dimension(100000),target::x
 4                    integer :: kmax
 5                    real,dimension(:),pointer::a,b
 :
 9                    a=>x(1:10000)
10                     b=>x(10001:20000)
11                     kmax = 1000000
12                  !$omp parallel
13   1                  do k=1,kmax
14   1                !$omp do
15   1                !ocl noalias
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 16)
                  <<<    SOFTWARE PIPELINING(IPC: 0.19, ITR: 96,
                                             MVE: 2, POL: L)
                  <<< Loop-information  End >>>
16   2  p  2v             do i=1,10000
17   2  p  2v               a(i)=2.0/b(i)+1.0
18   2  p  2v             end do
19   1                !$omp enddo nowait
20   1                  end do
21                  !$omp end parallel
```

Compiler notified about no dependency between a(i) and b(i)



Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
Before	1.10E+11	0.00%
After	1.26E+10	40.83%

SIMDization reduced total number of effective instructions

Fortran Loops Containing Pointer Variables (contiguous Attribute Specified)

FUJITSU

Data continuity is explicitly specified when the contiguous attribute is specified, which changes non-contiguous instructions into contiguous instructions. The result is facilitated optimization and significant improvement of the "No instruction commit due to L1D cache access for a floating-point load instruction," "No instruction commit waiting for a floating-point instruction to be completed," and other events.

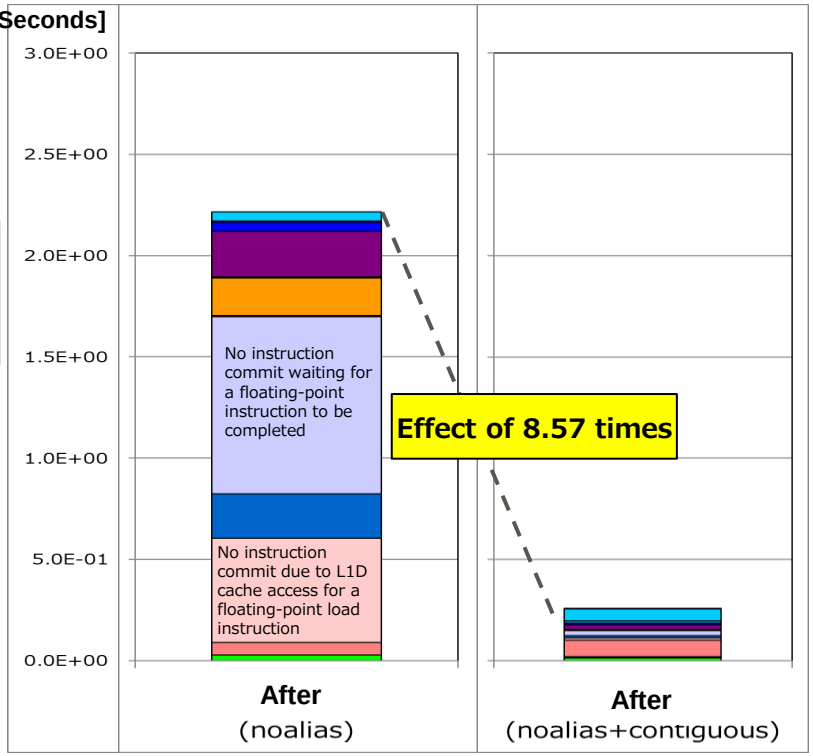
Source After Improvement (contiguous Attribute Specified)

```
 3                    real,dimension(100000),target::x
 4                    integer :: kmax
 5                    real,dimension(:),pointer,contiguous::a,b
 :
 9                    a=>x(1:10000)
10                     b=>x(10001:20000)
11                     kmax = 1000000
12                  !$omp parallel
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      (unknown)
                 <<< Loop-information  End >>>
13   1                  do k=1,kmax
14   1               !$omp do
15   1               !ocl noalias
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 16)
                 <<<    SOFTWARE PIPELINING(IPC: 2.62, ITR: 352,
                                            MVE: 3, POL: S)
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      (unknown)
                 <<< Loop-information  End >>>
16   2  p  2v             do i=1,10000
17   2  p  2v               a(i)=2.0/b(i)+1.0
18   2  p  2v             end do
19   1               !$omp enddo nowait
20   1                  end do
21                  !$omp end parallel
```

Compiler notified about pointer arrays a and b in separate contiguous areas

Compiler notified about no dependency between a(i) and b(i)

Non-contiguous instructions changed into contiguous instructions



Instruction	Load-store instruction					
	Load instruction			Store instruction		
	SIMD		Non-SIMD	SIMD		Non-SIMD
	Single vector contiguous load instruction	Non-contiguous gather load instruction	Non-SIMD load instruction	Single vector contiguous store instruction	Non-contiguous scatter store instruction	Non-SIMD store instruction
After (noalias)	1.10E+01	6.36E+08	1.08E+09	1.56E+02	6.36E+08	1.33E+08
After (noalias+contiguous)	6.36E+08	0.00E+00	5.00E+08	6.36E+08	0.00E+00	3.70E+07

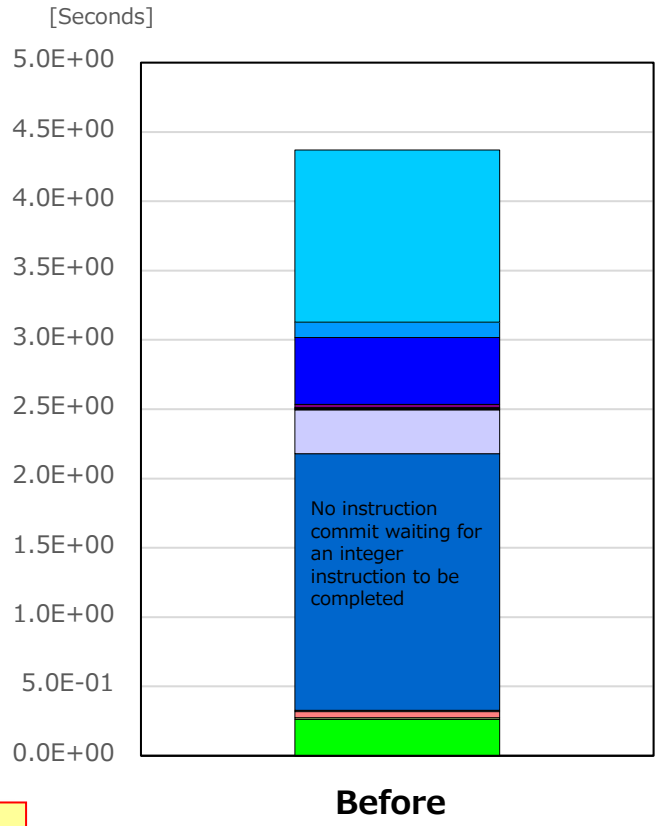
C/C++ Loops Containing Pointer Variables (Before Improvement)

FUJITSU

SIMDization is not facilitated because it is not clear that Pointer Variables a and b point to different storage areas. Consequently, the "No instruction commit waiting for an integer instruction to be completed" event occurs many times.

Source Before Improvement

```
17          #pragma omp parallel
18          {
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
19            for(k=0; k<kmax; k++)
20            {
21              #pragma omp for nowait
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SOFTWARE PIPELINING(IPC: 0.21, ITR: 8, MVE: 2, POL: L)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
22  p    2s       for(i=0; i<10000; i++)
23  p    2s       {
24  p    2s         a[i]=2.0/b[i]+1.0;
25  p    2s       }
26            }
27          }
```



No SIMDization

Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
Before	1.50E+11	0.00%

C/C++ Loops Containing Pointer Variables (Optimization Control Line Tuning)

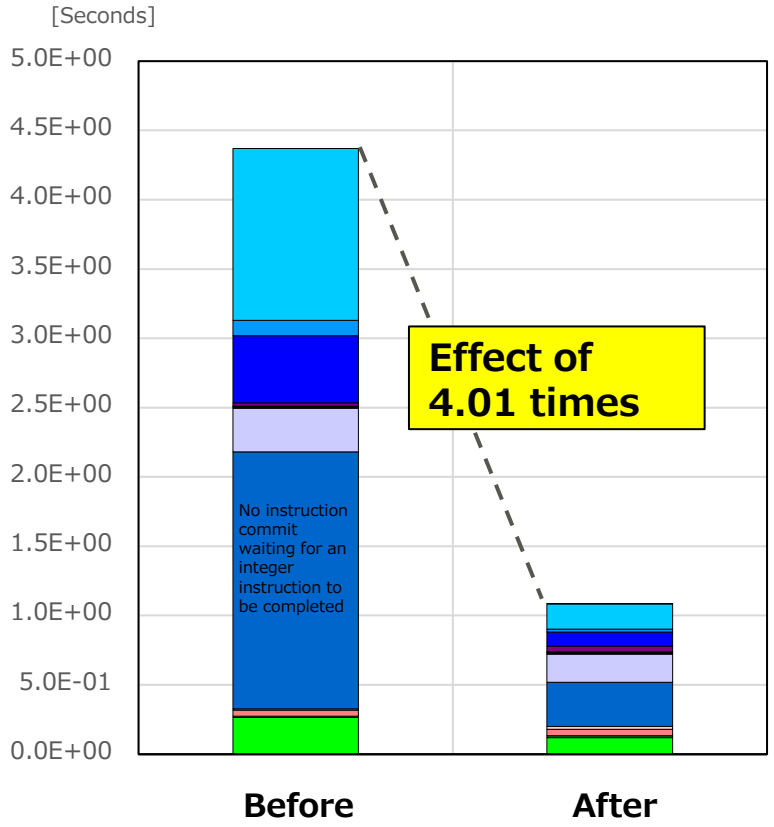
FUJITSU

Use the noalias specifier to explicitly specify no data dependency in order to facilitate SIMDization and software pipelining. The result is significant improvement of the "No instruction commit waiting for an integer instruction to be completed" event and other events.

Source After Improvement (Optimization Control Line Tuning)

```
17          #pragma omp parallel
18          {
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
19            for(k=0; k<kmax; k++)
20            {
21              #pragma omp for nowait
22              #pragma loop noalias
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 0.18, ITR: 64,
                                      MVE: 2, POL: L)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
23  p   2v      for(i=0; i<10000; i++)
24  p   2v      {
25  p   2v        a[i]=2.0/b[i]+1.0;
26  p   2v      }
27            }
28          }
```

Compiler notified about no dependency between a[i] and b[i]



Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
Before	1.50E+11	0.00%
After	2.27E+10	72.14%

SIMDization reduced total number of effective instructions

FUJITSU

Operation Wait (Hidden Latency)

- Loop Fission (Facilitation of software pipelining)
- Specifying the Appropriate Number of Unrolls and Suppressing Software Pipelining
- Specifying the Number of Striping (Interleaving) Expansions and Suppressing Software Pipelining
- Software Pipelining in an Outer Loop
- Rerolling
- Loop Unswitching

FUJITSU

Loop Fission (Facilitation of software pipelining)

- Loop Fission (Facilitation of software pipelining) (Before Improvement)
- Loop Fission (Facilitation of software pipelining) (Source Tuning)

Fortran Loop Fission (Facilitation of software pipelining) (Before Improvement)

FUJITSU

Optimization of scheduling such as SWPL is not possible because a long chain of operations requires many registers. Consequently, the "No instruction commit waiting for a floating-point instruction to be completed" event occurs many times.

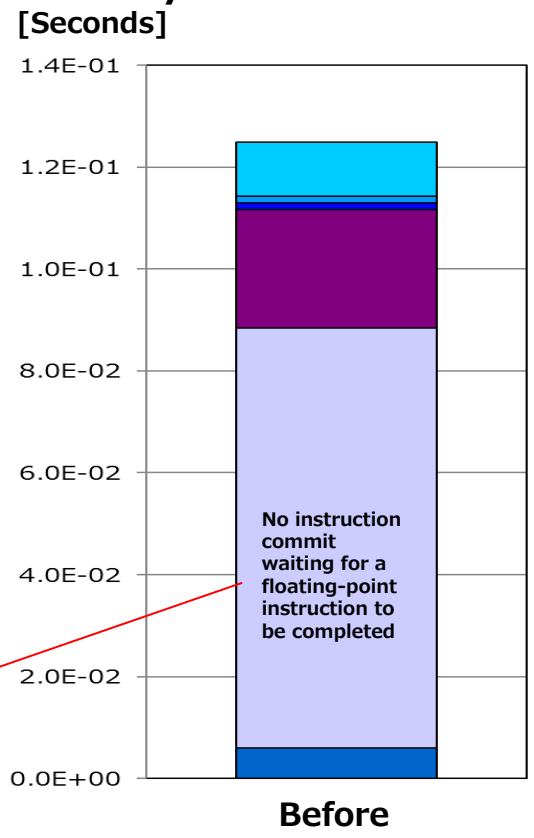
Source Before Improvement

```
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      x1, y
                <<< Loop-information  End >>>
18    2     2v          do i = 1, n
19    2     2v            y(i) = c0 / x1(i) / c1 / x1(i) &
20    2                       / c2 / x1(i) / c3 / x1(i) / c4 / x1(i)
21    2     2v          end do
```

* Compiler option
-Knoeval specified

Software pipelining cannot be applied because long operation chain uses many registers

Event appears large since scheduling is not optimized



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	7.69E+06	1.61E+03	0.00	71.62%	27.38%	1.00%	1.32E+02	0.00	48.92%	59.71%	0.00

Fortran Loop Fission (Facilitation of software pipelining) (Source Tuning)

FUJITSU

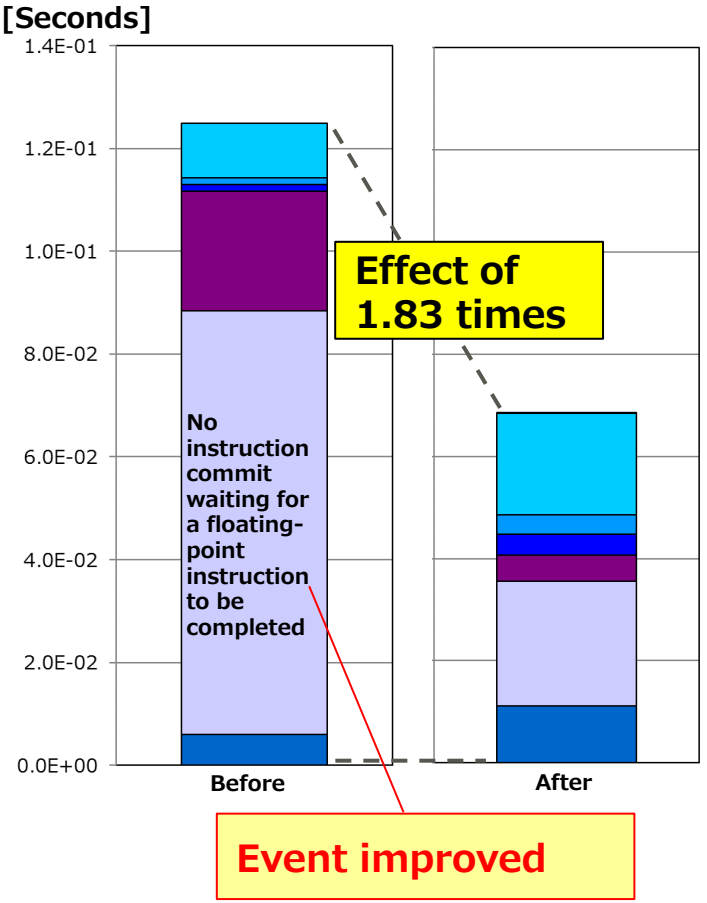
Loop fission shortens the chain of operations, reducing the registers used by a single loop. As a result, scheduling such as SWPL is optimized and the event is improved.

Source After Improvement

```
20    1          !ocl loop_nofusion
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELINING(IPC: 1.02, …
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      y, x1
                 <<<    SPILLS :
                 <<<      GENERAL   : SPILL 0  FILL 0
                 <<<      SIMD&FP   : SPILL 0  FILL 0
                 <<<      SCALABLE  : SPILL 0  FILL 27
                 <<<      PREDICATE : SPILL 0  FILL 0
                 <<< Loop-information  End >>>
21    2    2v          do i = 1, n
22    2    2v            y(i) = c2 / x1(i) / c3 / x1(i) / c4 / x1(i)
23    2    2v          end do
                   <<< Loop-information Start >>>
                   <<<  [OPTIMIZATION]
                   <<<    SIMD(VL: 8)
                   <<<    SOFTWARE PIPELINING(IPC: 1.08, …
                   <<<    PREFETCH(HARD) Expected by compiler :
                   <<<      x1, y
                   <<< Loop-information  End >>>
24    2    2v          do i = 1, n
25    2    2v            y(i) = c0 / x1(i) / c1 / x1(i) /  y(i)
26    2    2v          end do
```

Loop fusion suppressed

Loop fission



* Compile option -Knoeval specified

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
Before	0.00	7.69E+06	1.61E+03	0.00	71.62%	27.38%	1.00%
After	0.00	3.65E+07	1.73E+03	0.00	70.79%	28.98%	0.23%

Loop Fission (Facilitation of software pipelining) (Before Improvement)

FUJITSU

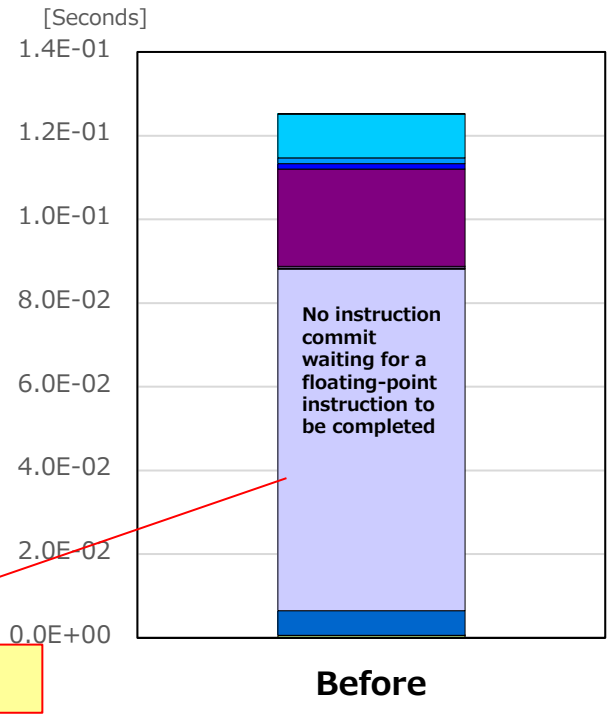
Optimization of scheduling such as SWPL is not possible because a long chain of operations requires many registers. Consequently, the "No instruction commit waiting for a floating-point instruction to be completed" event occurs many times.

Source Before Improvement

```
18          for (iter = 0; iter < 10000; iter++){
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
19        2v     for (i = 0; i < n; i++){
20        2v       y[i] = c0 / x1[i] / c1 / x1[i] / c2 / x1[i] / c3 / x1[i] / c4 / x1[i];
21        2v     }
22           }
```

※compile option: -Knoeval

Software pipelining cannot be applied because long operation chain uses many registers



Event appears large since scheduling is not optimized

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	1.11E+09	1.72E+06	0.00	99.92%	0.08%	-0.01%	4.01E+03	0.00	86.58%	26.24%	0.00%

Loop Fission (Facilitation of software pipelining) (Source Tuning)

FUJITSU

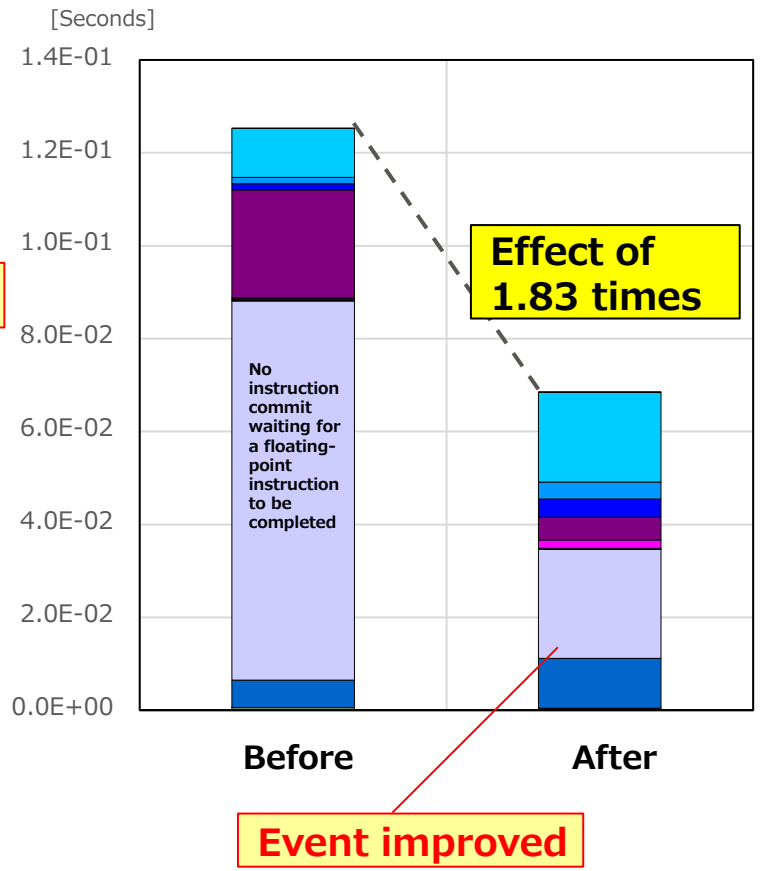
Loop fission shortens the chain of operations, reducing the registers used by a single loop. As a result, scheduling such as SWPL is optimized and the event is improved.

Source After Improvement

```
18          for (iter = 0; iter < 10000; iter++){
19          #pragma loop loop_nofusion
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 1.02, ITR: 112, MVE: 3, POL: L)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<<    SPILLS :
           <<<      GENERAL   : SPILL 0  FILL 0
           <<<      SIMD&FP   : SPILL 0  FILL 0
           <<<      SCALABLE  : SPILL 0  FILL 27
           <<<      PREDICATE : SPILL 0  FILL 0
           <<< Loop-information  End >>>
20        2v      for (i = 0; i < n; i++){
21        2v        y[i] = c2 / x1[i] / c3 / x1[i] / c4 / x1[i];
22        2v      }
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 1.08, ITR: 64, MVE: 2, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
23        2v      for (i = 0; i < n; i++){
24        2v        y[i] = c0 / x1[i] / c1 / x1[i] / y[i];
25        2v      }
26            }
```

Loop fusion suppressed

Loop fission



※compile option: -Knoeval

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
Before	0.00	1.11E+09	1.72E+06	0.00	99.92%	0.08%	-0.01%
After	0.00	6.46E+08	9.82E+05	0.00	99.88%	0.11%	0.00%

Loop Fission (Facilitation of software pipelining) (Before Improvement)

FUJITSU

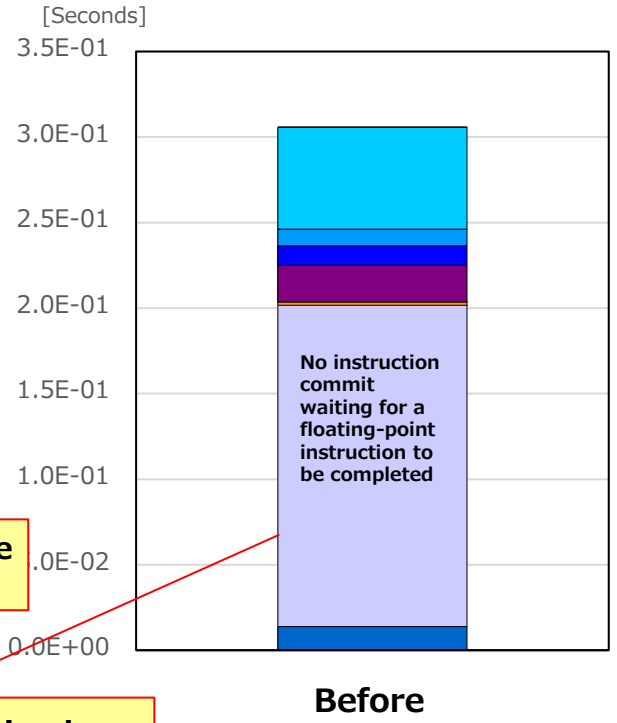
Optimization of scheduling such as SWPL is not possible because a long chain of operations requires many registers. Consequently, the "No instruction commit waiting for a floating-point instruction to be completed" event occurs many times.

Source Before Improvement

```
19          for (iter = 0; iter < 10000; iter++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8 Interleave: 1)
          <<<    SPILLS :
          <<<      GENERAL   : SPILL 0  FILL 0
          <<<      SIMD&FP   : SPILL 0  FILL 8
          <<<      SCALABLE  : SPILL 4  FILL 6
          <<<      PREDICATE : SPILL 0  FILL 0
          <<< Loop-information End >>>
20        v     for (i = 0; i < n; i++){
21                y[i] =    sin(x1[i]*c0)
22                        +cos(x1[i]*c1)
23                        +atan(x1[i]*c2)
24                        +log(x1[i]*c3);
25              }
26  }
```

Software pipelining cannot be applied because long operation chain uses many registers

※Based on the compiler's characteristics, the example is made using mathematical functions.



Event appears large since scheduling is not optimized

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	1.76E+09	2.04E+06	0.00	99.85%	0.14%	0.01%	9.95E+03	0.00	88.46%	18.51%	0.00%

Loop Fission (Facilitation of software pipelining) (Source Tuning)

FUJITSU

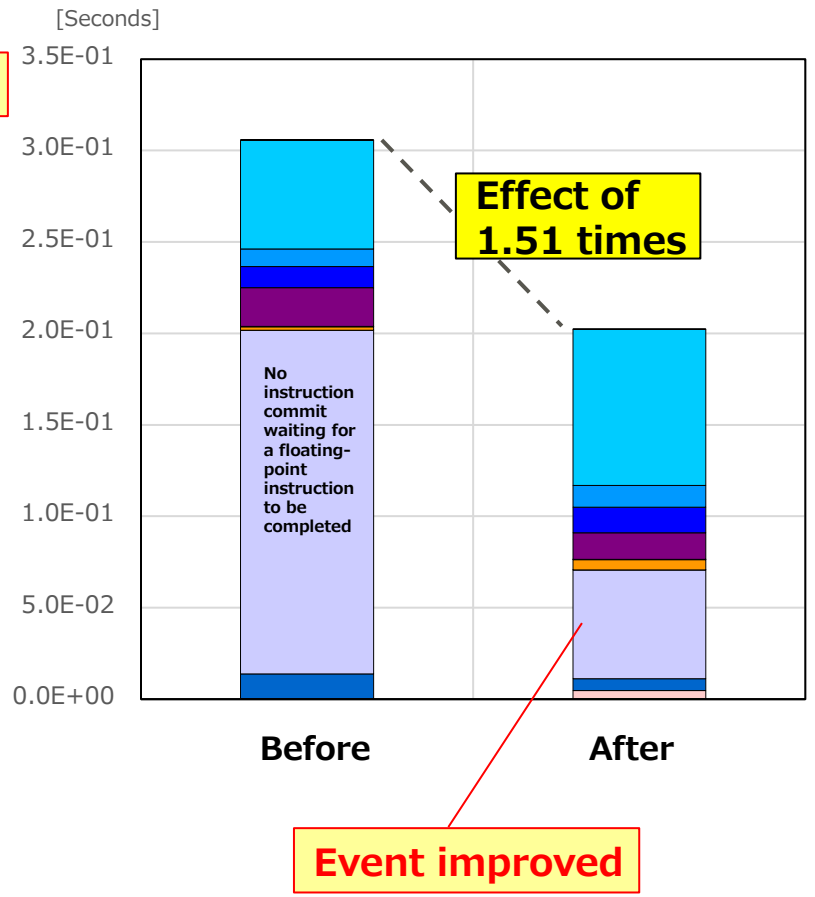
Loop fission shortens the chain of operations, reducing the registers used by a single loop. As a result, scheduling such as SWPL is optimized and the event is improved.

Source After Improvement

```
19          for (iter = 0; iter < 10000; iter++){
20            #pragma loop loop_nofusion
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8 Interleave: 1)
            <<<    SOFTWARE PIPELINING
            <<<    SPILLS :
            <<<      GENERAL   : SPILL 0  FILL 0
            <<<      SIMD&FP   : SPILL 0  FILL 0
            <<<      SCALABLE  : SPILL 8  FILL 17
            <<<      PREDICATE : SPILL 0  FILL 0
            <<< Loop-information End >>>
21        v     for (i = 0; i < n; i++){
22              y[i] =    sin(x1[i]*c0);
23            }
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            ...
            <<< Loop-information End >>>
24        v     for (i = 0; i < n; i++){
25              y[i] +=    cos(x1[i]*c1);
26            }
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            ...
            <<< Loop-information End >>>
27        v     for (i = 0; i < n; i++){
28              y[i] +=    atan(x1[i]*c2);
29            }
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            ...
            <<< Loop-information End >>>
30        v     for (i = 0; i < n; i++){
31              y[i] +=    log(x1[i]*c3);
32            }
33          }
```

Loop fusion suppressed

Loop fission



※ Based on the compiler's characteristics, the example is made using mathematical functions.

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
Before	0.00	1.76E+09	2.04E+06	0.00	99.85%	0.14%	0.01%
After	0.00	2.67E+09	2.39E+06	0.00	99.91%	0.10%	-0.01%

FUJITSU

Specifying the Appropriate Number of Unrolls and Suppressing Software Pipelining

- Loop Execution Operation After Software Pipelining
- Specifying the Appropriate Number of Unrolls and Suppressing Software Pipelining (Before Improvement)
- Specifying the Appropriate Number of Unrolls and Suppressing Software Pipelining (Optimization Control Line Tuning)
- Specifying the Appropriate Number of Unrolls and Suppressing Software Pipelining (Optimization Control Line)

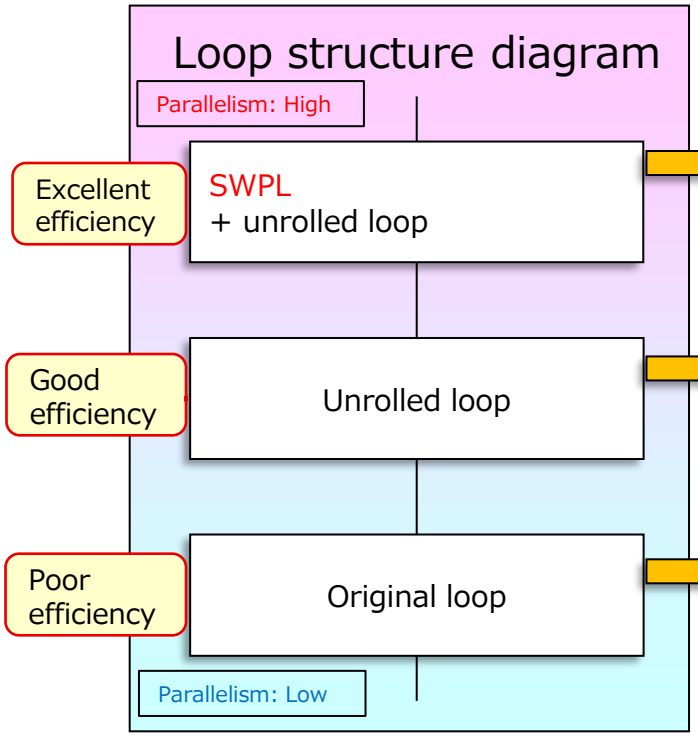
Loop Execution Operation After Software Pipelining

FUJITSU

- **Software pipelining determines processing routes as follows according to the number of loop iterations.**

```
Example      <<< Loop-information Start >>>
                                    :
             <<<  [OPTIMIZATION]
             <<<    SIMD(VL: 8)
             <<<    SOFTWARE PIPELINING(IPC: 2.50, ITR: 144,
                                        MVE: 4, POL: S)
             <<< Loop-information  End >>>
17    1  pp  2v       do i=1,N
18    1   p  2v        b(i)=a(i)+c
19    1   p  2v       enddo
```

Assumptions in example
Number of loop iterations n = 312
Number of unrolls = 2
SWPL
8SIMD



if (number of iterations corresponds to value presented by SWPL (144 or more iterations), then

software pipelining will be applied to the loop selected at execution when the number of loop iterations is 144 or more.
In the loop in the example, iterations from i=1 to i=288 are executed in this loop.

if (number of iterations excluding number of iterations immediately above is multiple of 16), then

2 unrolls x 8 (SIMD) = 16
In the loop in the example, iterations from i=289 to i=304 are executed in this loop.

if (number of iterations excluding number of iterations immediately above is multiple of 8), then

8 (SIMD)
In the loop in the example, the remaining iterations from i=305 to i=312 are executed in this loop.

In the above structural diagram of loops generated by the compiler in a multiplex manner, the higher in the hierarchy, the higher the parallelism at the instruction level.

As described above, the processing route is determined according to the number of loop iterations. Therefore, if the number of loop iterations is small, loops with high parallelism at the instruction level are not executed, so instruction scheduling has a small effect.

Fortran Specifying the Appropriate Number of Unrolls and Suppressing Software Pipelining (Before Improvement)

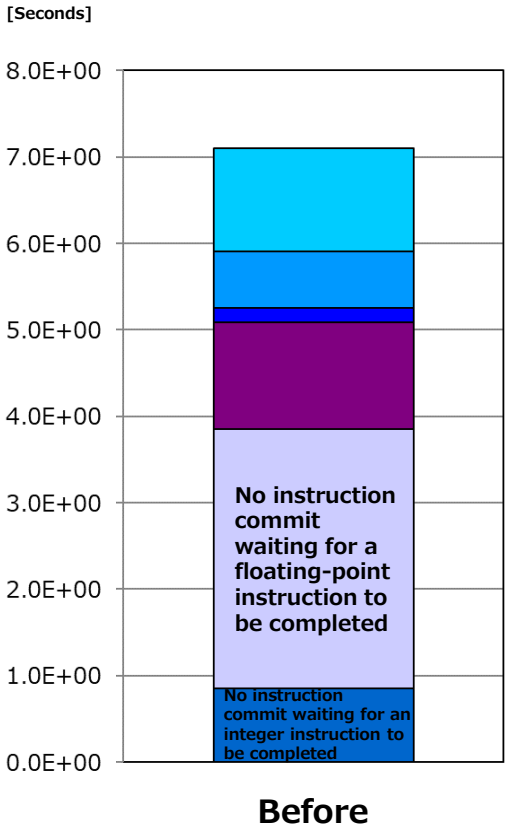
FUJITSU

Unrolling and software pipelining are not working effectively because the number of iterations is small. Consequently, the "No instruction commit waiting for a floating-point instruction to be completed" and "No instruction commit waiting for an integer instruction to be completed" events occur many times.

Source Before Improvement

```
        <<< Loop-information Start >>>
        <<<  [PARALLELIZATION]
        <<<    Standard iteration count:
        <<<  [OPTIMIZATION]
        <<<    SIMD(VL: 8)
        <<<    SOFTWARE PIPELINING(IPC: 1.62, ITR: 176,
                                   MVE: 6, POL: S)
        <<<    PREFETCH(HARD) Expected by compiler :
        <<<      a, b
        <<< Loop-information  End >>>
39    1  pp  2v   do i = 1 , n
40    1   p  2v     b(i) = c0 + a(i)*(c1 + a(i)*(c2 + a(i)*(c3 + a(i)*
41    1         &    (c4 + a(i)*(c5 + a(i)*(c6 + a(i)*(c7 + a(i)*
42    1         &    (c8 + a(i)*c9)))))))
43    1   p  2v          enddo
```

Number of loop iterations n = 40, while number of unrolls = 2



Problem: Small number of loop iterations

- Software-pipelined loop not executed

The software pipelining condition "ITR: 176" is not satisfied.

- Inappropriate number of unrolls

2 unrolls x 8 (SIMD) = 16

The original loop is executed for 8 iterations.

The execution of the original loop greatly affects performance.

Fortran

Specifying the Appropriate Number of Unrolls and Suppressing Software Pipelining (Optimization Control Line Tuning)

FUJITSU

Suppress software pipelining and specify a number of unrolls appropriate to the number of iterations to appropriately schedule instructions. The result is reduction of the "No instruction commit waiting for a floating-point instruction to be completed" and "No instruction commit waiting for an integer instruction to be completed" events.

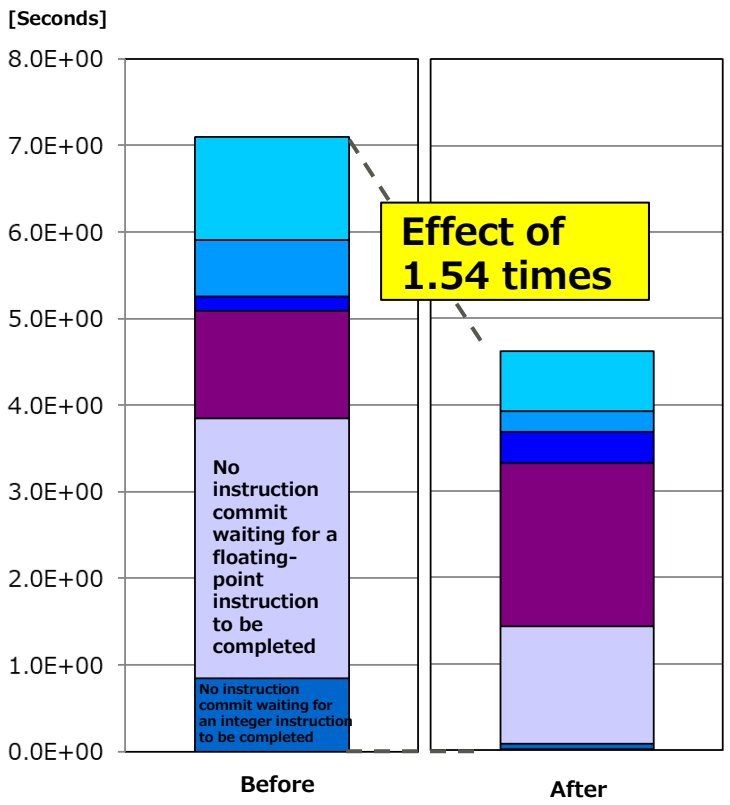
Source After Improvement (Optimization Control Line Tuning)

```
39              !ocl unroll(5)
40              !ocl noswp
          <<< Loop-information Start >>>
          <<<  [PARALLELIZATION]
          <<<    Standard iteration count: 55
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      a, b
          <<< Loop-information  End >>>
41    1  pp   5v  do i = 1 , n
42    1   p   5v     b(i) = c0 + a(i)*(c1 + a(i)*(c2 + a(i)*(c3 + a(i)*
43    1         &        (c4 + a(i)*(c5 + a(i)*(c6 + a(i)*(c7 + a(i)*
44    1         &        (c8 + a(i)*c9))))))))
45    1   p   5v    enddo
```

5 specified as number of unrolls

Software pipelining suppressed

Specifying 5 as the number of unrolls results in 5 unrolls x 8 (SIMD) = 40.
The unrolled loop is executed for all 40 iterations.



C/C++ Specifying the Appropriate Number of Unrolls and Suppressing Software Pipelining (Before Improvement)

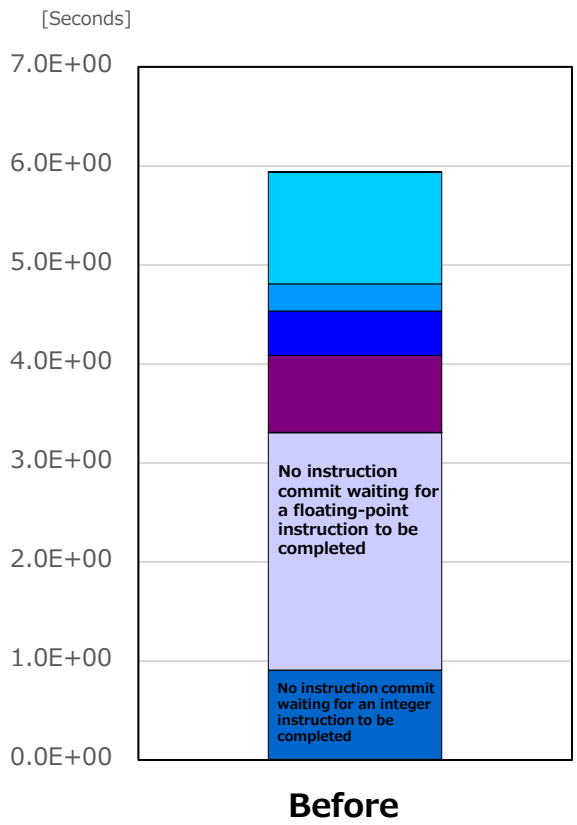
FUJITSU

Unrolling and software pipelining are not working effectively because the number of iterations is small. Consequently, the "No instruction commit waiting for a floating-point instruction to be completed" and "No instruction commit waiting for an integer instruction to be completed" events occur many times.

Source Before Improvement

```
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 1.62, ITR: 176,
                                     MVE: 6, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
31        2v   for (i = 0; i < n; i++){
32        2v     b[i] = c0 + a[i]*(c1 + a[i]*(c2 + a[i]*(c3 + a[i]*
33                  (c4 + a[i]*(c5 + a[i]*(c6 + a[i]*(c7 + a[i]*
34                  (c8 + a[i]*c9))))))));
35        2v   }
36           }
```

Number of loop iterations n = 40, while number of unrolls = 2



Problem: Small number of loop iterations

- Software-pipelined loop not executed
The software pipelining condition "ITR: 176" is not satisfied.

- Inappropriate number of unrolls
2 unrolls x 8 (SIMD) = 16
The original loop is executed for 8 iterations.

The execution of the original loop greatly affects performance.

Specifying the Appropriate Number of Unrolls and Suppressing Software Pipelining (Optimization Control Line Tuning)

FUJITSU

Suppress software pipelining and specify a number of unrolls appropriate to the number of iterations to appropriately schedule instructions. The result is reduction of the "No instruction commit waiting for a floating-point instruction to be completed" and "No instruction commit waiting for an integer instruction to be completed" events.

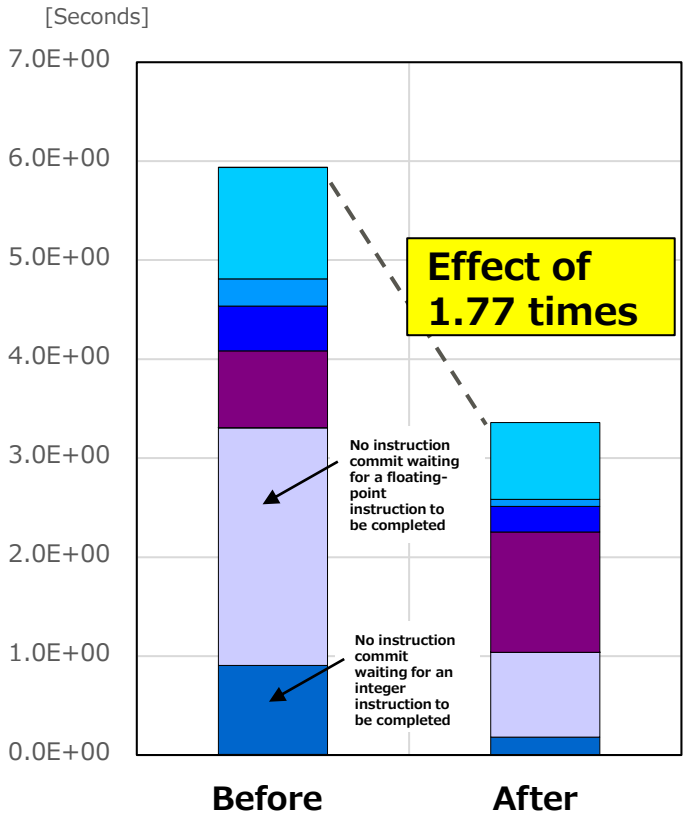
Source After Improvement (Optimization Control Line Tuning)

```
31          #pragma loop unroll 5
32          #pragma loop noswp
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    PREFETCH(HARD) Expected by compiler
            <<<      (unknown)
            <<< Loop-information  End >>>
33        5v   for (i = 0; i < n; i++){
34        5v     b[i] = c0 + a[i]*(c1 + a[i]*(c2 + a[i]*(c3 + a[i]*
35                   (c4 + a[i]*(c5 + a[i]*(c6 + a[i]*(c7 + a[i]*
36                   (c8 + a[i]*c9)))))));
37        5v   }
38           }
```

5 specified as number of unrolls

Software pipelining suppressed

Specifying 5 as the number of unrolls results in
5 unrolls x 8 (SIMD) = 40.
The unrolled loop is executed for all 40 iterations.



Specifying the Appropriate Number of Unrolls and Suppressing Software Pipelining (Optimization Control Line)

FUJITSU

Specify the following optimization specifiers. Alternatively, you can specify compiler options.

Optimization Specifier (Fortran)	Meaning	Optimization Control Line Specifiable?			
		By Program	By DO Loop	By Statement	By Array Assignment Statement
UNROLL(*n*)	Unrolls a DO loop. *n* is a decimal number (2 to 100) that represents the number of unrolls (multiplicity).	No	Yes	No	No
NOSWP	Disables the software pipelining function.	Yes	Yes	No	Yes

Optimization Specifier (C/C++)	Meaning	Optimization Control Line Specifiable?			
		global	procedure	loop	statement
unroll(*n*)	Unrolls a loop. *n* is a decimal number (2 to 100) that represents the number of unrolls (multiplicity).	No	No	Yes	No
noswp	Disables the software pipelining function.	Yes	Yes	Yes	No

Compiler Option	Functional Description
-Kunroll[=*N*] (2≦*N*≦100)	Specifies optimization of loop unrolling. *N* specifies the upper limit on the number of loop unrolls. If *N* is not specified, the compiler automatically decides the best value. If the -O0 or -O1 option is enabled, the default is -Knounroll. If the -O2 or higher option is enabled, the default is -Kunroll.
-Knoswp	Specifies that software pipelining not be optimized.

■**Use example**

```
$ frtpx -Kfast,parallel sample.f90 -Kunroll=5,noswp
$ fccpx -Kfast,parallel sample.f90 -Kunroll=5,noswp
```

◆**Note**

Unrolling optimization is not available in Clang Mode.

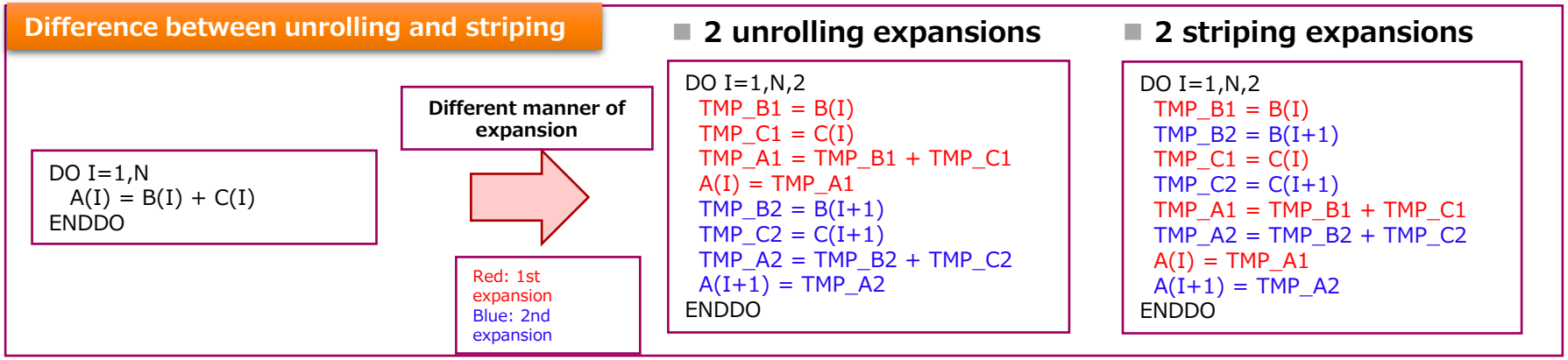
FUJITSU

Specifying the Number of Striping (Interleaving) Expansions and Suppressing Software Pipelining

- Loop Expansion
- Specifying the Number of Striping (Interleaving) Expansions and Suppressing Software Pipelining (Before Improvement)
- Specifying the Number of Striping (Interleaving) Expansions and Suppressing Software Pipelining (Optimization Control Line Tuning)
- Specifying the Number of Striping (Interleaving) Expansions and Suppressing Software Pipelining (Optimization Control Line)

Loop Expansion

- The two types of loop expansion are as follows:
  - Unrolling
  - Striping



- Points
  - Since coordination with SIMDization and software pipelining can be expected to have an effect, we recommend first applying unrolling. If there is no effect, apply striping.
  - Striping uses more registers than unrolling. Therefore, execution performance may degrade when the stripe length n is increased.
  - If the -Kstriping and -Kunroll options are concurrently specified, the one specified later is enabled.

Fortran
Specifying the Number of Striping (Interleaving) Expansions and Suppressing Software Pipelining (Before Improvement)

FUJITSU

Unrolling and software pipelining are not working effectively because the number of iterations is small. Consequently, the "No instruction commit waiting for a floating-point instruction to be completed" and "No instruction commit waiting for an integer instruction to be completed" events occur many times.

Source Before Improvement

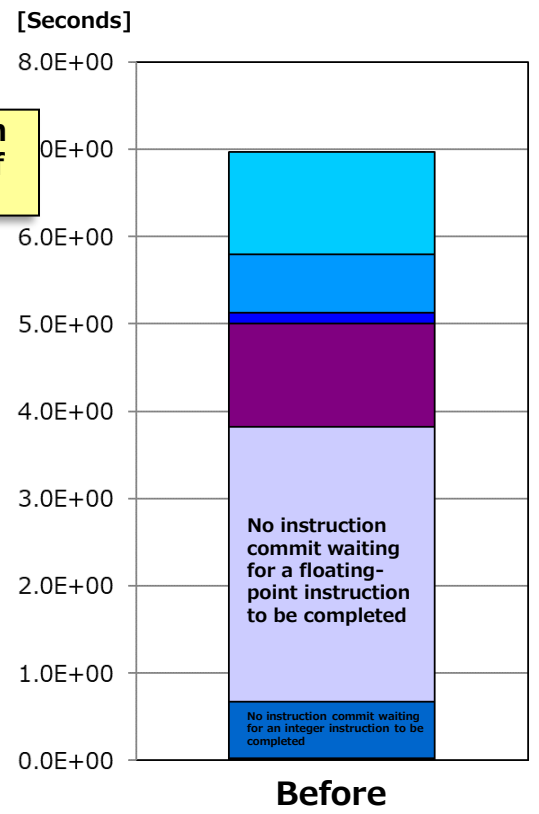
```
                <<< Loop-information Start >>>
                <<< [PARALLELIZATION]
                <<<   Standard iteration count: 40
                <<< [OPTIMIZATION]
                <<<   SIMD(VL: 8)
                <<<   SOFTWARE PIPELINING(IPC: 1.62, ITR: 176,
                                          MVE: 6, POL: S)
                <<<   PREFETCH(HARD) Expected by compiler :
                <<<     a, b
                <<< Loop-information  End >>>
39    1  pp  2v     do i = 1 , n
40    1   p  2v       b(i) = c0 + a(i)*(c1 + a(i)*(c2 + a(i)*(c3 + a(i)*
41    1           &      (c4 + a(i)*(c5 + a(i)*(c6 + a(i)*(c7 + a(i)*
42    1           &      (c8 + a(i)*c9))))))))
43    1   p  2v       enddo
```

Number of loop iteration n = 40, while number of unrolls = 2

Problem: Small number of loop iterations

- **Software-pipelined loop not executed**
  The software pipelining condition "ITR: 176" is not satisfied.
- **Inappropriate number of unrolls**
  2 unrolls x 8 (SIMD) = 16
  The original loop is executed for 8 iterations.

  The execution of the original loop greatly affects performance.



Fortran

Specifying the Number of Striping (Interleaving) Expansions and Suppressing Software Pipelining (Optimization Control Line Tuning)

FUJITSU

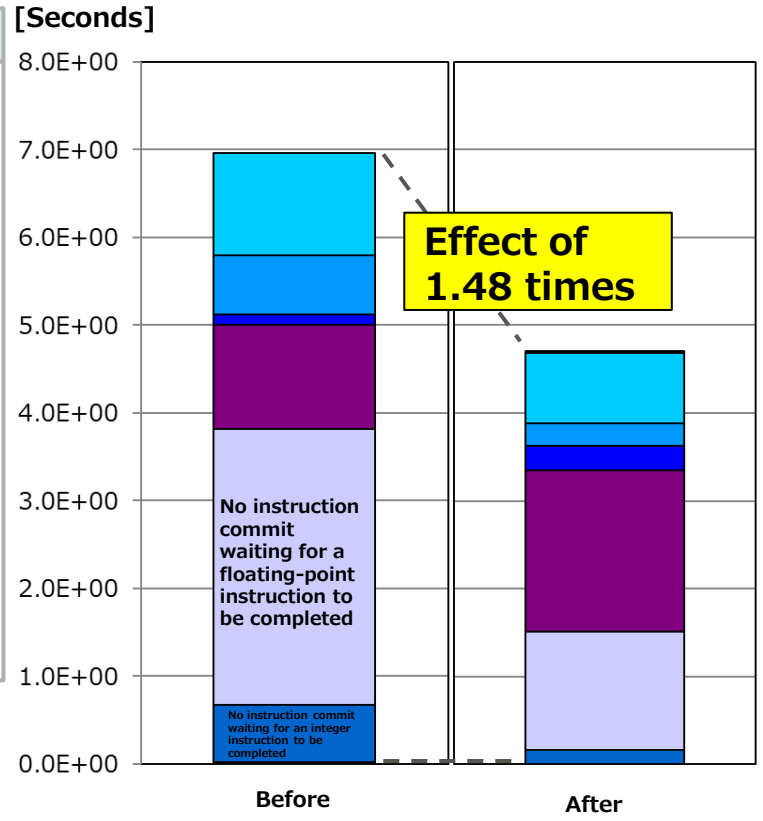
Suppress software pipelining and specify a number of striping expansions appropriate to the number of iterations to appropriately schedule instructions. The result is reduction of the "No instruction commit waiting for a floating-point instruction to be completed" and "No instruction commit waiting for an integer instruction to be completed" events.

Source After Improvement (Optimization Control Line Tuning)

```
39                !ocl striping(5)
40                !ocl nounroll
41                !ocl noswp
        <<< Loop-information Start >>>
        <<<  [PARALLELIZATION]
        <<<    Standard iteration count: 552
        <<<  [OPTIMIZATION]
        <<<    SIMD(VL: 8)
        <<<    PREFETCH(HARD) Expected by compiler :
        <<<      a, b
        <<< Loop-information  End >>>
42    1  pp   5v    do i = 1 , n
43    1   p   5v       b(i) = c0 + a(i)*(c1 + a(i)*(c2 + a(i)*(c3 + a(i)*
44    1               &     (c4 + a(i)*(c5 + a(i)*(c6 + a(i)*(c7 + a(i)*
45    1               &     (c8 + a(i)*c9)))))))
46    1   p   5v    enddo
```

5 specified as number of striping expansions

Unrolling and software pipelining suppressed



Specifying 5 as the number of striping expansions results in

5 striping expansions x 8 (SIMD) = 40.

The striped loop is executed for all 40 iterations.

C/C++ Specifying the Number of Striping (Interleaving) Expansions and Suppressing Software Pipelining (Before Improvement)

FUJITSU

Unrolling and software pipelining are not working effectively because the number of iterations is small. Consequently, the "No instruction commit waiting for a floating-point instruction to be completed" and "No instruction commit waiting for an integer instruction to be completed" events occur many times.

Source Before Improvement

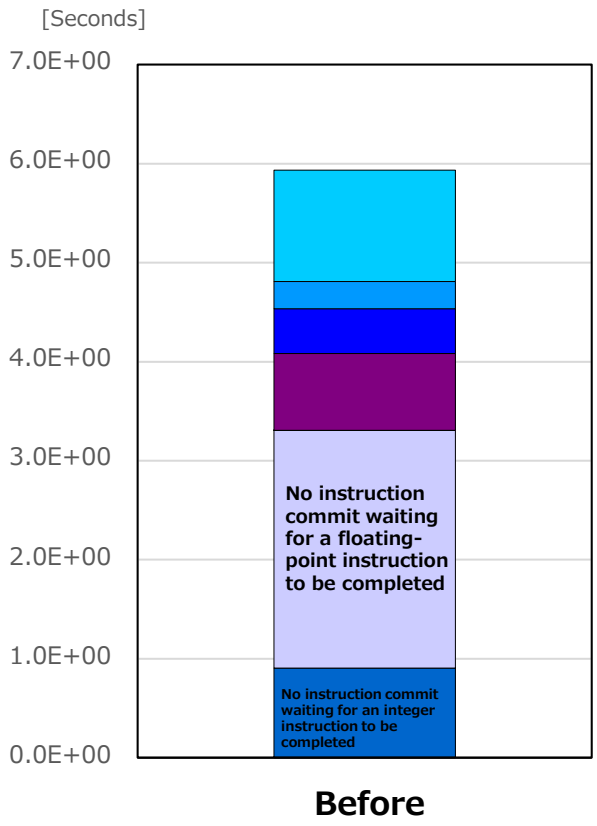
```
<<< Loop-information Start >>>
        <<<  [OPTIMIZATION]
        <<<    SIMD(VL: 8)
        <<<    SOFTWARE PIPELINING(IPC: 1.62, ITR: 176,
                        MVE: 6, POL: S)
        <<<    PREFETCH(HARD) Expected by compiler :
        <<<      (unknown)
        <<< Loop-information  End >>>
33      2v    for (i = 0; i < n; i++){
34      2v      b[i] = c0 + a[i]*(c1 + a[i]*(c2 + a[i]*(c3 + a[i]*
35                 (c4 + a[i]*(c5 + a[i]*(c6 + a[i]*(c7 + a[i]*
36                 (c8 + a[i]*c9)))))));
37      2v    }
38          }
```

Number of loop iteration n = 40, while number of unrolls = 2

Problem: Small number of loop iterations

- **Software-pipelined loop not executed**
  The software pipelining condition "ITR: 176" is not satisfied.
- **Inappropriate number of unrolls**
  2 unrolls x 8 (SIMD) = 16
  The original loop is executed for 8 iterations.

The execution of the original loop greatly affects performance.



C/C++ Specifying the Number of Striping (Interleaving) Expansions and Suppressing Software Pipelining (Optimization Control Line Tuning)

FUJITSU

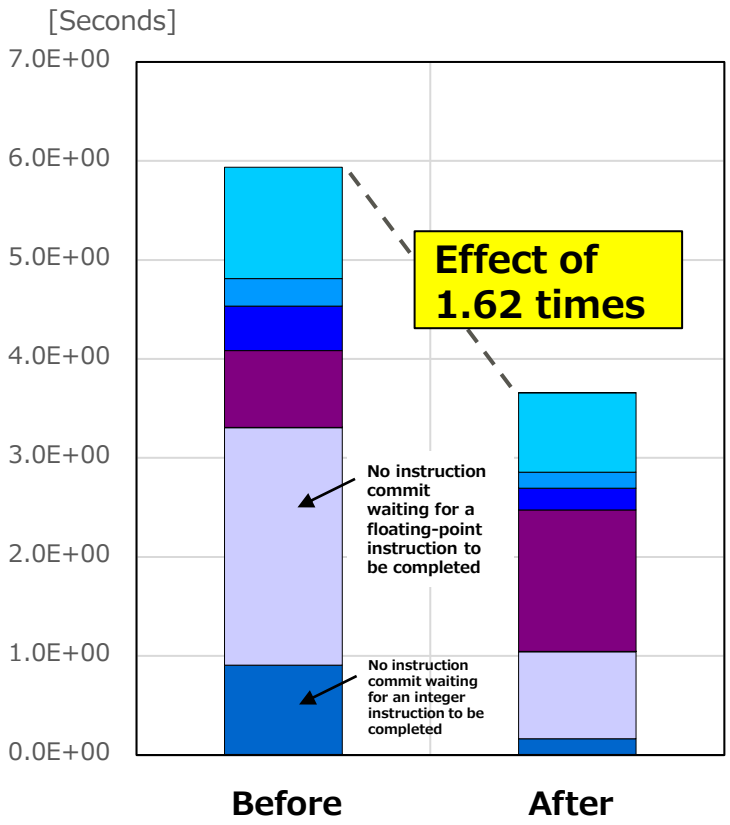
Suppress software pipelining and specify a number of striping expansions appropriate to the number of iterations to appropriately schedule instructions. The result is reduction of the "No instruction commit waiting for a floating-point instruction to be completed" and "No instruction commit waiting for an integer instruction to be completed" events.

Source After Improvement (Optimization Control Line Tuning)

```
33          #pragma loop striping 5
34          #pragma loop nounroll
35          #pragma loop noswp
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    STRIPING
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
36        v   for (i = 0; i < n; i++){
37        v     b[i] = c0 + a[i]*(c1 + a[i]*(c2 + a[i]*(c3 + a[i]*
38                  (c4 + a[i]*(c5 + a[i]*(c6 + a[i]*(c7 + a[i]*
39                  (c8 + a[i]*c9)))))));
40        v   }
41          }
```

5 specified as number of striping expansions

Unrolling and software pipelining suppressed



Specifying 5 as the number of striping expansions results in

5 striping expansions x 8 (SIMD) = 40.

The striped loop is executed for all 40 iterations.

Specifying the Number of Striping (Interleaving) Expansions and Suppressing Software Pipelining (Optimization Control Line)

FUJITSU

Specify the following optimization specifiers. Alternatively, you can specify compiler options.

Optimization Specifier (Fortran)	Meaning	Optimization Control Line Specifiable?			
		By Program	By DO Loop	By Statement	By Array Assignment Statement
STRIPING[(n)]	Enables the loop striping function. *n* is a decimal number (2 to 100) that represents the number of expansions (multiplicity).	Yes	Yes	No	Yes
NOSWP	Disables the software pipelining function.	Yes	Yes	No	Yes

Optimization Specifier (C/C++)	Meaning	Optimization Control Line Specifiable?			
		global	procedure	loop	statement
striping[(n)]	Enables the loop striping function. *n* is a decimal number (2 to 100) that represents the number of expansions (multiplicity).	Yes	Yes	Yes	No
noswp	Disables the software pipelining function.	Yes	Yes	Yes	No

Compiler Option	Functional Description
-Kstriping[=*N*] (2≦*N*≦100)	Specifies whether or not to optimize loop striping. You can specify the stripe length (number of expansions) in *N*. *N* can be a value from 2 to 100. If no value is specified in *N*, the compiler automatically decides a value. If the number of loop iterations in the source program is known and the value specified in *N* exceeds the number of iterations, the number of expansions automatically decided by the compiler is valid. The default is -Knostriping.
-Knoswp	Specifies that software pipelining not be optimized.

■Use example

```
$ frtpx -Kfast,parallel sample.f90 -Kstriping=5,noswp
$ fccpx -Kfast,parallel sample.f90 -Kstriping=5,noswp
```

◆**Note**

Striping optimization is not available in Clang Mode.

FUJITSU

Software Pipelining in an Outer Loop

- What is Software Pipelining in an Outer Loop?
- Software Pipelining in an Outer Loop (Before Improvement)
- Software Pipelining in an Outer Loop (Source Tuning)
- Software Pipelining in an Outer Loop (Using CLONE)
- Software Pipelining in an Outer Loop (Using CLONE) (Before Improvement)
- Software Pipelining in an Outer Loop (Using CLONE) (Source Tuning)

What is Software Pipelining in an Outer Loop?

- **If the number of iterations of the innermost loop is small, you can facilitate software pipelining in its outer loops through strip mining or other means to make the number of iterations equal to the SIMD length.**

Source Before Improvement

```
24   3  p          do j=1,M
             <<< Loop-information Start >>>
             <<<  [OPTIMIZATION]
             <<<    SIMD(VL: 8)
             <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                                        MVE: 7, POL: S)
             <<<    PREFETCH(HARD) Expected by compiler :
             <<<      b, a
             <<< Loop-information  End >>>
25   4  p  2v         do i=1,N
26   4  p  2v           a(i,j,k)=a(i,j,k)+c*b(i,j,k)
27   4  p  2v         enddo
28   3  p          enddo
```

Source After Improvement

```
25   3  p          do ii=1,N,blk
             <<< Loop-information Start >>>
             <<<  [OPTIMIZATION]
             <<<    SOFTWARE PIPELINING(IPC: 0.31, ITR: 192,
                                        MVE: 2, POL: L)
             <<<    PREFETCH(HARD) Expected by compiler :
             <<<      b, a
             <<< Loop-information  End >>>
26   4  p  8          do j=1,M
             <<< Loop-information Start >>>
             <<<  [OPTIMIZATION]
             <<<    SIMD(VL: 8)
             <<<    FULL UNROLLING
             <<< Loop-information  End >>>
27   5  p  fv           do i=ii,ii+blk-1
28   5  p  fv             a(i,j,k)=a(i,j,k)+c*b(i,j,k)
29   5  p  fv           enddo
30   4  p  8          enddo
31   3  p          enddo
```

The above example is valid for fixed-length SIMD.
The SIMD lengths when the SIMD width is 512 [bits] are as follows.

SIMD lengths when SIMD width (vector length) = 512 [bits]

Data Type	SIMD Length
Double-precision type	8
Single-precision type	16
Half-precision type	32
1-byte type	64

Fortran Software Pipelining in an Outer Loop (Before Improvement)

FUJITSU

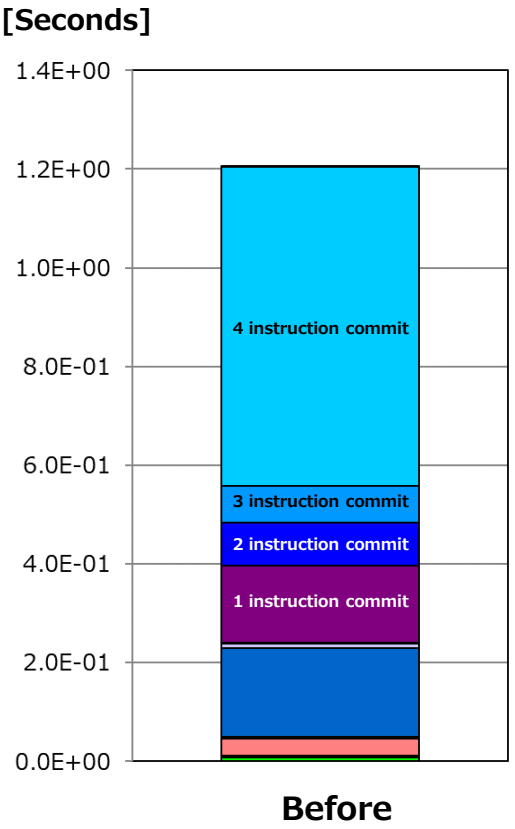
Software pipelining is not applied because the number (N) of iterations of the innermost loop is small compared with the software pipelining condition.

Source Before Improvement

```
2                integer,parameter::N=16,M=200,L=48
 :
18                real(8)::a(N,M,L),b(N,M,L)
19                real(8),parameter::c=0.5
20               !$omp parallel private(iter,i,j,k)
21    1            do iter=1,itmax
22    1           !$omp do
23    2  p           do k=1,L
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information
24    3  p             do j=1,M
                <<< Loop-information S
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                                                  MVE: 7, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
25    4  p  2v           do i=1,N
26    4  p  2v             a(i,j,k)=a(i,j,k)+c*b(i,j,k)
27    4  p  2v           enddo
28    3  p             enddo
29    2  p           enddo
30    1            !$omp enddo nowait
31    1             enddo
32                 !$omp end parallel
```

Does not enter software pipelining route because number (N) of iterations of innermost loop is smaller than ITR:192

N = 16



	GFLOPS	Effective instruction
Before	50.98	7.53E+10

Fortran Software Pipelining in an Outer Loop (Source Tuning)

FUJITSU

Strip mining fixes the innermost loop at the SIMD length. The result is facilitated software pipelining in its outer loops, improved operation efficiency, and more effective instructions.

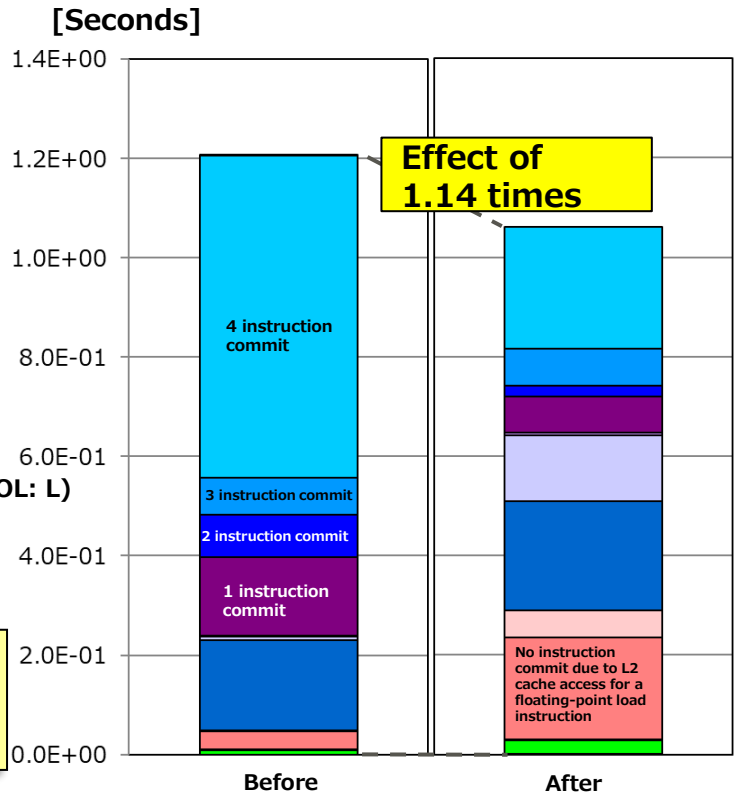
Source After Improvement

```
2                integer,parameter::N=16,M=200,L=48
:
18                real(8)::a(N,M,L),b(N,M,L)
19                real(8),parameter::c=0.5
20                integer,parameter::blk=8
21               !$omp parallel private(iter,ii,i,j,k)
22    1            do iter=1,itmax
23    1           !$omp do
24    2  p          do k=1,L
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
25    3  p            do ii=1,N,blk
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SOFTWARE PIPELINING(IPC: 0.31, ITR: 192, MVE: 2, POL: L)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
26    4  p  8           do j=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    FULL UNROLLING
                <<< Loop-information  End >>>
27    5  p  fv            do i=ii,ii+blk-1
28    5  p  fv              a(i,j,k)=a(i,j,k)+c*b(i,j,k)
29    5  p  fv            enddo
30    4  p  8           enddo
31    3  p             enddo
32    2  p           enddo
33    1             !$omp enddo nowait
34    1             enddo
35                 !$omp end parallel
```

Software pipelining applied

M = 200

Number of innermost loop iterations is fixed at SIMD length



	GFLOPS	Effective instruction
Before	50.98	7.53E+10
After	57.95	3.19E+10

Software Pipelining in an Outer Loop (Using CLONE)

FUJITSU

If the number of iterations of the innermost loop is a small fixed value, you can also use clone tuning.

- **What is clone tuning?**
  A tuning method that facilitates optimizations such as full unrolling by using the CLONE specifier to generate a conditional branch based on a variable value for the loop

Example of Source

```
!ocl clone(N==8)
do i=1,N
   A(i) = i
enddo
```

After Optimization (Image of Source)

```
If (N==8) then
   do i=1,8
      A(i)=i
   endo
else
   do i=1,N
      A(i)=i
   enddo
endif
```

Loop cloned when N==8, which fixes loop length and facilitates other optimizations

Optimization Specifier (Fortran)	Meaning	Optimization Control Line Specifiable?			
		By Program	By DO Loop	By Statement	By Array Assignment Statement
CLONE(var==n1[,n2]…)	Specifies that a conditional branch be generated as specified in arguments, assuming the variable *var* is invariable in the loop, and that optimization that clones the loop in an IF clause be performed. The conditional expression has equality with the variable *var* in the 1st argument and the values *n1*[,*n2*] and so on specified in the 2nd and subsequent arguments.	No	Yes	No	Yes

Optimization Specifier (C/C++)	Meaning	Optimization Control Line Specifiable?			
		global	procedure	loop	statement
clone(var==n1[,n2]…)	Specifies that a conditional branch be generated as specified in arguments, assuming the variable *var* is invariable in the loop, and that optimization that clones the loop in an IF clause be performed. The conditional expression has equality with the variable *var* in the 1st argument and the values *n1*[,*n2*] and so on specified in the 2nd and subsequent arguments.	No	No	Yes	No

Fortran Software Pipelining in an Outer Loop (Using CLONE) (Before Improvement)

FUJITSU

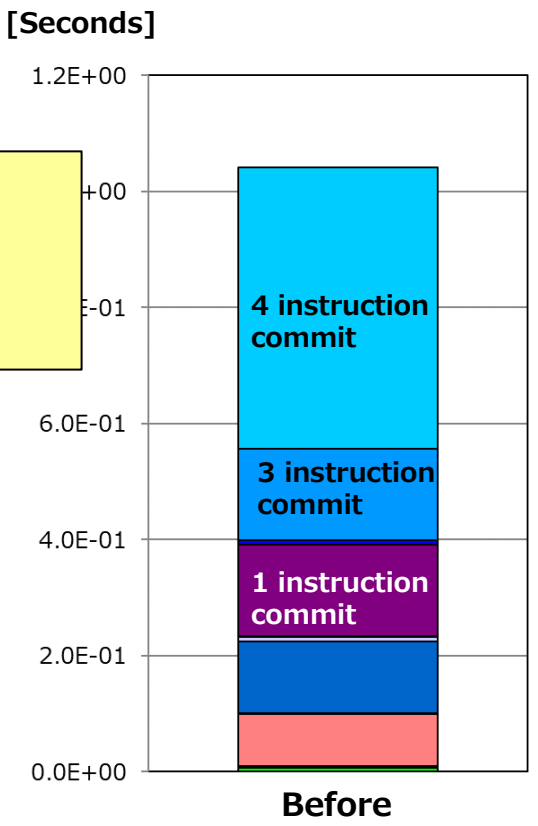
A condition for applying software pipelining is not satisfied because the number (N) of iterations of the innermost loop is small.

Source Before Improvement

```
2                integer,parameter::N=8,M=240,L=48
:
18                real(8)::a(N,M,L),b(N,M,L)
19                real(8),parameter::c=0.5
20               !$omp parallel private(iter,i,j,k)
21    1            do iter=1,itmax
22    1           !$omp do
23    2  p          do k=1,L
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
24    3  p            do j=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                                                  MVE: 7, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
25    4  p  2v           do i=1,N
26    4  p  2v             a(i,j,k)=a(i,j,k)+c*b(i,j,k)
27    4  p  2v           enddo
28    3  p            enddo
29    2  p          enddo
30    1           !$omp enddo nowait
31    1            enddo
32               !$omp end parallel
```

Does not enter software pipelining route because number (N) of innermost loop iterations is smaller than ITR:192

N = 8



	GFLOPS	Effective instruction
Before	29.51	6.18E+10

Fortran Software Pipelining in an Outer Loop (Using CLONE) (Source Tuning)

FUJITSU

CLONE fixes the number of iterations of the innermost loop at the SIMD length. The result is facilitated software pipelining in its outer loops, improved operation efficiency, and more effective instructions.

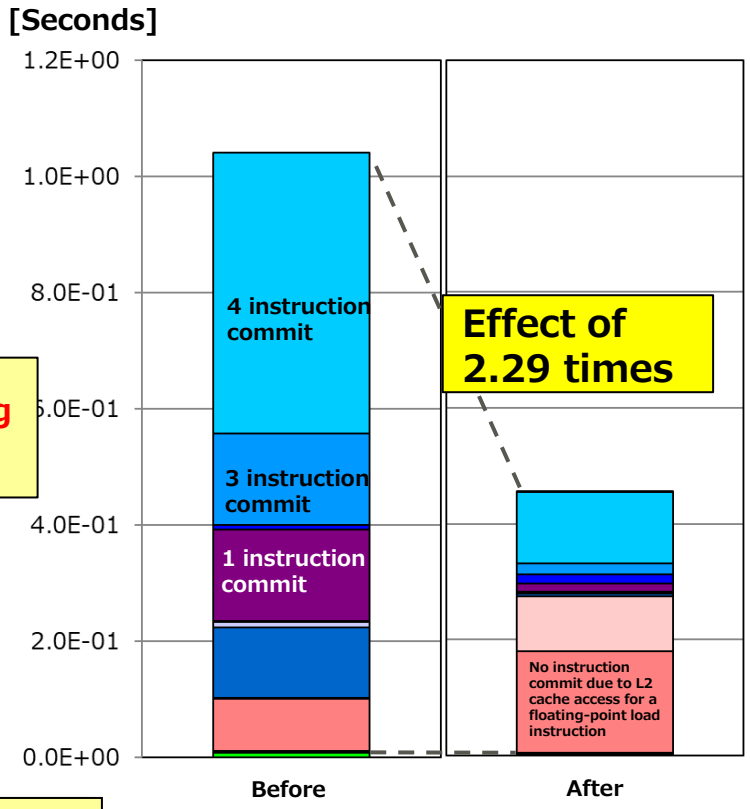
Source After Improvement (Source Tuning)

```
2                integer,parameter::N=8,M=240,L=48
:
18                real(8)::a(N,M,L),b(N,M,L)
19                real(8),parameter::c=0.5
20                integer NN
21                NN = N
22               !$omp parallel private(iter,i,j,k)
23    1            do iter=1,itmax
24    1           !$omp do
25    1           !ocl clone(NN==8)
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    CLONE
                <<< Loop-information  End >>>
26    2  p          do k=1,L
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SOFTWARE PIPELINING(IPC: 3.75, ITR: 208,
                                           MVE: 6, POL: S)
                <<< Loop-information  End >>>
27    3  p  2        do j=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                                           MVE: 7, POL: S)
                <<< Loop-information  End >>>
28    4  p  2v         do i=1,NN
29    4  p  2v           a(i,j,k)=a(i,j,k)+c*b(i,j,k)
30    4  p  2v         enddo
31    3  p  2        enddo
32    2  p          enddo
33    1           !$omp enddo nowait
34    1            enddo
35               !$omp end parallel
```

Software pipelining applied

M = 240

Number of innermost loop iterations is fixed at SIMD length



	GFLOPS	Effective instruction
Before	29.51	6.18E+10
After	67.60	1.44E+10

Software Pipelining in an Outer Loop (Using CLONE) (Before Improvement)

FUJITSU

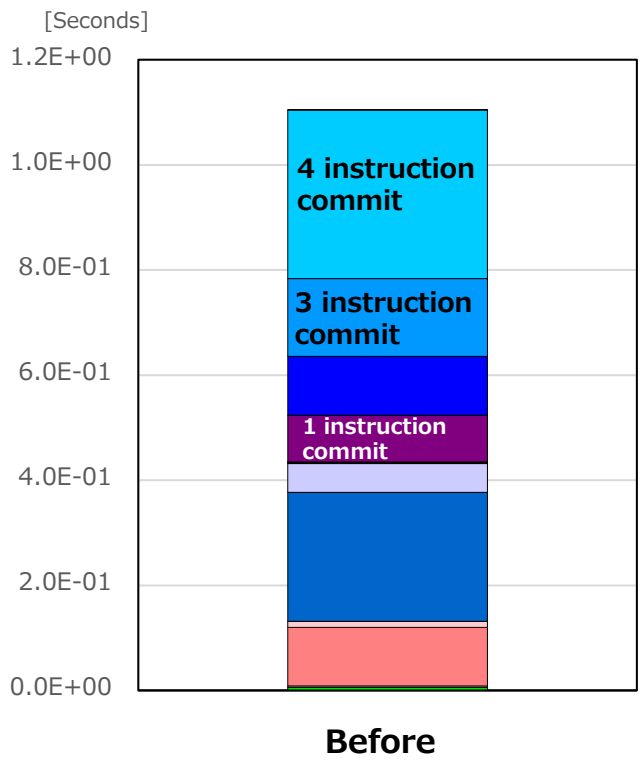
A condition for applying software pipelining is not satisfied because the number (N) of iterations of the innermost loop is small.

Source Before Improvement

```
35          #pragma omp parallel private(iter,i,j,k)
36          {
37            for (iter = 0; iter < itmax; iter++){
38          #pragma omp for nowait
39    p         for (k = 0; k < L; k++){
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Ex
                <<<      (unknown)
                <<< Loop-information  End >>>
40    p           for (j = 0; j < M; j++){
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                           MVE: 7, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      (unknown)
                <<< Loop-information  End >>>
41    p    2v         for (i = 0; i < N; i++){
42    p    2v           a[k][j][i] = a[k][j][i] + c * b[k][j][i];
43    p    2v         }
44    p             }
45    p           }
46              }
47            }
```

Does not enter software pipelining route because number (N) of innermost loop iterations is smaller than ITR:192

N = 8



Statistics	GFLOPS	Effective instruction
Before	33.39	4.87E+10

C/C++ Software Pipelining in an Outer Loop (Using CLONE) (Source Tuning)

FUJITSU

CLONE fixes the number of iterations of the innermost loop at the SIMD length. The result is facilitated software pipelining in its outer loops, improved operation efficiency, and more effective instructions.

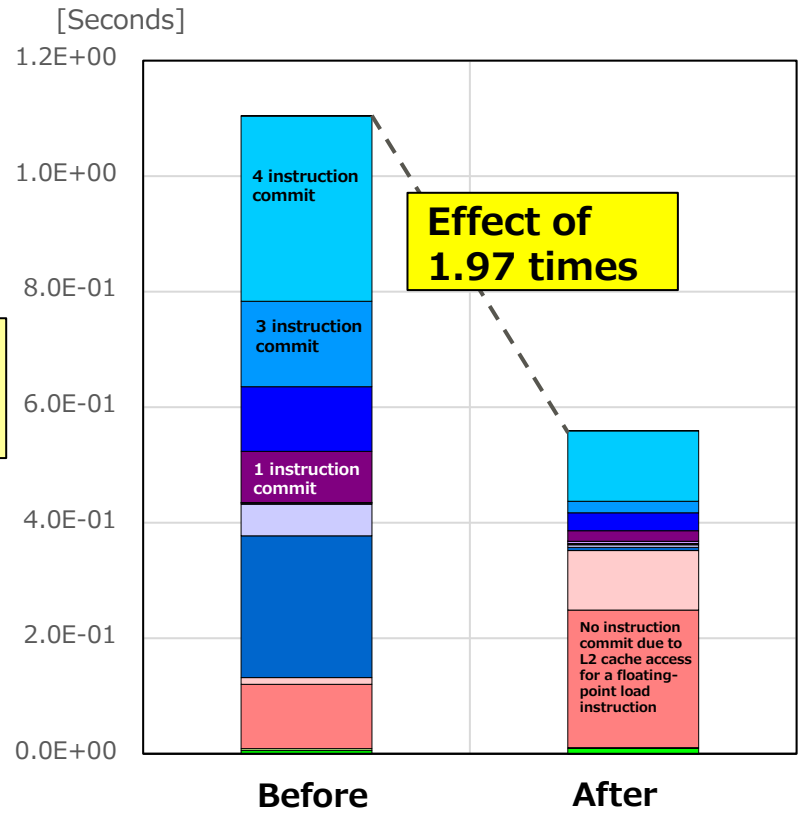
Source After Improvement (Source Tuning)

```
37          #pragma omp parallel private(iter,i,j,k)
38          {
39            for (iter = 0; iter < itmax; iter++){
40          #pragma omp for nowait
41          #pragma loop clone NN==8
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    CLONE
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
42   p          for (k = 0; k < L; k++){
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 176,
                                MVE: 6, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
43   p     2       for (j = 0; j < M; j++){
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                                 MVE: 7, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
44   p     2v        for (i = 0; i < NN; i++){
45   p     2v          a[k][j][i] = a[k][j][i] + c * b[k][j][i];
46   p     2v        }
47   p     2       }
48   p           }
49             }
50           }
```

Software pipelining applied

M = 240

Number of innermost loop iterations is fixed at SIMD length



Statistics	GFLOPS	Effective instruction
Before	33.39	4.87E+10
After	65.89	1.50E+10

FUJITSU

Rerolling

- What is Rerolling?
- Rerolling (Before Improvement)
- Rerolling (Source Tuning)

What is Rerolling?

FUJITSU

Rerolling is a tuning method that facilitates the optimization of a loop by restoring unrolled statements to loop statements

Unrolling
Frequent tuning of code that does not assume SIMDization

Normal Loop

```
do k=1,m
  do j=1,n
    do i=1,8
      a(i,j,k) = b(i,j,k) + a(i,j,k)
    enddo
  enddo
enddo
```

Manual Unrolling

```
do k=1,m
  do j=1,n
    a(1,j,k) = b(1,j,k) + a(1,j,k)
    a(2,j,k) = b(2,j,k) + a(2,j,k)
    a(3,j,k) = b(3,j,k) + a(3,j,k)
    a(4,j,k) = b(4,j,k) + a(4,j,k)
    a(5,j,k) = b(5,j,k) + a(5,j,k)
    a(6,j,k) = b(6,j,k) + a(6,j,k)
    a(7,j,k) = b(7,j,k) + a(7,j,k)
    a(8,j,k) = b(8,j,k) + a(8,j,k)
  enddo
enddo
```

Rerolling

```
do k=1,m
  do j=1,n
    do i=1,8
      a(i,j,k) = b(i,j,k) + a(i,j,k)
    enddo
  enddo
enddo
```

Downside of unrolled code
Many Gather Load and Scatter Store instructions due to distant access

Rerolling
Restoring unrolled loop to original loop

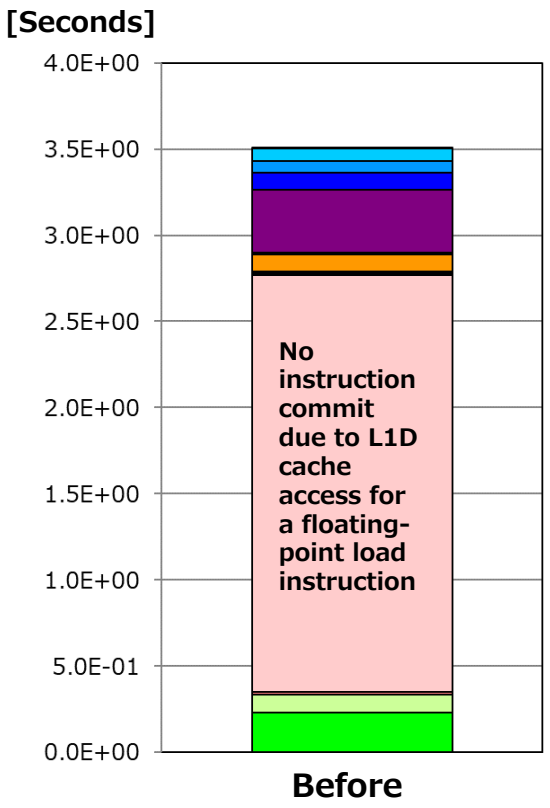
Fortran Rerolling (Before Improvement)

A loop is manually unrolled. The result is a lot of Gather Load and Scatter Store instructions. Consequently, the "No instruction commit due to L1D cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
40                 real(8) :: a(8,N,M),b(8,N,M)
41
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 2
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
42    1  pp           DO K=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.04, ITR: 40,
                                                MVE: 3, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
43    2  p   v          DO J=1,N
44    2  p   v            a(1,J,K) = b(1,J,K) + a(1,J,K)
45    2  p   v            a(2,J,K) = b(2,J,K) + a(2,J,K)
46    2  p   v            a(3,J,K) = b(3,J,K) + a(3,J,K)
47    2  p   v            a(4,J,K) = b(4,J,K) + a(4,J,K)
48    2  p   v            a(5,J,K) = b(5,J,K) + a(5,J,K)
49    2  p   v            a(6,J,K) = b(6,J,K) + a(6,J,K)
50    2  p   v            a(7,J,K) = b(7,J,K) + a(7,J,K)
51    2  p   v            a(8,J,K) = b(8,J,K) + a(8,J,K)
52    2  p   v          ENDDO
53    1  p           ENDDO
```

N = 128
M = 250



Rerolling (Source Tuning)

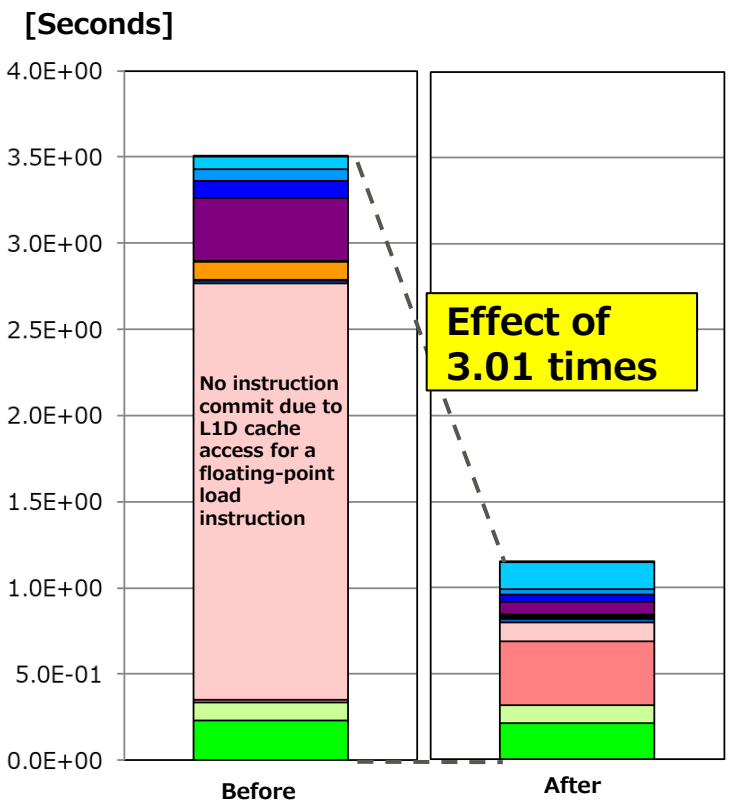
Data access becomes sequential due to rerolling (restoring to a loop), which facilitates optimizations such as effective software pipelining, SIMDization, and loop unrolling.

Source After Improvement (Source Tuning)

```
40                 real(8) :: a(8,N,M),b(8,N,M)
41
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration c
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                                           MVE: 7, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a, b
                <<< Loop-information  End >>>
42    1  pp  2v       DO K=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<< Loop-information  End
43    2   p   2         DO J=1,N
                <<< Loop-information Start
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<< Loop-information  End >>>
44    3   p   2           DO I=1,8
45    3   p   2v            a(I,J,K) = b(I,J,K) + a(I,J,K)
46    3   p   2v          ENDDO
47    2   p             ENDDO
48    1   p           ENDDO
```

Transformation into 1 loop

SIMDization and loop unrolling facilitated



Rerolling (Before Improvement)

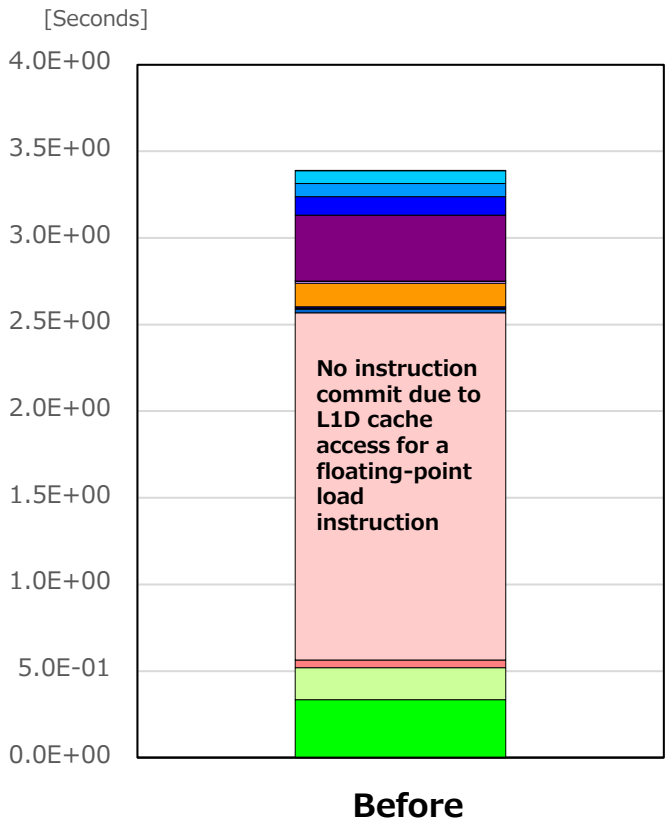
FUJITSU

A loop is manually unrolled. The result is a lot of Gather Load and Scatter Store instructions. Consequently, the "No instruction commit due to L1D cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

N=128
M=250

```
50          #pragma omp parallel for private(j)
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
51   p       for( k=0;k<M; k++)
52   p       {
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 2.04, ITR: 40,
                                MVE: 3, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
53   p     v     for(j=0; j<N; j++)
54   p     v     {
55   p     v        a[k][j][0] = b[k][j][0] + a[k][j][0];
56   p     v        a[k][j][1] = b[k][j][1] + a[k][j][1];
57   p     v        a[k][j][2] = b[k][j][2] + a[k][j][2];
58   p     v        a[k][j][3] = b[k][j][3] + a[k][j][3];
59   p     v        a[k][j][4] = b[k][j][4] + a[k][j][4];
60   p     v        a[k][j][5] = b[k][j][5] + a[k][j][5];
61   p     v        a[k][j][6] = b[k][j][6] + a[k][j][6];
62   p     v        a[k][j][7] = b[k][j][7] + a[k][j][7];
63   p     v     }
64   p       }
65
66            return;
67          }
```



Rerolling (Source Tuning)

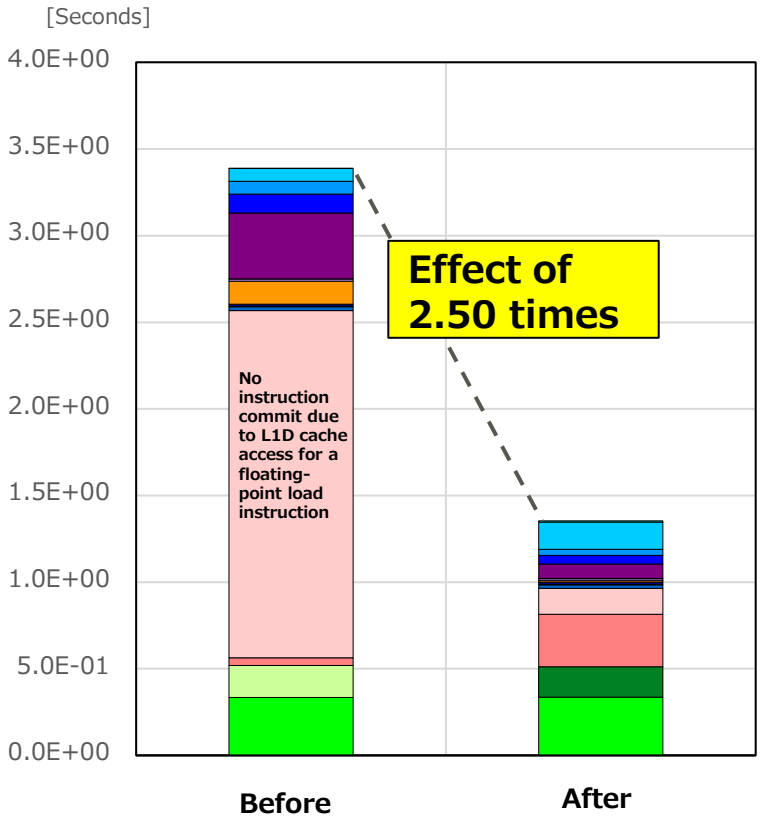
Data access becomes sequential due to rerolling (restoring to a loop), which facilitates optimizations such as effective software pipelining, SIMDization, and loop unrolling.

Source After Improvement (Source Tuning)

```
50          #pragma omp parallel for private(i,j) collapse(3)
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                         MVE: 7, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
51   p    2v   for (k=0; k< M; k++)
52        2    {
53   p    2      for(j=0; j<N; j++)
54        2      {
55   p    2        for(i=0; i<8; i++)
56   p    2v       {
57   p    2v         a[k][j][i] = b[k][j][i] + a[k][j][i];
58   p    2v       }
59   p    2v     }
60   p    2v   }
61
62           return;
63         }
```

Transformation into 1 loop

SIMDization and loop unrolling facilitated



FUJITSU

Loop Unswitching

- What is Loop Unswitching?
- Effect of Loop Unswitching (Before Improvement)
- Effect of Loop Unswitching (After Improvement)

What is Loop Unswitching?

FUJITSU

Loops may contain IF statements that have branches of invariant states. This optimization takes those IF statements out of the loops and creates loops for cases where the IF statement conditions are satisfied/not satisfied.

Source Code

```
!$omp do
  do i=1,n1
!ocl unswitching
    if (n1 >= q) then
      processing 1
    endif
!ocl unswitching
    if(n1 > r) then
      processing 2
    endif
!ocl unswitching
    if(n1 < s) then
   processing 3
    endif
  enddo
!$omp enddo
```

Optimization Image

```
!pattern (1)
if((con1 true).and.(con2 true).and.(con3 true))then
  do i=1,n1
     processing 1
   processing 2
     processing 3
  enddo
endif

!pattern (2)
if((con1 true).and.(con2 true).and.(con3 false))then
  do i=1,n1
     processing 1
   processing 2
  enddo
endif

!pattern (3)
if((con1 true).and.(con2 false).and.(con3 true))then
  do i=1,n1
   processing 1
     processing 3
  enddo
Endif

!pattern (4)
if((con1 ture).and.(con2 false).and.(con3 false))then
  do i=1,n1
     processing 1
  enddo
endif
```

```
!pattern (5)
if((con1 false).and.(con2 true).and.(con3 true))then
  do i=1,n1
   processing 2
     processing 3
   enddo
endif

!pattern (6)
if((con1 false).and.(con2 true).and.(con3 false))then
  do i=1,n1
   processing 2
   enddo
endif

!pattern(7)
if((con1 false).and.(con2 false).and.(con3 true))then
  do i=1,n1
     processing 3
  enddo
endif

!pattern(8)
if((con1 false).and.(con2 false).and.(con3 false))then
  do i=1,n1
  enddo
endif
```

Unrolled to 8 IF statements (DO statements)

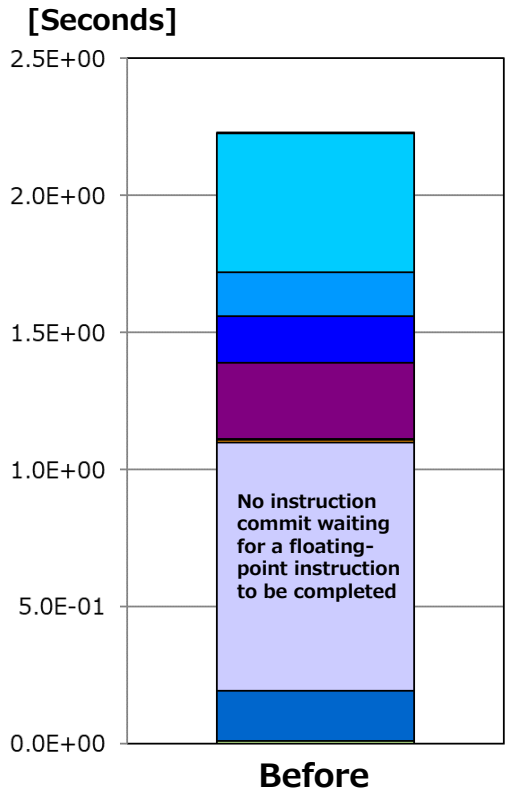
Fortran Effect of Loop Unswitching (Before Improvement)

FUJITSU

SIMDization and software pipelining are not performed effectively because the innermost loop contains an IF statement. Consequently, the "No instruction commit waiting for a floating-point instruction to be completed" event occurs many times.

Source Before Improvement

```
97    1          !$omp do
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELINING(IPC: 1.31, ITR: 96,
                                            MVE: 2, POL: L)
                 <<<    UNSWITCHING
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      a, b
                 <<< Loop-information  End >>>
98    2  p  2v     do i=1,n1
99    2
100     3  p  2v       if (n1 >= q) then
101     3  p  2v         a(i) = c0+b(i)*(c1+b(i)*(c2+b(i)*(c3+b(i)*c4)))
102     3  p  2v       endif
103     2
104     3  p  2v       if(n1 > r) then
105     3  p  2v         a(i) = c0*b(i)/(c1*b(i)/(c2*b(i)/(c3*b(i)/c4)))
106     3  p  2v       endif
107     2
108     3  p  2v       if(n1 < s) then
109     3  p  2v         a(i) = c0+b(i)/(c1+b(i)/(c2+b(i)/(c3+b(i)/c4)))
110     3  p  2v       endif
111     2  p  2v    enddo
112     1          !$omp enddo nowait
```



SIMD

	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
Before	7.47E+10	94.18%

Fortran Effect of Loop Unswitching (After Improvement)

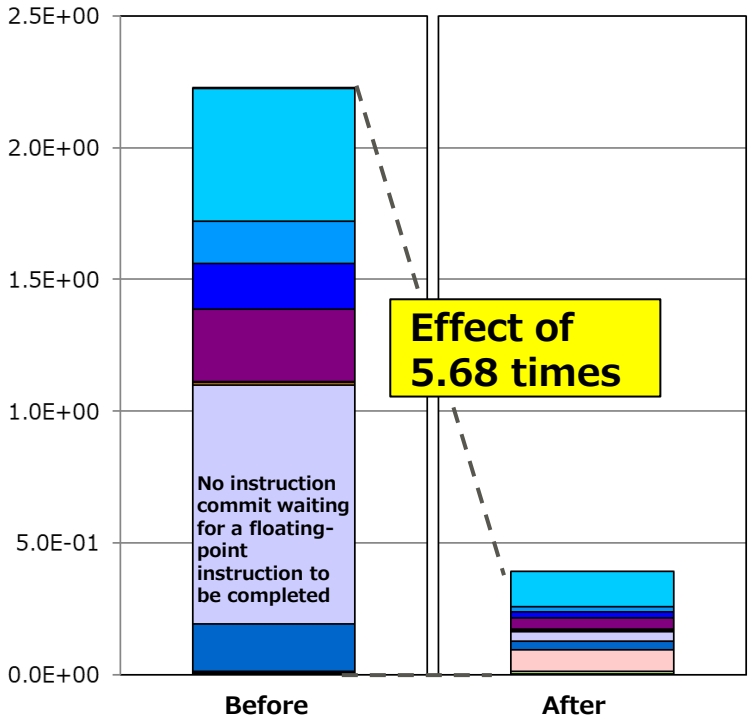
FUJITSU

Specify loop unswitching for IF statements to eliminate branching and facilitate SIMDization and software pipelining. The result is significant improvement of the "No instruction commit waiting for a floating-point instruction to be completed" event.

Source After Improvement (Optimization Control Line Tuning)

```
97    1          !$omp do
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELINING(IPC: 1.04, ITR: 80,
                                                MVE: 3, POL: S)
                 <<<    UNSWITCHING
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      b, a
                 <<< Loop-information  End >>>
98    2  p  2v    do i=1,n1
99    2          !ocl unswitching
100     3  p  2v      if (n1 >= q) then
101     3  p  2v        a(i) = c0+b(i)*(c1+b(i)*(c2+b(i)*(c3+b(i)*c4)))
102     3  p  2v      endif
103     2          !ocl unswitching
104     3  p  2v      if(n1 > r) then
105     3  p  2v        a(i) = c0*b(i)/(c1*b(i)/(c2*b(i)/(c3*b(i)/c4)))
106     3  p  2v      endif
107     2          !ocl unswitching
108     3  p  2v      if(n1 < s) then
109     3  p  2v        a(i) = c0+b(i)/(c1+b(i)/(c2+b(i)/(c3+b(i)/c4)))
110     3  p  2v      endif
111     2  p  2v    enddo
112     1          !$omp enddo nowait
```

[Seconds]



SIMD

	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
Before	7.47E+10	94.18%
After	1.60E+10	75.83%

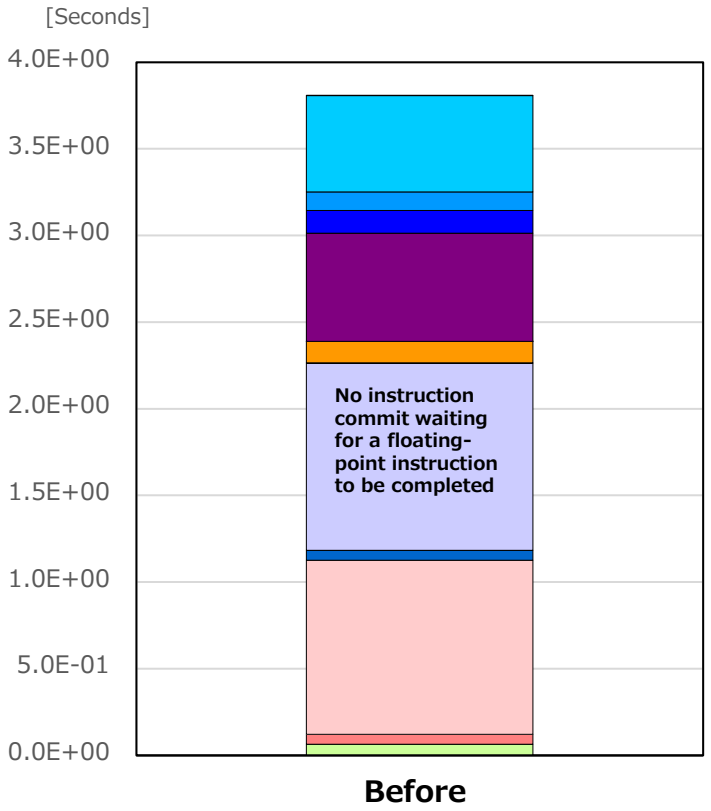
C/C++ Effect of Loop Unswitching (Before Improvement)

FUJITSU

SIMDization and software pipelining are not performed effectively because the innermost loop contains an if statement. Consequently, the "No instruction commit waiting for a floating-point instruction to be completed" event occurs many times.

Source Before Improvement

```
91                #pragma omp for nowait
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    UNSWITCHING
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      (unknown)
                  <<< Loop-information  End >>>
92   p    2v       for(i=0; i<n1; i++)
93   p    2v       {
94   p    2v         if (n1 >= q)
95   p    2v         {
96   p    2v           a[i] = c0+b[i]*(c0+b[i]*(c0+b[i]*(c0+b[i]*c0)));
97   p    2v         }
98
99   p    2v         if(n1 > r)
100  p    2v         {
101  p    2v           a[i] = c0*b[i]/(c0*b[i]/(c0*b[i]/(c0*b[i]/c0)));
102  p    2v         }
103
104  p    2v         if(n1 < s)
105  p    2v         {
106  p    2v           a[i] = c0+b[i]/(c0+b[i]/(c0+b[i]/(c0+b[i]/c0)));
107  p    2v         }
108  p    2v       }
109               }
```



Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
Before	8.54E+10	89.49%

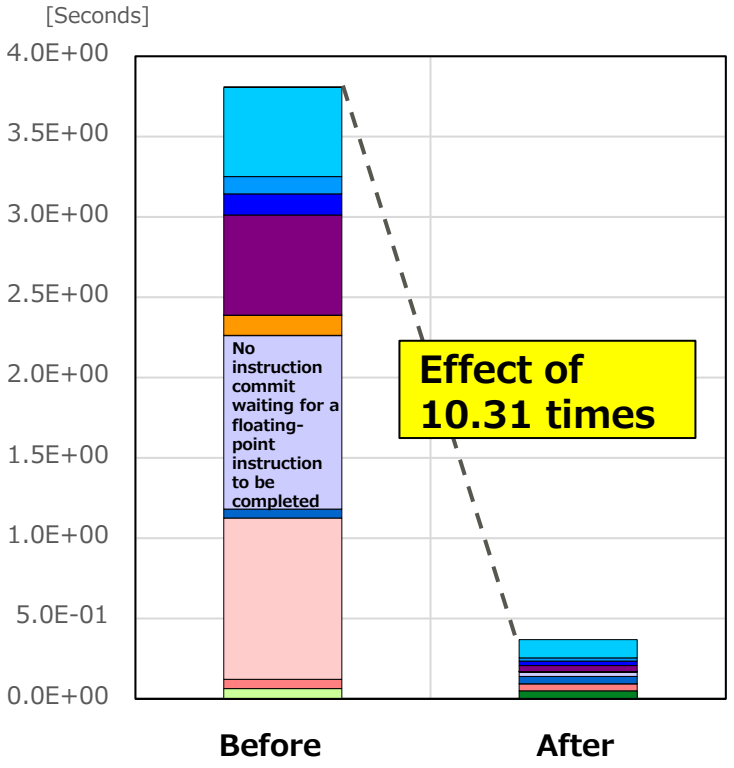
C/C++ Effect of Loop Unswitching (After Improvement)

FUJITSU

Specify loop unswitching for if statements to eliminate branching and facilitate SIMDization and software pipelining. The result is significant improvement of the "No instruction commit waiting for a floating-point instruction to be completed" event.

Source After Improvement (Optimization Control Line Tuning)

```
89              for( j=0; j<iter; j++ )
90              {
91                #pragma omp for nowait
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 1.12, ITR: 96, MVE: 3, POL: S)
          <<<    UNSWITCHING
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
92   p    2v       for(i=0; i<n1; i++)
93   p    2v       {
94             #pragma statement unswitching
95   p    2v         if (n1 >= q)
96   p    2v         {
97   p    2v           a[i] = c0+b[i]*(c0+b[i]*(c0+b[i]*(c0+b[i]*c0)));
98   p    2v         }
99
100            #pragma statement unswitching
101  p    2v         if(n1 > r)
102  p    2v         {
103  p    2v           a[i] = c0*b[i]/(c0*b[i]/(c0*b[i]/(c0*b[i]/c0)));
104  p    2v         }
105
106            #pragma statement unswitching
107  p    2v         if(n1 < s)
108  p    2v         {
109  p    2v           a[i] = c0+b[i]/(c0+b[i]/(c0+b[i]/(c0+b[i]/c0)));
110  p    2v         }
111  p    2v       }
112             }
```



Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
Before	8.54E+10	89.49%
After	1.68E+10	71.58%

FUJITSU

Microarchitecture-Dependent Bottlenecks

- Avoiding the Scatter Store Instruction
- Facilitating Gathering by the Gather Load Instruction
- Avoiding Excessive SFI
- Using the Multiple Structures Instruction
- Adjusting the Hardware Prefetch Distance
- SVE Vector Register Size (SIMD Width)
- Using the Half-Precision Real Type

FUJITSU

Avoiding the Scatter Store Instruction

- Avoiding the Scatter Store Instruction (Before Improvement)
- Avoiding the Scatter Store Instruction (Source Tuning)

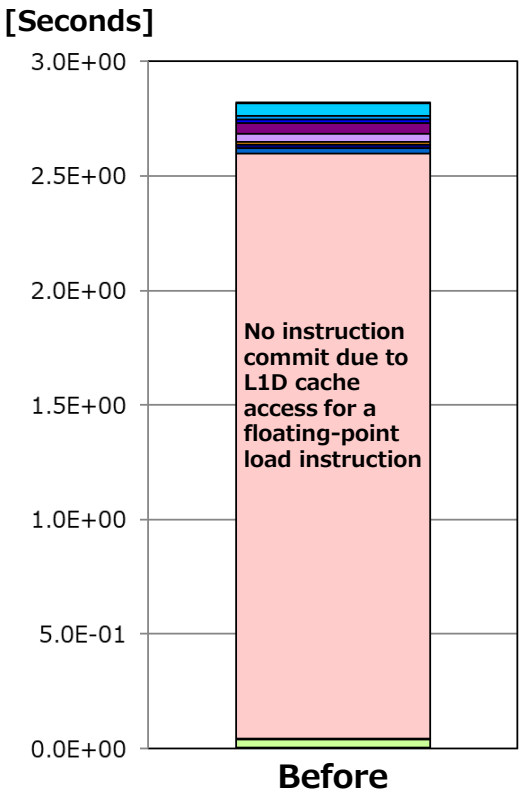
Fortran Avoiding the Scatter Store Instruction (Before Improvement)

FUJITSU

Performance degrades when the number of Scatter Store (non-sequential store) instructions is large. Consequently, the "No instruction commit due to L1D cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
42                !$omp parallel
43    1             DO K=1, ITER
44    1            !$omp do
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b
                <<< Loop-information  End >>>
45    2  p          DO J=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.60, ITR: 80,
                                                MVE: 3, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b
                <<< Loop-information  End >>>
46    3  p  2v          DO I=1,M
47    3  p  2v            a(J,I) = b(I,J)
48    3  p  2v          ENDDO
49    2  p            ENDDO
50    1             !$omp end do nowait
51    1             ENDDO
52                !$omp end parallel
```



High L1D miss rates

More Scatter Store instructions than other instructions

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
Before	0.00	1.62E+09	1.50E+09	0.92	99.98%	0.02%	0.00%

	Store instruction				
	SIMD				
	Single vector contiguous store instruction	Multiple vector contiguous structure store instruction	Non-contiguous scatter store instruction	Floating-point register spill instruction	Predicate register spill instruction
Before	1.60E+01	0.00E+00	1.30E+08	1.07E+04	5.49E+03

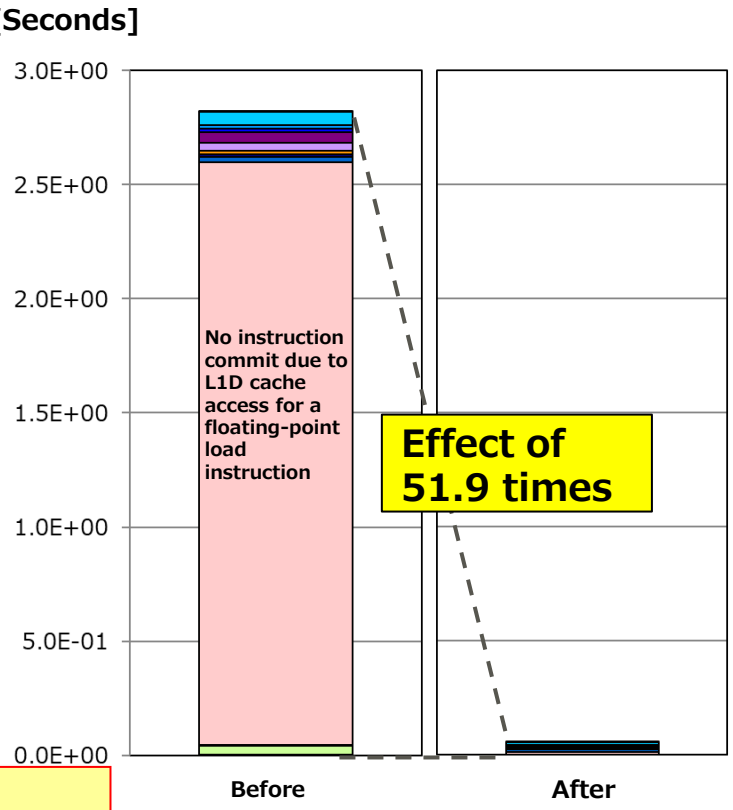
Fortran Avoiding the Scatter Store Instruction (Source Tuning)

FUJITSU

Change the loop index to prevent the occurrence of Scatter Store instructions. The result is significant improvement in L1D misses.

Source After Improvement

```
42                !$omp parallel
43    1             DO K=1, ITER
44    1           !$omp do
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a
                <<< Loop-information  End >>>
45    2  p           DO J=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.87, ITR: 256,
                                                MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a
                <<< Loop-information  End >>>
46    3  p  4v          DO I=1,M
47    3  p  4v            a(I,J) = b(J,I)
48    3  p  4v          ENDDO
49    2  p           ENDDO
50    1           !$omp end do nowait
51    1             ENDDO
52                !$omp end parallel
```



L1D misses significantly improved

Number of Scatter Store instructions successfully reduced

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	hardware prefetch rate (%) (/L1D miss)	software prefetch rate (%) (/L1D miss)
Before	0.00	1.62E+09	1.50E+09	0.92	99.98%	0.02%	0.00%
After	0.00	3.64E+08	3.03E+06	0.01	99.97%	0.02%	0.01%

Store instruction				
SIMD				
Single vector contiguous store instruction	Multiple vector contiguous structure store instruction	Non-contiguous scatter store instruction	Floating-point register spill instruction	Predicate register spill instruction
1.60E+01	0.00E+00	1.30E+08	1.07E+04	5.49E+03
1.30E+08	0.00E+00	0.00E+00	6.40E+01	2.38E+02

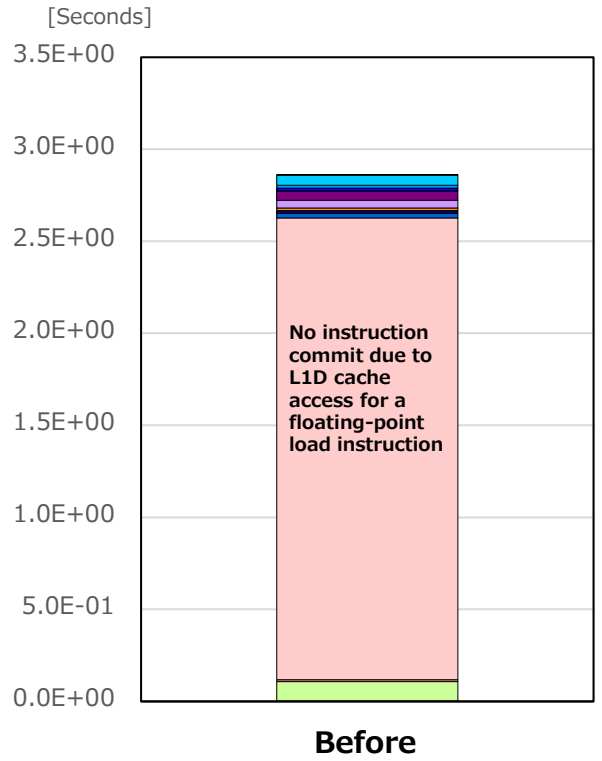
C/C++ Avoiding the Scatter Store Instruction (Before Improvement)

FUJITSU

Performance degrades when the number of Scatter Store (non-sequential store) instructions is large. Consequently, the "No instruction commit due to L1D cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
44          #pragma omp parallel
45            {
46              for(k=0; k<iter; k++)
47              {
48            #pragma omp for nowait
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      b
            <<< Loop-information  End >>>
49   p            for(j=0; j<m; j++)
50   p            {
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.40, ITR: 80,
                                MVE: 3, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      b
            <<< Loop-information  End >>>
51   p    2v          for(i=0;i<m;i++)
52   p    2v          {
53   p    2v            a[i][j] = b[j][i];
54   p    2v          }
55   p              }
56                }
57              }
```



High L1D miss rates

More Scatter Store instructions than other instructions

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
Before	0.00	1.58E+09	1.50E+09	0.95	100.00%	0.00%	0.00%

Single vector contiguous store instruction	Multiple vector contiguous structure store instruction	Non-contiguous scatter store instruction	Floating-point register spill instruction	Predicate register spill instruction
0.00E+00	0.00E+00	1.30E+08	1.17E+04	5.88E+03

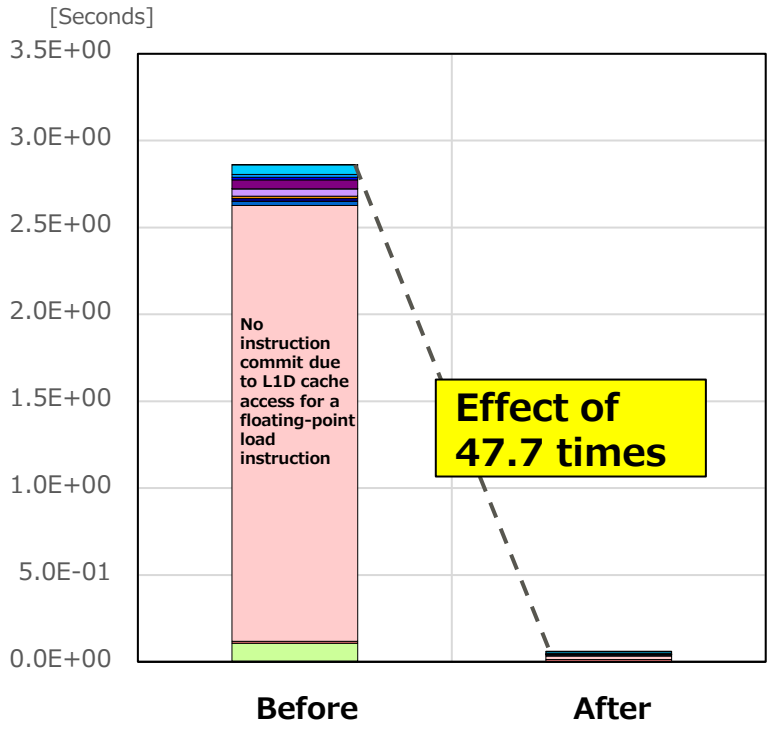
C/C++ Avoiding the Scatter Store Instruction (Source Tuning)

FUJITSU

Change the loop index to prevent the occurrence of Scatter Store instructions. The result is significant improvement in L1D misses.

Source After Improvement

```
44          #pragma omp parallel
45           {
46             for(k=0; k<iter; k++)
47             {
48          #pragma omp for nowait
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      a
            <<< Loop-information  End >>>
49   p          for(j=0; j<m; j++)
50   p          {
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 1.28, ITR: 96,
                                MVE: 2, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      a
            <<< Loop-information  End >>>
51   p    4v       for(i=0;i<m;i++)
52   p    4v       {
53   p    4v         a[j][i] = b[i][j];
54   p    4v       }
55   p           }
56             }
57           }
```



L1D misses significantly improved

Number of Scatter Store instructions successfully reduced

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	hardware prefetch rate (%) (/L1D miss)	software prefetch rate (%) (/L1D miss)	Single vector contiguous store instruction	Multiple vector contiguous structure store instruction	Non-contiguous scatter store instruction	Floating-point register spill instruction	Predicate register spill instruction
Before	0.00	1.58E+09	1.50E+09	0.95	100.00%	0.00%	0.00%	0.00E+00	0.00E+00	1.30E+08	1.17E+04	5.88E+03
After	0.00	4.13E+08	3.67E+06	0.01	99.35%	0.67%	-0.01%	1.30E+08	0.00E+00	0.00E+00	3.20E+01	1.70E+01

FUJITSU

Facilitating Gathering by the Gather Load Instruction

- Gathering Function of the Gather Instruction
- Facilitating Gathering by the Gather Load Instruction (Before Improvement)
- Facilitating Gathering by the Gather Load Instruction (Source Tuning)

Gathering Function of the Gather Instruction

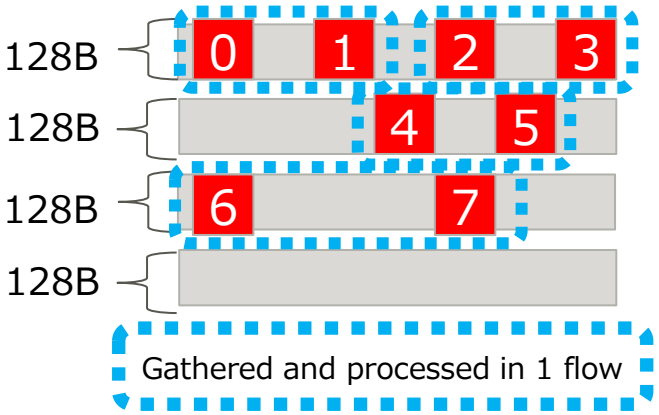
FUJITSU

What is the gathering function of the Gather instruction?

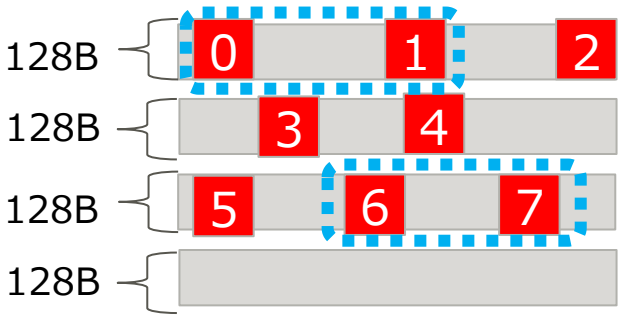
If two elements issued simultaneously from one FP are adjacent to each other, the Gather instruction can process them at a high speed. If the addresses of the two adjacent elements match each other within 128 bytes, the instruction can speed up processing by gathering the elements and processing them in one flow.

- If two adjacent elements belong to the same 128-byte block, they are gathered and processed in one flow. In the following address pattern examples, ⬚ indicates the gathered parts, and the two elements are processed in one L1D$ pipeline flow.

◆ Address pattern example 1



◆ Address pattern example 2



Pay attention when implementing the starting addresses of arrays to make full use of the gathering function of the Gather instruction.

Fortran Facilitating Gathering by the Gather Load Instruction (Before Improvement)

FUJITSU

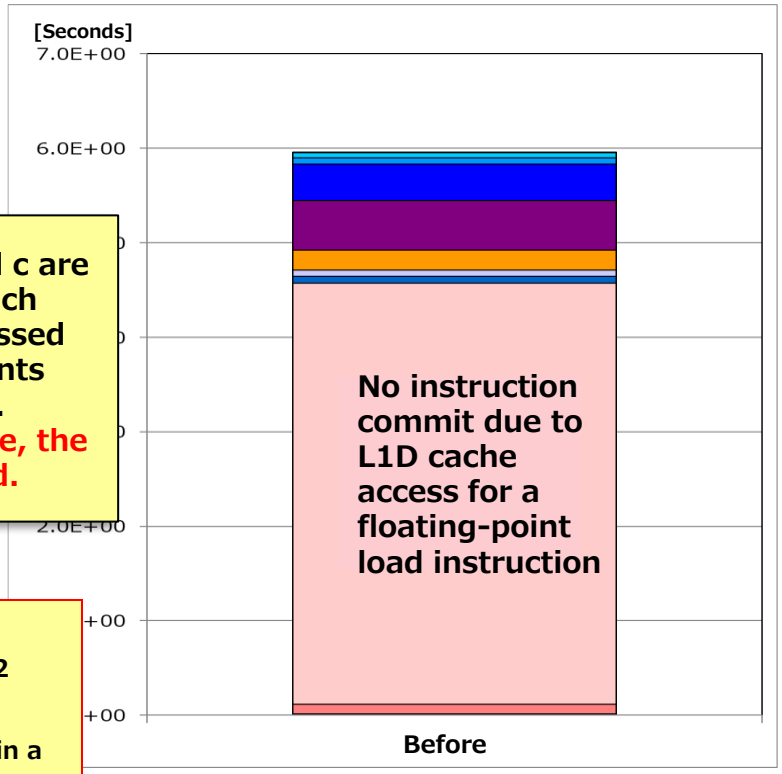
The Gather Load and Scatter Store instructions occur due to stride access, and the "No instruction commit due to L1D cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
30                real(kind=8),dimension(n,m) :: a
31                real(kind=8),dimension(n,m) :: b,c
32
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC
                <<< Loop-information  End >>>
33    1     v       do i = 1, m
34    1     v         a( 1,i) = b( 1,i) + c( 1,i)
35    1     v         a( 2,i) = b( 2,i) + c( 2,i)
36    1     v         a( 3,i) = b( 3,i) + c( 3,i)
37    1     v         a( 4,i) = b( 4,i) + c( 4,i)
38    1     v         a( 5,i) = b( 5,i) + c( 5,i)
39    1     v         a( 6,i) = b( 6,i) + c( 6,i)
40    1     v         a( 7,i) = b( 7,i) + c( 7,i)
41    1     v         a( 8,i) = b( 8,i) + c( 8,i)
42    1     v         a( 9,i) = b( 9,i) + c( 9,i)
43    1     v         a(10,i) = b(10,i) + c(1
44    1     v         a(11,i) = b(11,i) + c(1
45    1     v         a(12,i) = b(12,i) + c(1
46    1     v         a(13,i) = b(13,i) + c(1
47    1     v         a(14,i) = b(14,i) + c(1
48    1     v         a(15,i) = b(15,i) + c(1
49    1     v         a(16,i) = b(16,i) + c(1
50    1     v       end do
```

n = 16

Although arrays a, b, and c are sequentially accessed, each array (a(1,i) etc.) is accessed with a stride of 16 elements (128 bytes) per iteration.
-> Since access is discrete, the Gather instruction is used.



The non-contiguous Gather Load instruction is used, but 128 or more bytes are between the addresses of 2 adjacent elements. Therefore, the gathering function of the Gather instruction is not working, resulting in a high L1 busy rate.

Instruction

	Load-store instruction	
	Load instruction	Store instruction
	Non-contiguous gather load instruction	Non-contiguous scatter store instruction
Before	9.60E+08	4.80E+08

Busy	L1 busy rate (%)	L2 busy rate (%)	Memory busy rate (%)
Before	76.29%	2.15%	0.00%

Extra	Gather instruction rate (%)		
	0 flow rate (%)	1 flow rate (%)	2 flow rate (%)
Before	0.00%	0.00%	100.00%

Fortran Facilitating Gathering by the Gather Load Instruction (Source Tuning)

FUJITSU

Arrays were split so that there would be less than 128 bytes between the addresses of two adjacent elements. The gathering function of the Gather function now works. The result is improvement of the "No instruction commit due to L1D cache access for a floating-point load instruction" event.

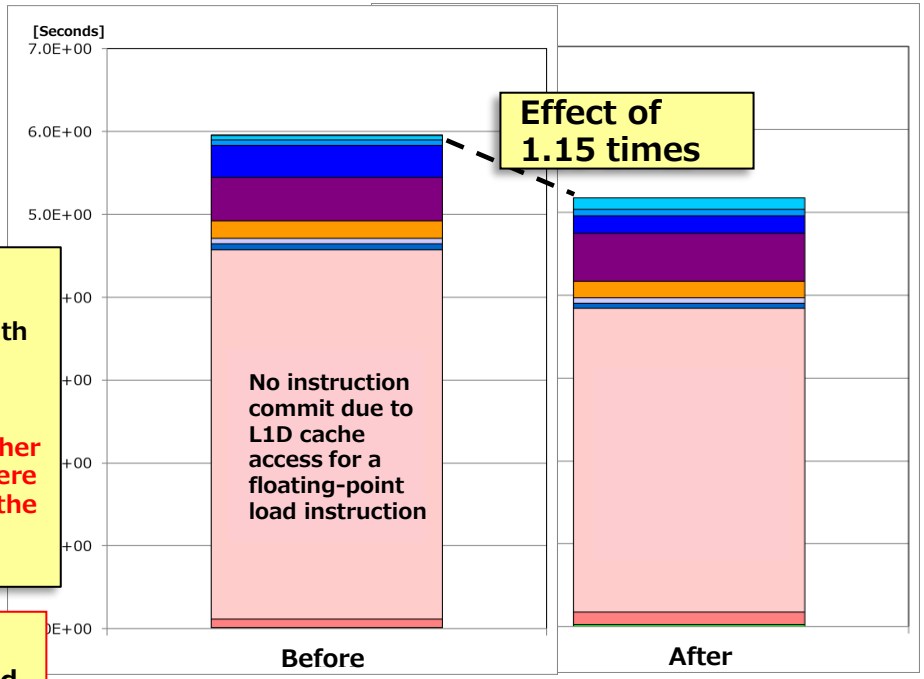
Source After Improvement (Source Tuning)

```
30              real(kind=8),dimension(n,m) :: a1,a2,a3,a4
31              real(kind=8),dimension(n,m) :: b1,b2,b3,b4,c1,c2,c3,c4
32
                <<< Loop-informati
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELI
                <<< Loop-information  En
33    1     v       do i = 1, m
34    1     v         a1( 1,i) = b1( 1,i) +
35    1     v         a1( 2,i) = b1( 2,i) +
36    1     v         a1( 3,i) = b1( 3,i) +
37    1     v         a1( 4,i) = b1( 4,i) +
38    1     v         a2( 1,i) = b2( 1,i) +
39    1     v         a2( 2,i) = b2( 2,i) +
40    1     v         a2( 3,i) = b2( 3,i) +
41    1     v         a2( 4,i) = b2( 4,i) +
42    1     v         a3( 1,i) = b3( 1,i) + c3( 1,i)
43    1     v         a3( 2,i) = b3( 2,i
44    1     v         a3( 3,i) = b3( 3,i
45    1     v         a3( 4,i) = b3( 4,i
46    1     v         a4( 1,i) = b4( 1,i
47    1     v         a4( 2,i) = b4( 2,i
48    1     v         a4( 3,i) = b4( 3,i
49    1     v         a4( 4,i) = b4( 4,i) + c4( 4,i)
50    1     v       end do
```

n = 4

Although arrays a, b, and c are sequentially accessed, each array (a1(1,i) and so on) is accessed with a stride of 4 elements (32 bytes) per iteration.
->
The gathering function of the Gather instruction is working because there are less than 128 bytes between the addresses of pairs of adjacent elements.

The gathering function of the Gather instruction is working, and 1 L1D$ pipeline flow is now processing 2 adjacent elements.



Busy

	L1 busy rate (%)	L2 busy rate (%)	Memory busy rate (%)
Before	76.29%	2.15%	0.00%
After	66.21%	2.84%	0.00%

Instruction

	Load-store instruction	
	Load instruction	Store instruction
	Non-contiguous gather load instruction	Non-contiguous scatter store instruction
Before	9.60E+08	4.80E+08
After	9.60E+08	4.80E+08

Extra

	Gather instruction rate (%)		
	0 flow rate (%)	1 flow rate (%)	2 flow rate (%)
Before	0.00%	0.00%	100.00%
After	0.00%	75.00%	25.00%

Facilitating Gathering by the Gather Load Instruction (Before Improvement)

FUJITSU

The Gather Load and Scatter Store instructions occur due to stride access, and the "No instruction commit due to L1D cache access for a floating-point load instruction" event occurs many times.

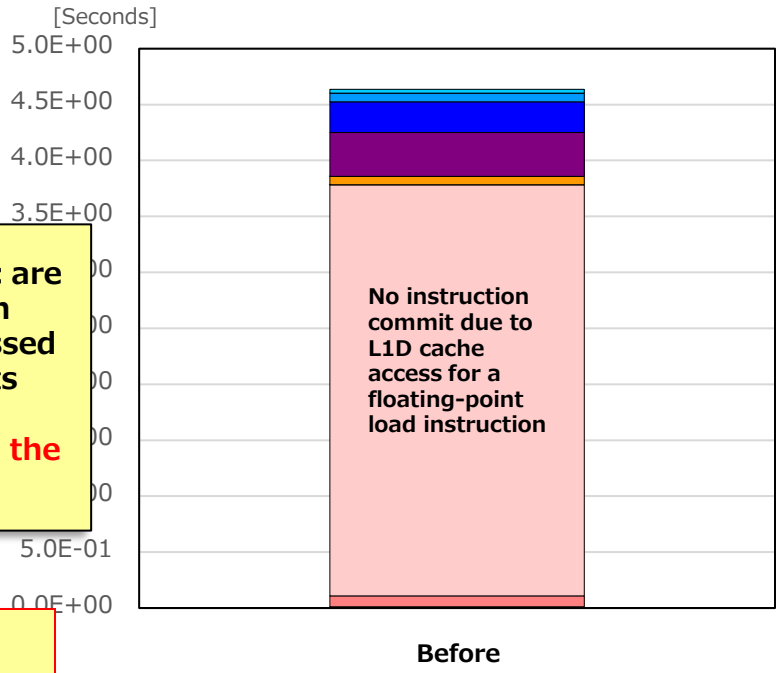
Source Before Improvement

```
50        void sub(int n, int m,double (* restrict a)[n],
            double (* restrict b)[n],double (* restrict c)[n])
51        {
52          int i;
53
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2
          <<< Loop-information  End >>>
54      v   for(i=0; i<m; i++)
55      v   {
56      v     a[i][ 0] = b[i][ 0] + c[i][ 0];
57      v     a[i][ 1] = b[i][ 1] + c[i][ 1];
58      v     a[i][ 2] = b[i][ 2] + c[i][ 2];
59      v     a[i][ 3] = b[i][ 3] + c[i][ 3];
60      v     a[i][ 4] = b[i][ 4] + c[i][ 4];
61      v     a[i][ 5] = b[i][ 5] + c[i][ 5];
62      v     a[i][ 6] = b[i][ 6] + c[i][ 6]
63      v     a[i][ 7] = b[i][ 7] + c[i][ 7]
64      v     a[i][ 8] = b[i][ 8] + c[i][ 8]
65      v     a[i][ 9] = b[i][ 9] + c[i][ 9]
66      v     a[i][10] = b[i][10] + c[i][1
67      v     a[i][11] = b[i][11] + c[i][1
68      v   }
70          return;
71        }
```

n = 16

Although arrays a, b, and c are sequentially accessed, each array (a[i][1] etc.) is accessed with a stride of 16 elements (128 bytes) per iteration.
-> Since access is discrete, the Gather instruction is used.

The non-contiguous Gather Load instruction is used, but 128 or more bytes are between the addresses of 2 adjacent elements. Therefore, the gathering function of the Gather instruction is not working, resulting in a high L1 busy rate.



	L1 busy rate (%)	L2 busy rate (%)	Memory busy rate (%)
Before	77.75%	2.77%	0.00%

	Load-store instruction	
	Load instruction	Store instruction
	Non-contiguous gather load instruction	Non-contiguous scatter store instruction
Before	7.20E+08	3.60E+08

	Gather instruction rate (%)		
	0 flow rate (%)	1 flow rate (%)	2 flow rate (%)
Before	0.00%	0.00%	8.33%

C/C++ Facilitating Gathering by the Gather Load Instruction (Source Tuning)

FUJITSU

Arrays were split so that there would be less than 128 bytes between the addresses of two adjacent elements. The gathering function of the Gather function now works. The result is improvement of the "No instruction commit due to L1D cache access for a floating-point load instruction" event.

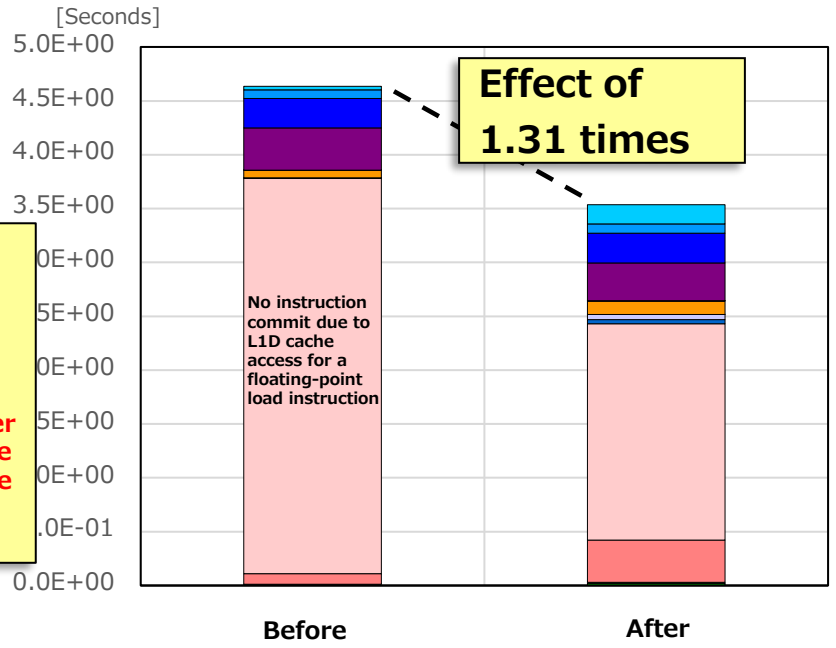
Source After Improvement (Source Tuning)

```
73        void sub(int n, int m,double (* restrict a1)[n],double (* restrict a2)[n],
74                 double (* restrict a3)[n],
75                 double (* restrict b1)[n],double (* restrict b2)[n],
76                 double (* restrict b3)[n],
77                 double (* restrict c1)[n],double (* restrict c2)[n],
78                 double (* restrict c3)[n])
79        {
80          int i;
81
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IP
          <<< Loop-information  End >>>
82        v   for(i=0; i<m; i++)
83        v   {
84        v     a1[i][ 0] = b1[i][ 0] + c1[i][ 0
85        v     a1[i][ 1] = b1[i][ 1] + c1[i][ 1
86        v     a1[i][ 2] = b1[i][ 2] + c1[i][ 2
87        v     a1[i][ 3] = b1[i][ 3] + c1[i][ 3
88        v     a2[i][ 0] = b2[i][ 0] + c2[i][ 0];
89        v     a2[i][ 1] = b2[i][ 1] + c2[i][ 1];
90        v     a2[i][ 2] = b2[i][ 2] + c2[i][ 2];
91        v     a2[i][ 3] = b2[i][ 3] + c2[i][ 3];
92        v     a3[i][ 0] = b3[i][ 0] + c3[i][ 0];
93        v     a3[i][ 1] = b3[i][ 1]
94        v     a3[i][ 2] = b3[i][ 2]
95        v     a3[i][ 3] = b3[i][ 3]
96        v   }
98          return;
99        }
```

n = 4

Although arrays a, b, and c are sequentially accessed, each array (a1[i][1] and so on) is accessed with a stride of 4 elements (32 bytes) per iteration.
->
The gathering function of the Gather instruction is working because there are less than 128 bytes between the addresses of pairs of adjacent elements.

The gathering function of the Gather instruction is working, and 1 L1D$ pipeline flow is now processing 2 adjacent elements.



Busy	L1 busy rate (%)	L2 busy rate (%)	Memory busy rate (%)
Before	77.75%	2.77%	0.00%
After	59.49%	1.07%	0.00%

Instruction

	Load-store instruction	
	Load instruction	Store instruction
	Non-contiguous gather load instruction	Non-contiguous scatter store instruction
Before	7.20E+08	3.60E+08
After	7.20E+08	3.60E+08

Extra

	Gather instruction rate (%)		
	0 flow rate (%)	1 flow rate (%)	2 flow rate (%)
Before	0.00%	0.00%	8.33%
After	0.00%	100.00%	0.00%

FUJITSU

Avoiding Excessive SFI

- What is Excessive SFI?
- Excessive SFI Occurrence Case 1
- Excessive SFI Occurrence Case 2
- Avoiding Excessive SFI (Before Improvement)
- Avoiding Excessive SFI (Source Tuning)

What is Excessive SFI?

FUJITSU

- What is SFI (Store Fetch Interlock)?
  - If the preceding store instruction and the following load instruction have different addresses, this control mechanism sets an interlock to prevent the load operation from passing the store operation.
  - Basically, only addresses for store are locked.

- Excessive SFI

  Excessive SFI is a phenomenon where the above control locks excessive addresses. This occurs in the following cases:
  - For masked SIMD, addresses whose mask determination value is 0 (do not store) are locked.
  - If the gathering function of GatherLoad works, all the elements on a cache line are subject to the SFI check for GatherLoad. In this case, SFI may be detected from addresses that are not actually for load, and the control may determine that they be locked.

Excessive SFI Occurrence Case 1

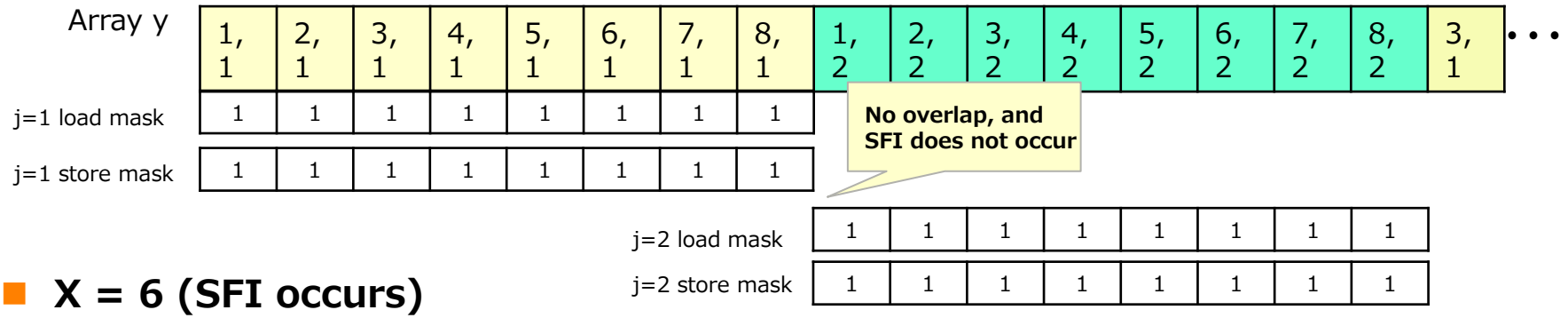
FUJITSU

In the right example, SFI does not occur when X=8, but addresses whose mask value is 0 are locked when X=6, resulting in excessive SFI.

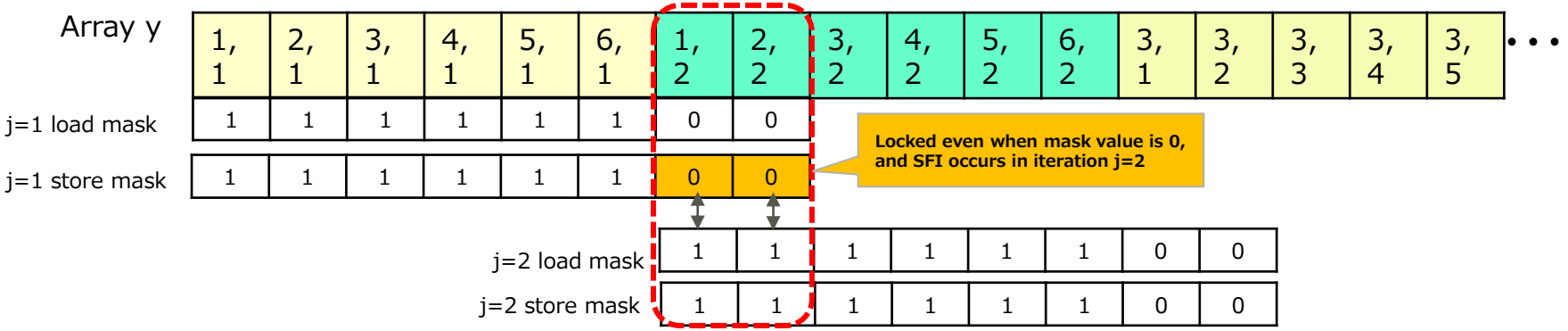
Source Code

```
real*8 y(X,n), x1(X,n)    <- X = 8 or 6
  Do k = 1, iter
    Do j = 1, n
      Do i = 1, X         <- X = 8 or 6
        y(i,j) = y(i,j) + x1(i,j)
      End Do
    End Do
  End Do
```

X = 8 (SFI does not occur)



X = 6 (SFI occurs)



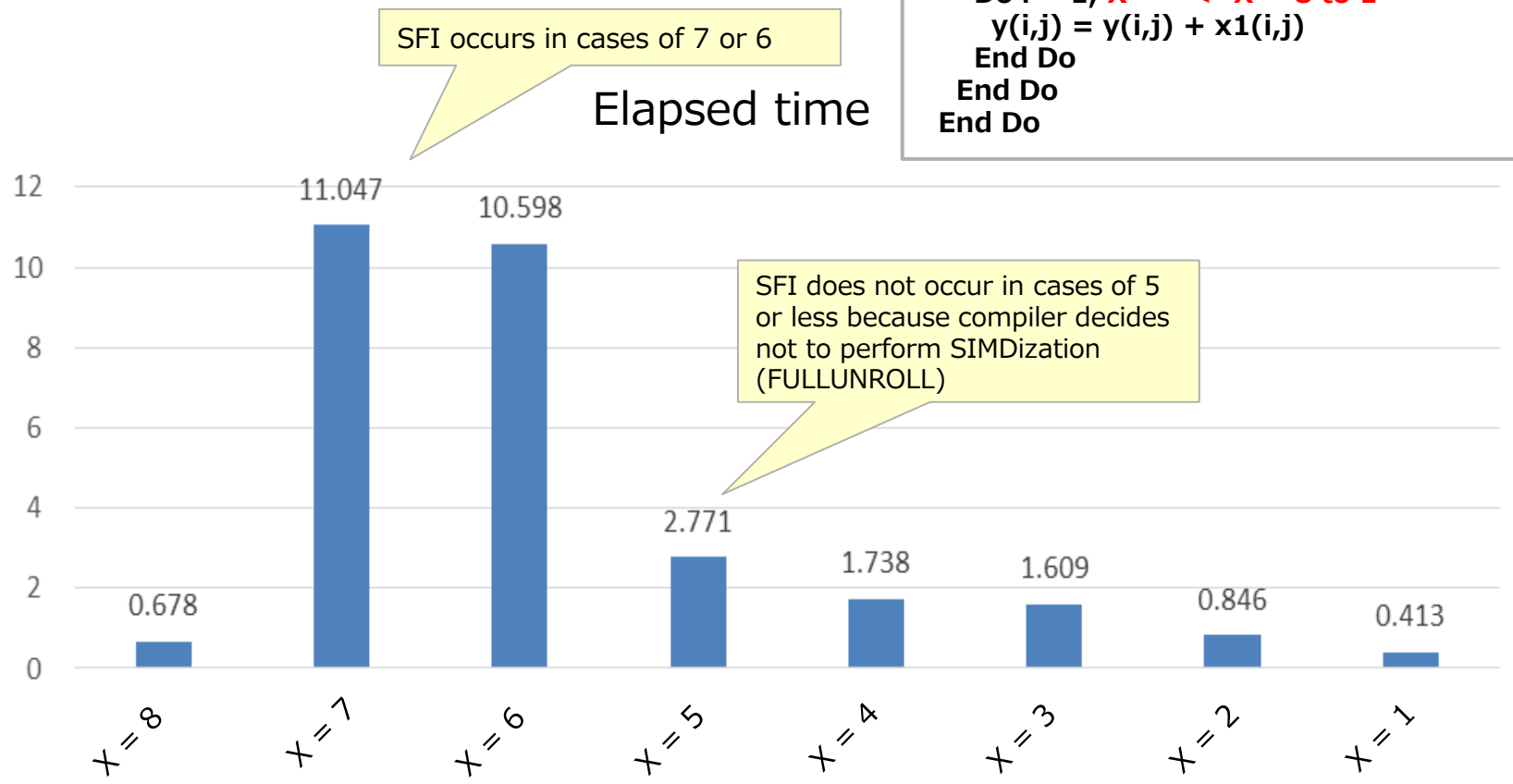
Excessive SFI Occurrence Case 1

FUJITSU

In the right example, SFI occurs when X=7 or 6. SFI does not occur when X=5 or less because SIMDization is not performed.

Source Code

```
real*8 y(X,n), x1(X,n)    <- X = 8 to 1
 Do k = 1, iter
  Do j = 1, n
   Do i = 1, X       <- X = 8 to 1
    y(i,j) = y(i,j) + x1(i,j)
   End Do
  End Do
 End Do
```



Excessive SFI Occurrence Case 2

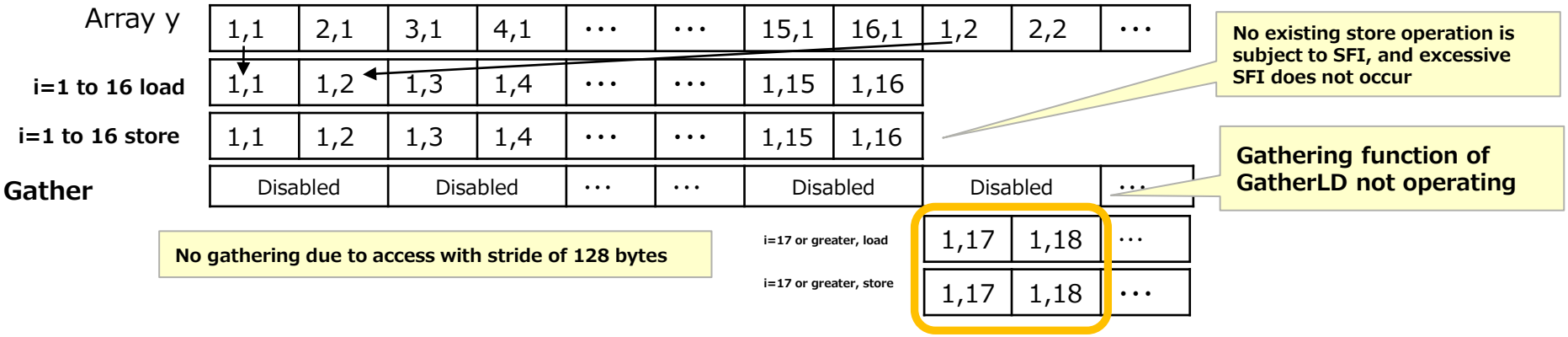
FUJITSU

In the right example, SFI does not occur when X=1, but excessive SFI occurs when X=16 because no existing store operation is subject to SFI.

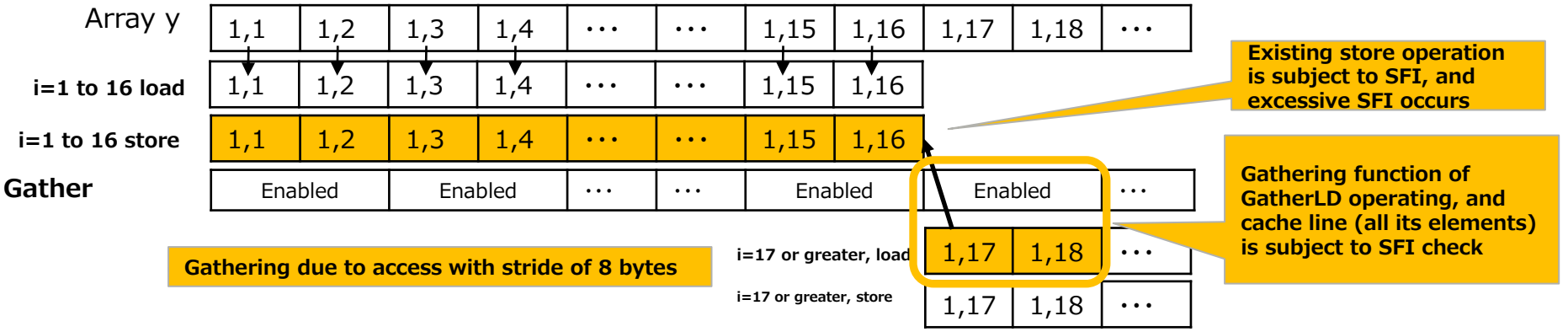
Source Code

```
integer n  <- 1 or 16
real*8 y(n, m), x1(n, m)
  Do k = 1, iter
    Do i = 1, m
       y(1, i) = y(1, i) + x1(1, i)
    End Do
  End Do
```

- **X = 1 (SFI does not occur)**



- **X = 16 (SFI occurs)**

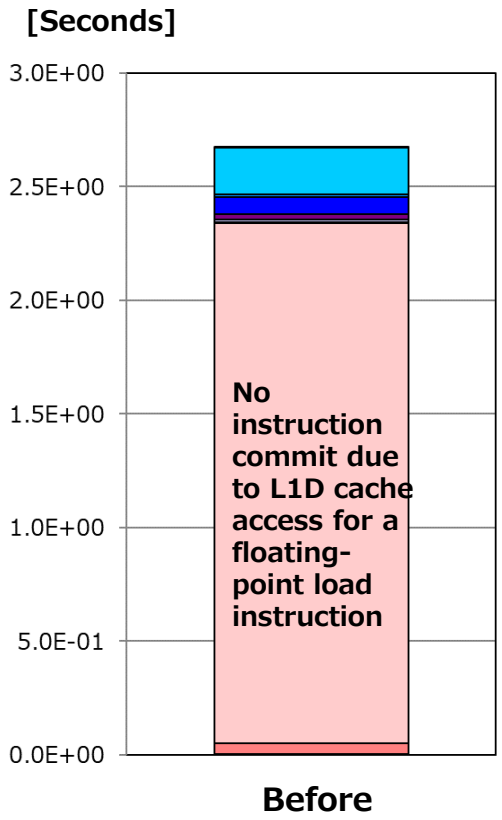


Fortran Avoiding Excessive SFI (Before Improvement)

The innermost loop applies to Case 1. Addresses with a mask value of 0 are locked, and excessive SFI occurs. Point: The case has a small number of loop iterations and no SWPL.

Source Before Improvement

```
39                  real(4) :: a(20,M), b(20,M)
40                  integer(4) :: M, ITER
41                  real(4),parameter :: c=0.5
 :
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 304,
                                                MVE: 7, POL: S)
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      a, b
                 <<< Loop-information  End >>>
46    2  p              DO J=1,M
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 16)
                 <<< Loop-information  End >>>
47    3  p   v            DO I=1,20
48    3  p   v              a(I,J) = a(I,J) + c * b(I,J)
49    3  p   v            ENDDO
50    2  p              ENDDO
```



Busy	SFI(Store Fetch Interlock) rate
Before	0.44

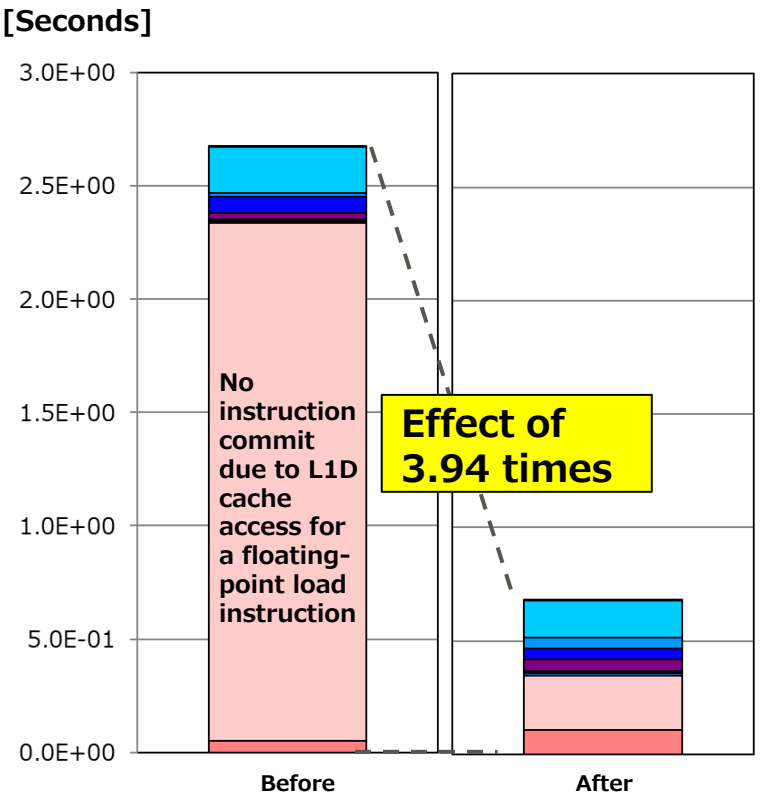
Avoiding Excessive SFI (Source Tuning)

FUJITSU

Excessive SFI was successfully avoided by changing the number of array elements through padding to a multiple of the SIMD length.

Source After Improvement

```
39                  real(4) :: a(32,M), b(32,M)
40                  integer(4) :: M, ITER
41                  real(4),parameter :: c=0.5
 :
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 304,
                                                MVE: 7, POL: S)
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      a, b
                 <<< Loop-information  End >>>
46    2  p              DO J=1,M
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 16)
                 <<< Loop-information  End >>>
47    3  p   v            DO I=1,20
48    3  p   v              a(I,J) = a(I,J) + c * b(I,J)
49    3  p   v            ENDDO
50    2  p              ENDDO
```



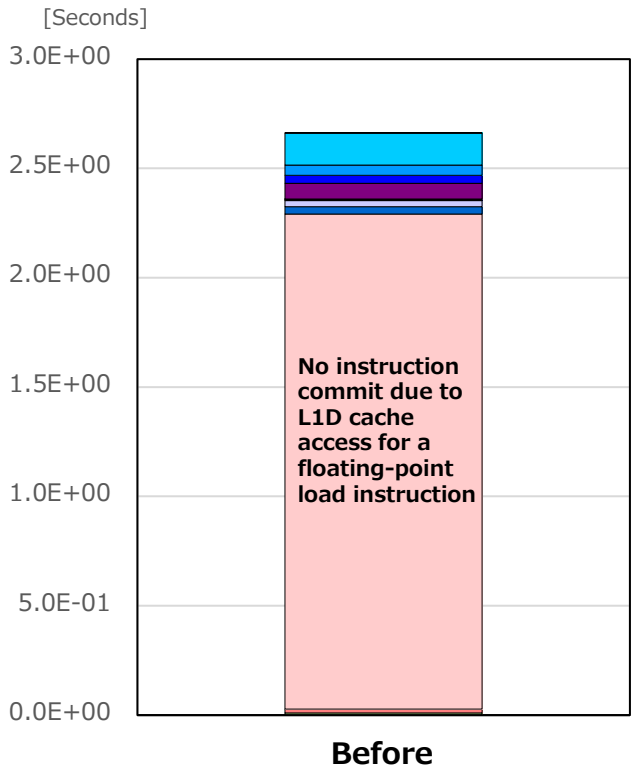
Busy	SFI(Store Fetch Interlock) rate
Before	0.43
After	0.01

C/C++ Avoiding Excessive SFI (Before Improvement)

The innermost loop applies to Case 1. Addresses with a mask value of 0 are locked, and excessive SFI occurs. Point: The case has a small number of loop iterations and no SWPL.

Source Before Improvement

```
42 void sfil1(float (* restrict a)[20], float (* restrict b)[20],
                    int m, int iter)
43          {
44            float  c=0.5;
45            int i,j,k;
46
47          #pragma omp parallel
48          {
           <<< Loop-information Start >>>
:
           <<< Loop-information  End >>>
49            for(k=0; k<iter; k++)
50            {
51          #pragma omp for nowait
           <<< Loop-information Start >>>
 :
           <<< Loop-information  End >>>
52  p           for(j=0; j<m; j++)
53  p           {
           <<< Loop-information Start >>>
 :
           <<< Loop-information  End >>>
54  p     v       for(i=0; i< 20; i++)
55  p     v       {
56  p     v         a[j][i] = a[j][i] + c * b[j][i];
57  p     v       }
58  p           }
59            }
60          }
62            return;
63          }
```



Busy	SFI(Store Fetch Interlock) rate
Before	0.44%

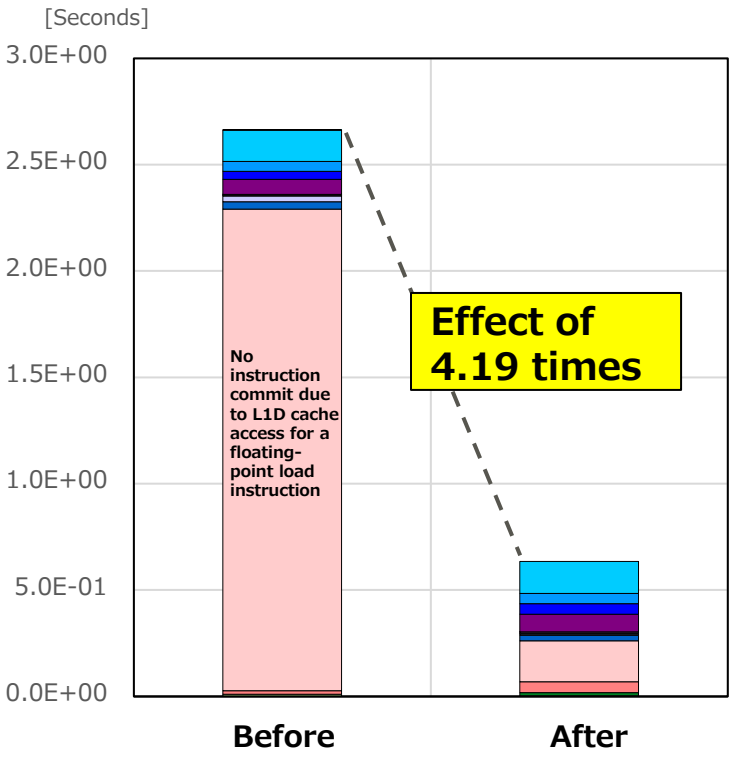
Avoiding Excessive SFI (Source Tuning)

FUJITSU

Excessive SFI was successfully avoided by changing the number of array elements through padding to a multiple of the SIMD length.

Source After Improvement

```
42 void sfil1(float (* restrict a)[32], float (* restrict b)[32],
                                  int m, int iter)
43          {
44            float  c=0.5;
45            int i,j,k;
46
47          #pragma omp parallel
48          {
           <<< Loop-information Start >>>
 :
           <<< Loop-information  End >>>
49            for(k=0; k<iter; k++)
50            {
51          #pragma omp for nowait
           <<< Loop-information Start >>>
 :
           <<< Loop-information  End >>>
52  p           for(j=0; j<m; j++)
53  p           {
           <<< Loop-information Start >>>
 :
           <<< Loop-information  End >>>
54  p   v         for(i=0; i< 20; i++)
55  p   v         {
56  p   v           a[j][i] = a[j][i] + c * b[j][i];
57  p   v         }
58  p           }
59            }
60          }
62            return;
63          }
```



Busy	SFI(Store Fetch Interlock) rate
Before	0.44%
After	0.01%

FUJITSU

Using the Multiple Structures Instruction

- Conditions for Applying the Multiple Structures Instruction
- Using the Multiple Structures Instruction (Before Improvement)
- Using the Multiple Structures Instruction (Source Tuning)

Conditions for Applying the Multiple Structures Instruction

FUJITSU

- **Supported by Fortran and C/C++ (Trad mode/Clang mode)**
- **Options can set on/off for Fortran and C/C++ (Trad mode).**

-Ksimd_use_multiple_structures |
-Ksimd_nouse_multiple_structures

Default is -Ksimd_use_multiple_structures, and can suppress with -Ksimd_nouse_multiple_structures

- **For an array of structures (AoS), the instruction applies when the innermost dimension consists of 2, 3, or 4 elements and all the elements are accessed. If the innermost dimension contains 5 or more elements, the instruction does not apply.**

- **Example of applicable case(Fortran) :**

```
real*8 y(n),x (4,n)

Do j = 1, iter
  Do i = 1, n
    y(i) = x(1,i) + x(2,i) + x(3,i) + x(4,i)
  End Do
End Do
```

- **Example of applicable case(C/C++) :**

```
double y[n],x[N][4];

for (j = 0; j < iter; j++) {
  for (i = 0; i < N; i++) {
    y(i) = x[i][0] + x[i][1] + x[i][2] + x[i][3];
  }
}
```

- **If higher performance through sequential load is expected from rewriting an array of structures (AoS) to a structure of arrays (SoA), we recommend doing so.**

Fortran Using the Multiple Structures Instruction (Before Improvement)

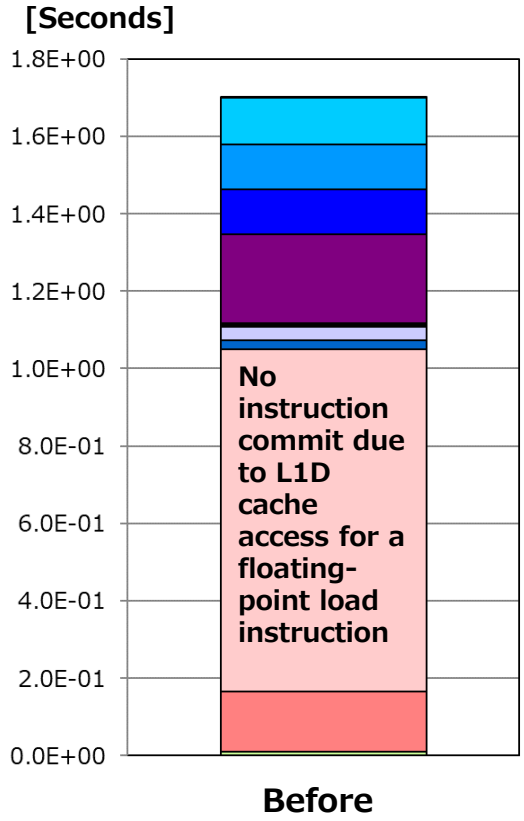
FUJITSU

An array of structures contains six elements and the Gather Load instruction is used (the Multiple Structures instruction is not applicable). Consequently, the "No instruction commit due to L1D cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.35, ITR: 96,
                                                MVE: 3, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a, b
                <<< Loop-information  End >>>
22    2  p  2v          do i=1,N
23    2  p  2v            b(i)=a(1,i) + a(2,i) + a(3,i) + &
                              a(4,i) + a(5,i) + a(6,i)
24    2  p  2v          enddo
```

The Multiple Structures instruction is not applicable because Array a contains 6 elements.



Fortran Using the Multiple Structures Instruction (Source Tuning)

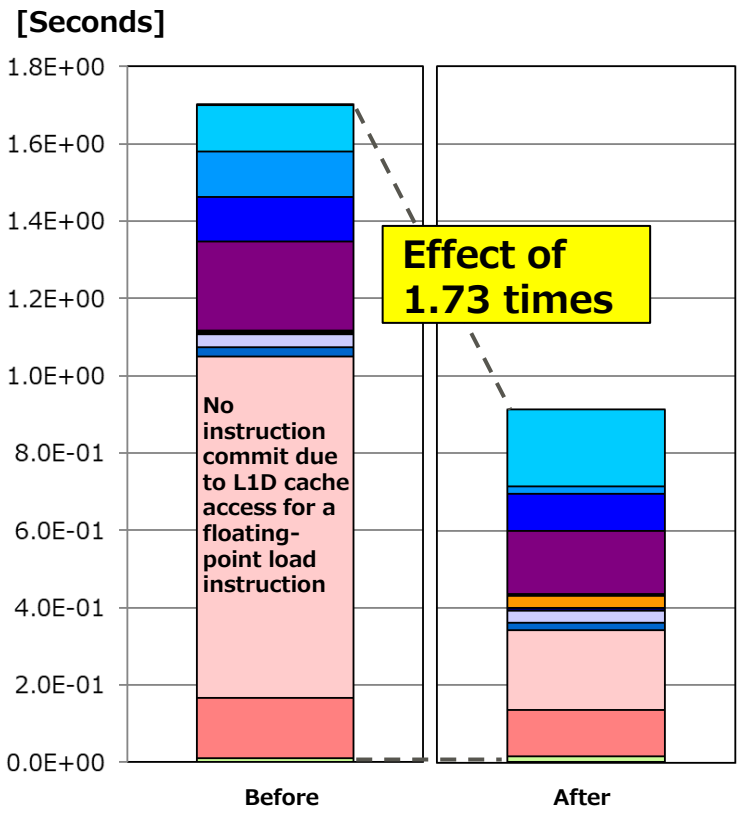
FUJITSU

Through division to form arrays containing three elements each, applying the Multiple Structures instruction reduces the "No instruction commit due to L1D cache access for a floating-point load instruction" event.

Source After Improvement (Source Tuning)

```
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 1.33, ITR: 56,
                                                MVE: 2, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, a, b
                <<< Loop-information  End >>>
22    2  p   v          do i=1,N
23    2  p   v            b(i) = a(1,i) + a(2,i) + a(3,i) + &
                                  c(1,i) + c(2,i) + c(3,i)
24    2  p   v          enddo
```

Applying the Multiple Structures instruction
Because Array a contained 6 elements, Array c was added so that each array contains 3 elements.



C/C++ Using the Multiple Structures Instruction (Before Improvement)

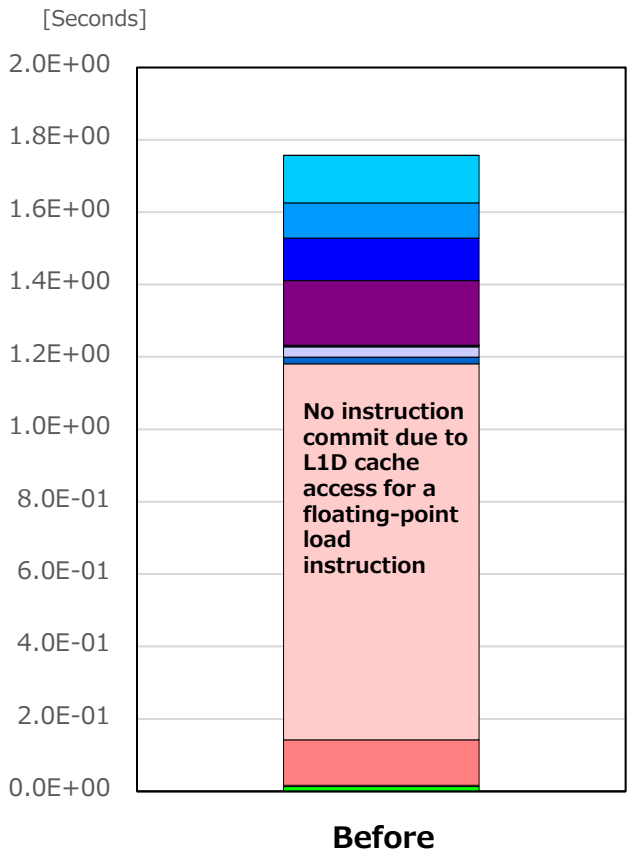
FUJITSU

An array of structures contains six elements and the Gather Load instruction is used (the Multiple Structures instruction is not applicable). Consequently, the "No instruction commit due to L1D cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
33          void MultiStructure(int n, int m, int iter)
34          {
35            int i,k;
36
37            #pragma omp parallel
38            {
            <<< Loop-information Start >>>
 :
            <<< Loop-information  End >>>
39              for(k=0; k< iter; k++)
40              {
41           #pragma omp for nowait
            <<< Loop-information Start >>>
 :
            <<< Loop-information  End >>>
42  p   2v        for(i=0; i< n; i++)
43  p   2v        {
44  p   2v          b[i]=a[i][0] + a[i][1] + a[i][2]
                        + a[i][3] + a[i][4] + a[i][5];
45  p   2v        }
46              }
47            }
48
49            return;
50          }
```

The Multiple Structures instruction is not applicable because Array a contains 6 elements.



C/C++ Using the Multiple Structures Instruction (Source Tuning)

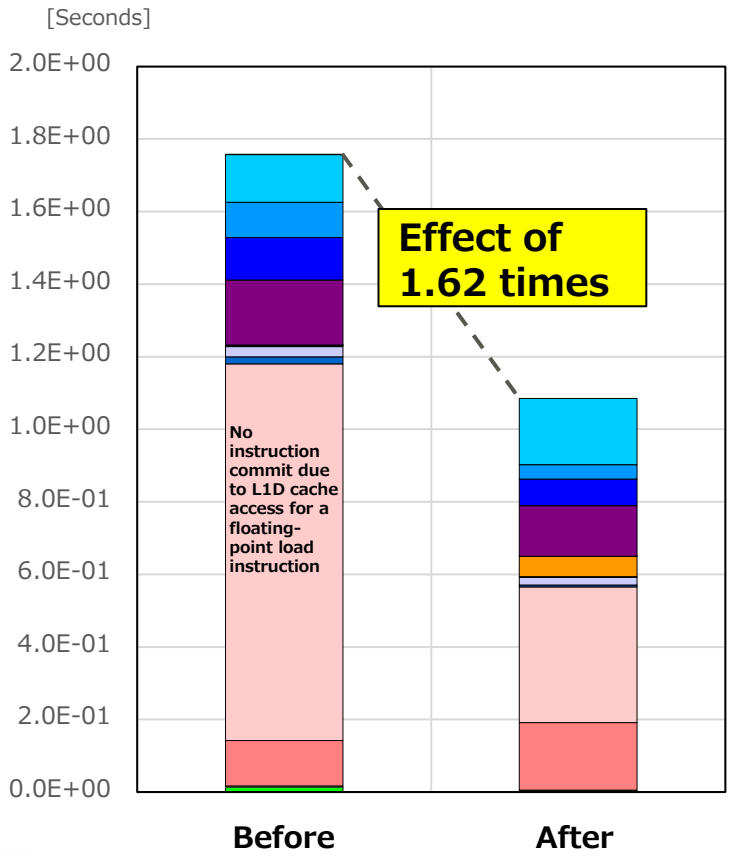
FUJITSU

Through division to form arrays containing three elements each, applying the Multiple Structures instruction reduces the "No instruction commit due to L1D cache access for a floating-point load instruction" event.

Source After Improvement (Source Tuning)

```
32          void MultiStructure(int n, int m, int iter)
33          {
34            int i,k;
35
36          #pragma omp parallel
37            {
           <<< Loop-information Start >>>
 :
           <<< Loop-information  End >>>
38            for(k=0; k<iter; k++)
39            {
40          #pragma omp for nowait
           <<< Loop-information Start >>>
 :
           <<< Loop-information  End >>>
41  p   v       for(i=0; i<N; i++)
42  p   v       {
43  p   v         b[i] = a[i][0] + a[i][1] + a[i][2]
                       + c[i][0] + c[i][1] + c[i][2];
44  p   v       }
45            }
46          }
47
48          return;
49        }
```

Applying the Multiple Structures instruction
Because Array a contained 6 elements, Array c was added so that each array contains 3 elements.



FUJITSU

Adjusting the Hardware Prefetch Distance

- Prefetch Distance
- Distance Setting Function for Hardware Prefetch
- Result of Hardware Prefetch Distance Adjustment (on L2)
- Result of Hardware Prefetch Distance Adjustment (on Memory)

Prefetch Distance

FUJITSU

- **Hardware and software prefetch distances**
  - **Hardware prefetch and software prefetch are performed on data located on lines ahead as shown below.**

Hardware Prefetch		Software Prefetch	
L1 Prefetch	L2 Prefetch	L1 Prefetch	L2 Prefetch
Up to 6 lines	**Up to 40 lines**	**Automatic**	**Automatic**

Hardware prefetch allows users to set the distance.

Software prefetch provides automatic distance adjustment.

- Prefetch distances are dependent on application access. Thrashing may occur when prefetching cache lines far ahead. We recommend prefetching nearby cache lines.

Distance Setting Function for Hardware Prefetch

FUJITSU

- Command for setting the hardware prefetch distance (hwpfctl)

Item	Description
Syntax	hwpfctl [--disableL1] [--disableL2] [--distL1 lines_l1] [--distL2 lines_l2] [--weakL1] [--weakL2] [--verbose] command {arguments ...} hwpfctl --default [--verbose] command {arguments ...} hwpfctl --reset [--verbose] hwpfctl --help
Explanation	The hwpfctl command changes the prefetch behavior (stream detect mode) of hardware mounted on the A64FX. Process affinity determines the CPU cores that are subject to the change.
Option	--disableL1 --disableL2 Disables hardware prefetch for the L1/L2 cache. If omitted, hardware prefetch is enabled. --distL1=lines_l1 --distL2=lines_l2 Specifies the prefetched cache line in the L1/L2 cache, as a number of cache lines counted from the cache line where a cache miss occurs. You can specify a value from 1 to 15 in lines_l1 to prefetch a line in the L1 cache, and a value from 1 to 60 in lines_l2 to prefetch a line in the L2 cache. However, the specified lines_l2 value is rounded up to the nearest multiple of 4, and the resulting value is written in the system register. If 0 is specified, the operation uses the default value of the CPU. If the option is omitted or the specified value is invalid, 0 is assumed specified. --weakL1 --weakL2 Sets "weak" as the priority of prefetch requests to the L1/L2 cache. If omitted, "strong" is the priority. --default Starts the command with the default settings. Options other than --verbose are ignored. --reset Initializes the system register values. Options other than --verbose are ignored. --verbose Outputs the values before and after a system register change. --help Displays usage instructions.

- Example of using the command for setting the hardware prefetch distance (hwpfctl)

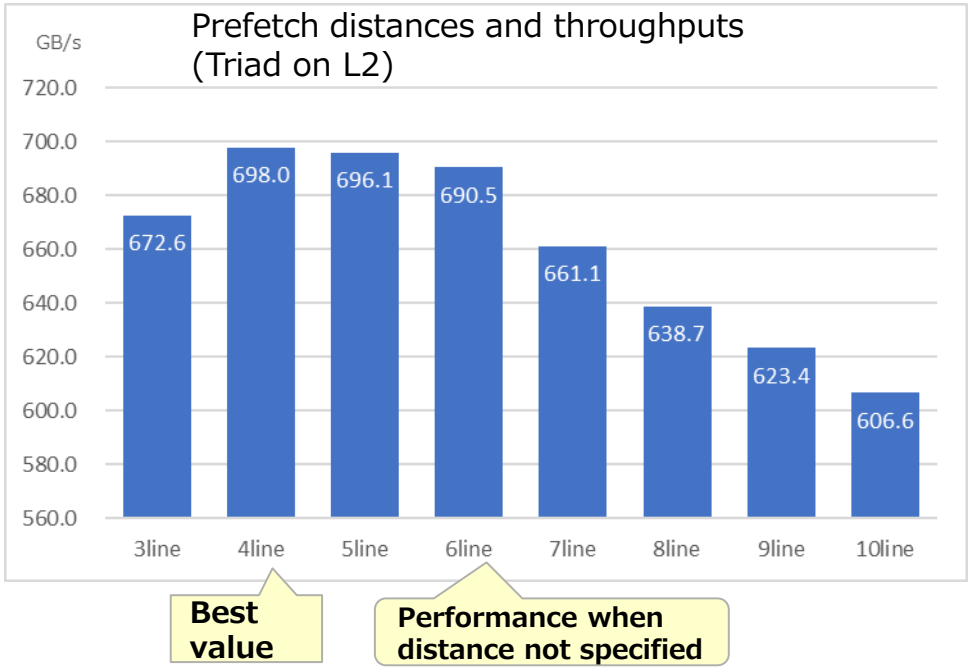
```
hwpfctl  --distL1=6  --distL2=40  a.out
```

Result of Hardware Prefetch Distance Adjustment (on L2)

FUJITSU

The following figure shows the result of hardware prefetch distance adjustment.

- **L1 prefetch distance evaluation using Triad (in L2 cache access)**



Triad

```
!$omp parallel
  Do j = 1, iter
!$omp do
     Do i = 1, n
       y(i)=x1(i) + c0 * x2(i)
     End Do
!$omp end do nowait
  End Do
!$omp end parallel
```

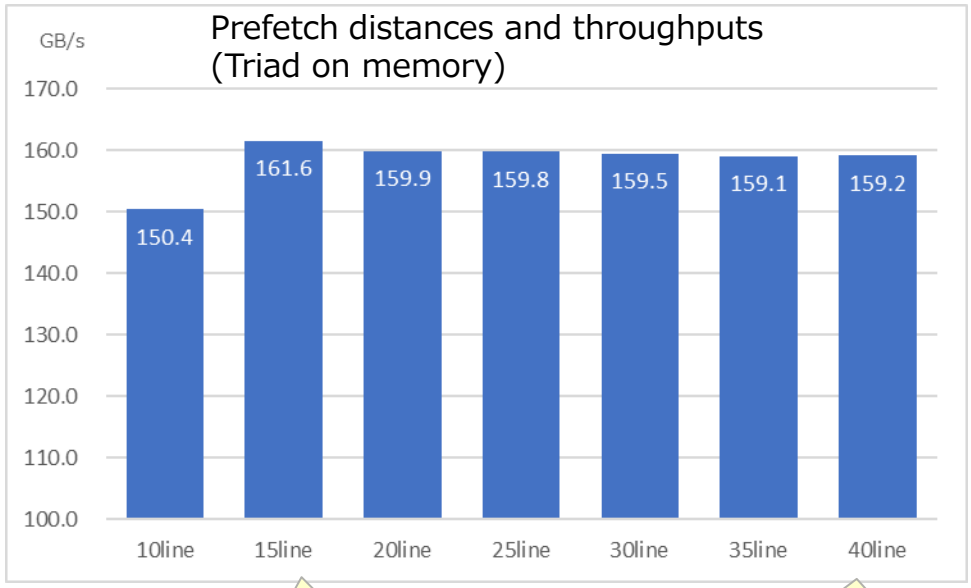
- **Setting command**

```
hwpfctl  –distL1=3~10   a.out
```

Result of Hardware Prefetch Distance Adjustment (on Memory)

FUJITSU

- **L2 prefetch distance evaluation using Triad (in memory access)**



Triad

```
!$omp parallel
  Do j = 1, iter
!$omp do
     Do i = 1, n
       y(i)=x1(i) + c0 * x2(i)
     End Do
!$omp end do nowait
  End Do
!$omp end parallel
```

* The values for these throughputs do not include reading of the cache lines for written data.

- **Setting command**

```
hwpfctl  –distL2=10~40  a.out
```

FUJITSU

SVE Vector Register Size (SIMD Width)

- SVE Vector Register Size (SIMD Width)
- Effect on Optimization With -Ksimd_reg_size=agnostic Specified (Caution)

SVE Vector Register Size (SIMD Width)

FUJITSU

- SIMD widths supported by the processor

  The ARM Scalable Vector Extension (SVE) allows the implementing system to freely decide the vector length (SIMD width) in units of 128 bits within a range of 128 bits to 2,048 bits.
  The A64FX is implemented with a vector length of 512 bits.

  - The A64FX supports the following vector lengths:
    - 512 bits
    - 256 bits
    - 128 bits

- Compiler option

  - -Ksimd_reg_size={ 128 | 256 | 512 | agnostic }

    The default is -Ksimd_reg_size=512.

    - simd_reg_size={ 128 | 256 | 512 }

      This option specifies the SVE vector register size in units of bits. The optimization by the compiler at the compile time assumes that the value specified by this option is the SVE vector register size. However, the generated executable program operates normally only in a CPU architecture implementing the SVE vector register of the size specified by the option.

    - simd_reg_size=agnostic

      This option specifies compilation with no specific size assumed for the SVE vector register to generate an executable program that decides the SVE vector register size at execution. This executable program can be executed independently from the size of the SVE vector register implemented in the CPU architecture. However, execution performance may be worse (degraded) than when the -Ksimd_reg_size={128|256|512} option is specified.

Effect on Optimization With -Ksimd_reg_size=agnostic Specified (Important Points)

FUJITSU

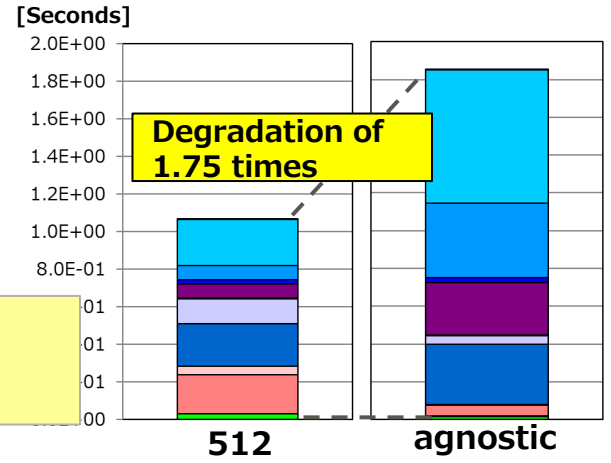
The optimization performed when -Ksimd_reg_size=agnostic is specified may not be equivalent to that with -Ksimd_reg_size=512 (default). In that case, execution performance may degrade.

Source With -Ksimd_reg_size=512 (default)

```
24    2  p          do k=1,L
25    3  p           do ii=1,N,blk
                     <<< Loop-information Start >>>
                     <<<  [OPTIMIZATION]
                     <<<    SOFTWARE PIPELINING(IPC: 0.31, ITR: 192,
                                                MVE: 2, POL: L)
                     <<<    PREFETCH(
                     <<<      b, a
                     <<< Loop-informa
26    4  p  8         do j=1,M
                     <<< Loop-information Start >>>
                     <<<  [OPTIMIZATION]
                     <<<    SIMD(VL: 8)
                     <<<    FULL UNROLLING
                     <<< Loop-information  End >>>
27    5  p  fv          do i=ii,ii+blk-1
28    5  p  fv            a(i,j,k)=a(i,j,k)+c*b(i,j,k)
29    5  p  fv          enddo
30    4  p  8         enddo
31    3  p           enddo
32    2  p          enddo
```

blk = 8

Since innermost loop corresponds to SIMD length, software pipelining applied in outer loop



Degradation of 1.75 times

Source When -Ksimd_reg_size=agnostic is Specified

```
24    2  p          do k=1,L
25    3  p           do ii=1,N,blk
26    4  p             do j=1,M
                     <<< Loop-information Start >>>
                     <<<  [OPTIMIZATION]
                     <<<    SIMD(VL: AGNOSTIC; VL: 2 in 128-bit)
                     <<<    PREFETCH(HARD) Expected by compiler :
                     <<<      b, a
                     <<< Loop-information  End >>>
27    5  p  2v          do i=ii,ii+blk-1
28    5  p  2v            a(i,j,k)=a(i,j,k)+c*b(i,j,k)
29    5  p  2v          enddo
30    4  p             enddo
31    3  p           enddo
32    2  p          enddo
```

Software pipelining not applied

Note

- **After compilation with -Ksimd_reg_size=agnostic specified, optimization that depends on the SIMD width (software pipelining in the above example) is not performed.**

Effect on Optimization With -Ksimd_reg_size=agnostic Specified (Important Points)

FUJITSU

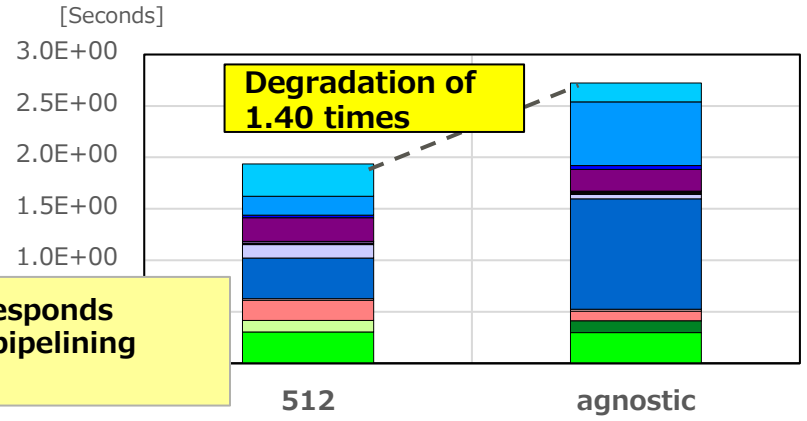
The optimization performed when -Ksimd_reg_size=agnostic is specified may not be equivalent to that with -Ksimd_reg_size=512 (default). In that case, execution performance may degrade.

Source With -Ksimd_reg_size=512 (default)

```
46            for(iter=1; iter<=itmax; iter++) {
48              #pragma omp for
49  p            for(k=0;k<L; k++) {
51  p             for(ii=0;ii<N;ii+=blk) {
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
 :
          <<< Loop-information  End >>>
53  p               for(j=0;j<M;j++) {
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                                     MVE: 7, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
55  p   2v            for(i=ii; i<ii+blk-1; i++) {
57  p   2v              a[k][j][i]=a[k][j][i]+c*b[k][j][i];
58  p   2v            }
59  p               }
60  p             }
61  p           }
62            }
```

blk = 8

Since innermost loop corresponds to SIMD length, software pipelining applied in outer loop



Source When -Ksimd_reg_size=agnostic is Specified

```
46            for(iter=1; iter<=itmax; iter++) {
48              #pragma omp for
49  p            for(k=0;k<L; k++) {
51  p             for(ii=0;ii<N;ii+=blk) {
53  p               for(j=0;j<M;j++) {
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: AGNOSTIC; VL: 2 in 128-bit)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
55  p   2v            for(i=ii; i<ii+blk-1; i++) {
57  p   2v              a[k][j][i]=a[k][j][i]+c*b[k][j][i];
58  p   2v            }
59  p               }
60  p             }
61  p             }
62            }
```

Software pipelining not applied

Note

- After compilation with -Ksimd_reg_size=agnostic specified, optimization that depends on the SIMD width (software pipelining in the above example) is not performed.

FUJITSU

Using the Half-Precision Real Type

- Using the Half-Precision Real Type (Before Improvement)
- Using the Half-Precision Real Type (Source Tuning)

Fortran Using the Half-Precision Real Type (Before Improvement)

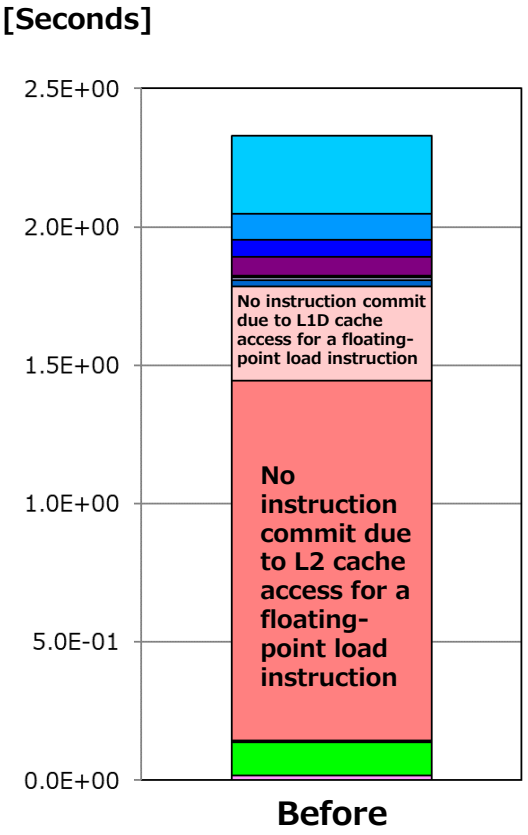
For double-precision real type data, the SIMD length is 8. However, by reducing data precision, you can increase the SIMD length to effectively use the bandwidth and functional unit.

Source Before Improvement

```
 2                integer,parameter::N=60000
 :
19                real(8)::x1(N),x2(N),y(N)
20                real(8),parameter::c=0.5
 :
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 144,
                                                 MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      x2, x1, y
                <<< Loop-information  End >>>
24    2  p  2v          do i=1,N
25    2  p  2v            y(i) = x1(i) + c * x2(i)
26    2  p  2v          enddo
```

SIMD length when SIMD width (vector length)=512 [bits]

Data Type	SIMD Length
Double-precision type	8
Single-precision type	16
Half-precision type	32
1-byte type	64



	GFLOPS	Floating-point operation peak ratio (%)
Before	51.52	6.71%

Fortran Using the Half-Precision Real Type (Source Tuning)

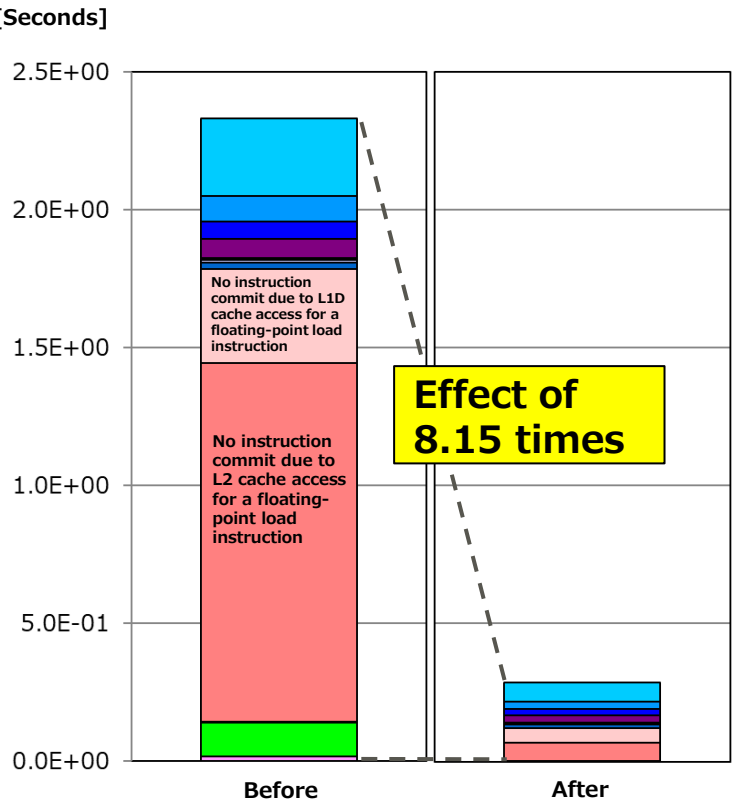
FUJITSU

You can extend the SIMD length to 32 by using the half-precision real type, which has fewer digits.

The reduction in data volume results in improvement of the "No instruction commit due to access for a floating-point load instruction" event.

Source After Improvement

```
2                integer,parameter::N=60000
:
19               real(2)::x1(N),x2(N),y(N)
20               real(2),parameter::c=0.5
:
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    SIMD(VL: 32)
              <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 576,
                                         MVE: 4, POL: S)
              <<<    PREFETCH(HARD) Expected by compiler :
              <<<      x2, x1, y
              <<< Loop-information  End >>>
24    2  p  2v         do i=1,N
25    2  p  2v           y(i) = x1(i) + c * x2(i)
26    2  p  2v         enddo
```



	GFLOPS	Floating-point operation peak ratio (%)
Before	51.52	6.71%
After	421.57	54.89%

Using the Half-Precision Real Type (Before Improvement)

FUJITSU

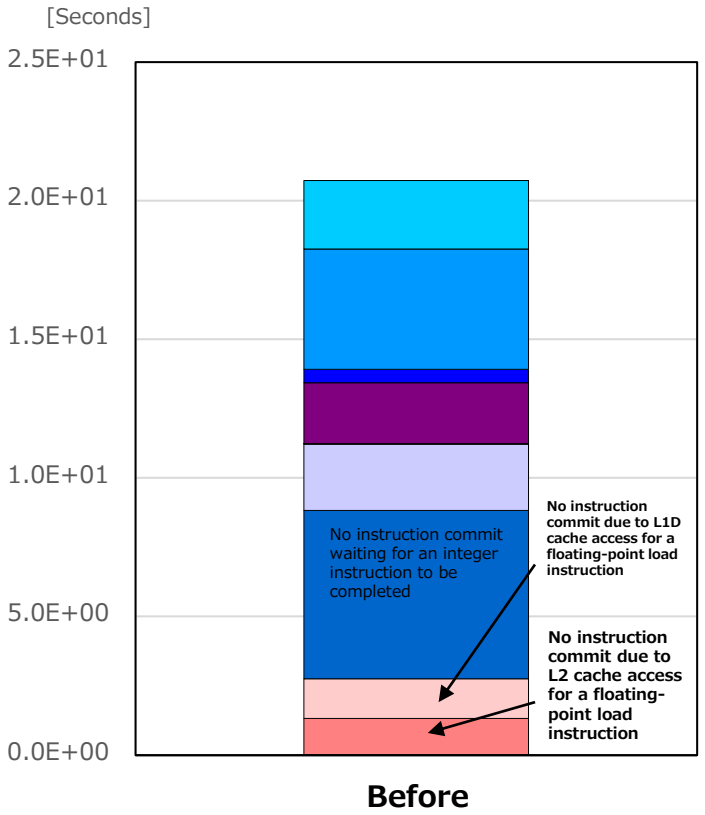
For double-precision real type data, the SIMD length is 8. However, by reducing data precision, you can increase the SIMD length to effectively use the bandwidth and functional unit.

Source Before Improvement

```
34          double const c=0.5;
35          int i,k;
36
37      4    for(k=0;k<itmax; k++)
38          {
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    SIMD(VL: AGNOSTIC;
                          VL: 2 in 128-bit Interleave: 1)
              <<< Loop-information End >>>
39         v     for(i=0;i<N; i++)
40            {
41              y[i] = x1[i] + c * x2[i];
42            }
43          }
```

N = 60000
itmax = 1000000

Array declaration
double x1[N], x2[N], y[N];



SIMD length when SIMD width (vector length)=512 [bits]

Data Type	SIMD Length
Double-precision type	8
Single-precision type	16
Half-precision type	32
1-byte type	64

Statistics	GFLOPS	Floating-point operation
Before	5.79	1.20E+11

C/C++ Clang Mode: Using the Half-Precision Real Type (Source Tuning)

FUJITSU

You can extend the SIMD length to 32 by using the half-precision real type, which has fewer digits.

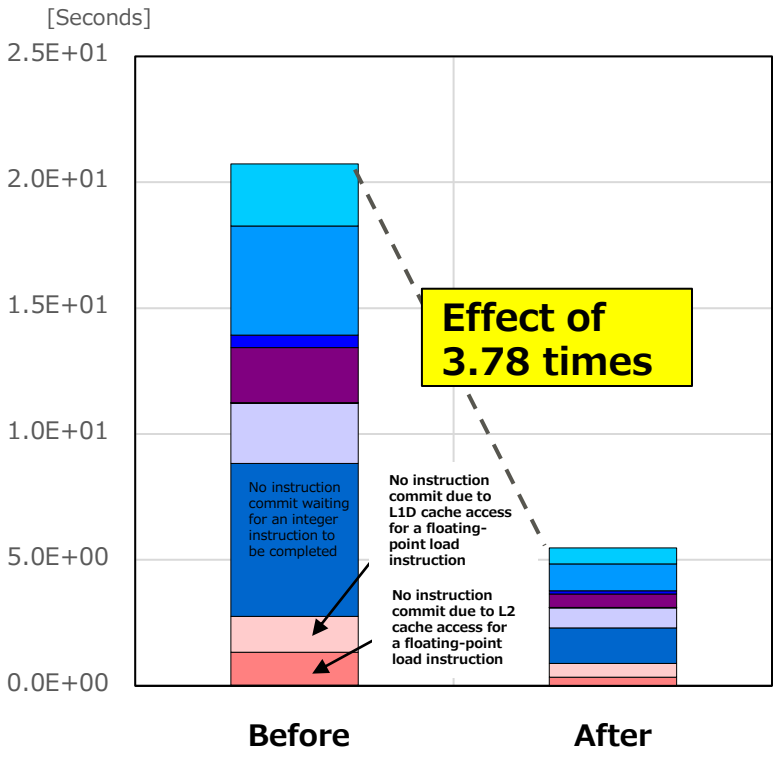
The reduction in data volume results in improvement of the "No instruction commit due to access for a floating-point load instruction" event.

Source After Improvement

```
34          _Float16 c = 0.5f16;
35          int i,k;
36
37      4   for(k=0;k<itmax; k++)
38          {
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    SIMD(VL: AGNOSTIC;
                       VL: 8 in 128-bit Interleave: 1)
              <<< Loop-information End >>>
39        v   for(i=0;i<N; i++)
40            {
41              y[i] = x1[i] + c * x2[i];
42            }
43          }
```

N = 60000
itmax = 1000000

Array declaration
double x1[N], x2[N], y[N];



Note

- **Half-Precision Real Type is not available in Trad Mode of C/C++.**

Statistics	GFLOPS	Floating-point operation
Before	5.79	1.20E+11
After	21.91	1.20E+11

FUJITSU

Thread Parallelization Tuning

- Improving the Thread Parallelization Ratio
- Improving Thread Parallelization Execution Efficiency
- Improving Execution Efficiency by Setting Large Pages

FUJITSU

Improving the Thread Parallelization Ratio

- What is the Thread Parallelization Ratio?
- Increasing the Thread Parallelization Ratio

What is the Thread Parallelization Ratio?

FUJITSU

The thread parallelization ratio is a proportion of the part that can be executed in parallel during one parallel execution.

At 1 parallel execution

At 2 parallel executions

The part that can be executed in parallel at 2 parallel executions is half of that at 1 parallel execution.

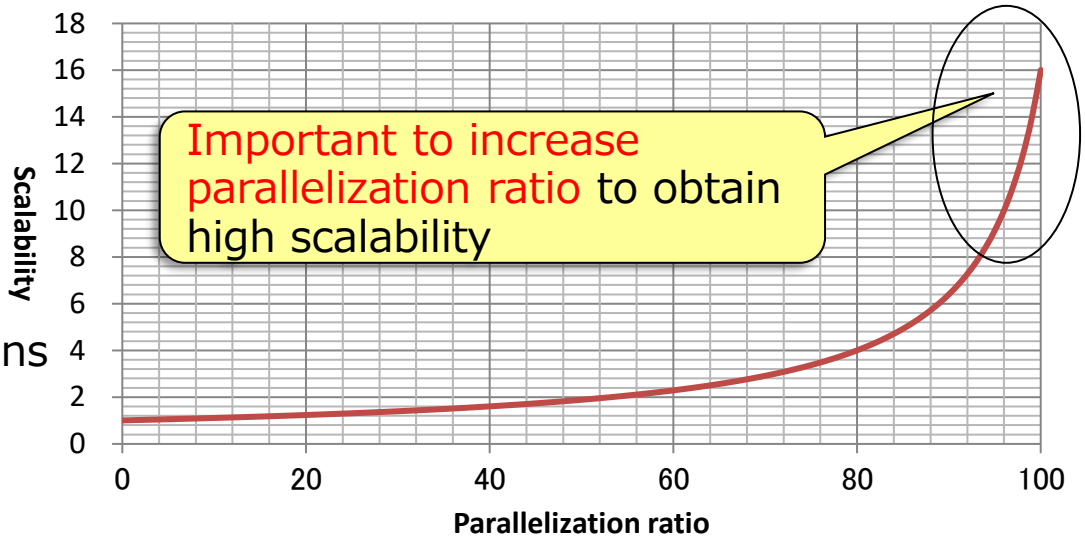
Amdahl's law can be used to express the relationship between the thread parallelization ratio and scalability at the time of *n* parallel executions.

Amdahl's law

$$\textbf{Scalability} = \frac{1}{(1-p)+\frac{p}{n}}$$

- *p*: Parallelization ratio
- *n*: Number of parallel executions

Case where *n* = 16 (Ideal 16 parallel executions)



FUJITSU

Increasing the Thread Parallelization Ratio

- Loops With an Unclear Relationship Between Definition and Citation
- Loops Containing Pointer Variables
- Loops With Data Dependency

Loops With an Unclear Relationship Between Definition and Citation

FUJITSU

- **!OCL NORECURRENCE**

  In the following program, the main processing system cannot determine whether loops can be sliced on Array a without problems, since the subscript expression of Array a is another array element y(j). If the programmer knows that loops can be sliced on Array a without problems, the programmer can parallelize loops by specifying the NORECURRENCE specifier.

Source Before Improvement	Source After Improvement
<<< Loop-information Start >>> <<< [OPTIMIZATION] <<< SOFTWARE PIPELINING(IPC: 0.32, ITR: 6, MVE: 2, POL: S) <<< PREFETCH(HARD) Expected by compiler : <<< b, y <<< Loop-information End >>> 6 1 s 2s do i=1,160000 7 1 m 2m a(y(i))=a(y(i))+b(i) 8 1 p 2v end do : jwd5228p-i "a.f90", line 7: This DO loop cannot be parallelized because the order of data definition and citation is different from that of sequential execution. jwd6228s-i "a.f90", line 7: SIMDization of this DO loop is not possible because the order of data definition and citation may be different from that of sequential execution.	5 !ocl norecurrence(a) <<< Loop-information Start >>> <<< [PARALLELIZATION] <<< Standard iteration count: 762 <<< [OPTIMIZATION] <<< SIMD(VL: 16) <<< SOFTWARE PIPELINING(IPC: 2.33, ITR: 416, MVE: 7, POL: S) <<< PREFETCH(HARD) Expected by compiler : <<< y, b <<< Loop-information End >>> 6 1 pp 2v do i=1,160000 7 1 p 2v a(y(i))=a(y(i))+b(i) 8 1 p 2v end do

! Caution !

- **If loops cannot be sliced on the array for which the NORECURRENCE specifier is specified, the main processing system may incorrectly slice loops.**
- **If the array name is omitted, the specifier is enabled for all arrays within the scope.**

Loops Containing Pointer Variables

FUJITSU

- **!OCL NOALIAS**

  Since which storage area part is occupied by a pointer variable is determined at execution, data dependency is unknown and parallelization is not possible. If the programmer knows that pointer variables do not point to the same storage area, the programmer can perform parallelization by specifying the NOALIAS specifier.

Source Before Improvement	Source After Improvement
1 real,dimension(100000),target::x 2 real,dimension(:),pointer::a,b 3 a=>x(1:10000) 4 b=>x(10001:20000) 5 6 <<< Loop-information Start >>> <<< [OPTIMIZATION] <<< PREFETCH(HARD) Expected by compiler : <<< x <<< Loop-information End >>> 7 1 s 4s do i=1,100000 8 1 s 4s b(i) = a(i)+1.0 9 1 s 4s end do jwd5228p-i "a.f90", line 8: This DO loop cannot be parallelized because the order of data definition and citation is different from that of sequential execution. jwd6228s-i "a.f90", line 8: SIMDization of this DO loop is not possible because the order of data definition and citation may be different from that of sequential execution.	1 real,dimension(100000),target::x 2 real,dimension(:),pointer::a,b 3 a=>x(1:10000) 4 b=>x(10001:20000) 5 6 !ocl noalias <<< Loop-information Start >>> <<< [PARALLELIZATION] <<< Standard iteration count: 1143 <<< [OPTIMIZATION] <<< SIMD(VL: 16) <<< SOFTWARE PIPELINING(IPC: 2.75, ITR: 288, MVE: 4, POL: S) <<< PREFETCH(HARD) Expected by compiler : <<< (unknown) <<< Loop-information End >>> 7 1 pp 2v do i=1,100000 8 1 p 2v b(i) = a(i)+1.0 9 1 p 2v end do

Loops With Data Dependency

FUJITSU

- **Parallelization using peeling**

The following loop is not parallelized because it has dependency with regard to Array a when i=1 and i=n. Take the beginning or end of the loop out of the loop to facilitate parallelization.

Source Before Improvement	Source Before Improvement
<<< Loop-information Start >>> <<< [OPTIMIZATION] <<< PREFETCH(HARD) Expected by compiler : <<< b, a <<< Loop-information End >>> 4 1 s 2s do i=1,n 5 1 s 2m a(i)=a(1)+b(i)+a(n) 6 1 s 2v end do jwd5202p-i "a.f90", line 5: This DO loop cannot be parallelized because the order of data definition and citation is different from that of sequential execution. (Name: a) jwd5208p-i "a.f90", line 5: The order of definition and citation is unknown. For this reason, the order of definition and citation may be different from that of sequential execution, and this DO loop cannot be parallelized. (Name: a)	4 a(1)=a(1)+b(1)+a(n) <<< Loop-information Start >>> <<< [PARALLELIZATION] <<< Standard iteration count: 843 <<< [OPTIMIZATION] <<< SIMD(VL: 16) <<< SOFTWARE PIPELINING(IPC: 3.00, ITR: 384, MVE: 4, POL: S) <<< PREFETCH(HARD) Expected by compiler : <<< a, b <<< Loop-information End >>> 5 1 pp 2v do i=2,n-1 6 1 p 2v a(i)=a(1)+b(i)+a(n) 7 1 p 2v enddo 8 a(n)=a(1)+b(n)+a(n)

FUJITSU

Improving Thread Parallelization Execution Efficiency

- Improving False Sharing
- Loops With Irregular Throughput
- Parallelization in the Appropriate Parallelization Dimension

FUJITSU

Improving False Sharing

- What is False Sharing?
- False Sharing (Before Improvement)
- False Sharing (Source Tuning)

What is False Sharing?

FUJITSU

False sharing is a phenomenon where cache line invalidation and copy back between threads are frequent occurrences.

Example where 4-thread parallelization is assumed

Source Before Improvement

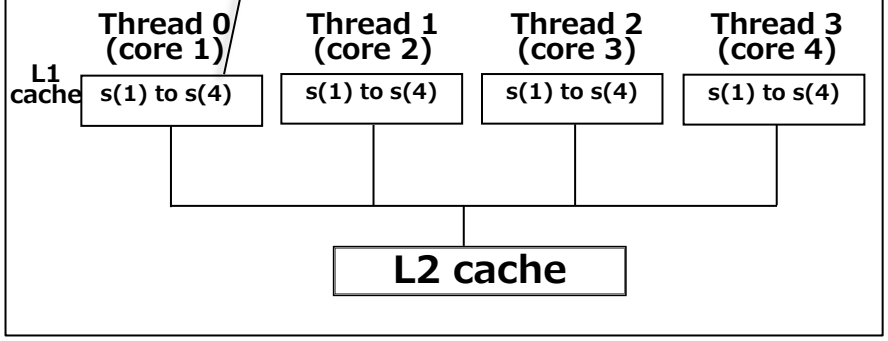
```
 1                    subroutine sub(s,a,b,ni,nj)
 2                    real*8 a(ni,nj),b(ni,nj)
 3                    real*8 s(nj)
 4
 5   1  pp              do j = 1, nj
 6   1   p               s(j)=0.0
 7   2   p   8v            do i = 1, ni
 8   2   p   8v              s(j)=s(j)+a(i,j)*b(i,j)
 9   2   p   8v            end do
10   1   p               end do
11
12                    end
```

nj = 4
ni = 2000

Data cached in units of cache lines

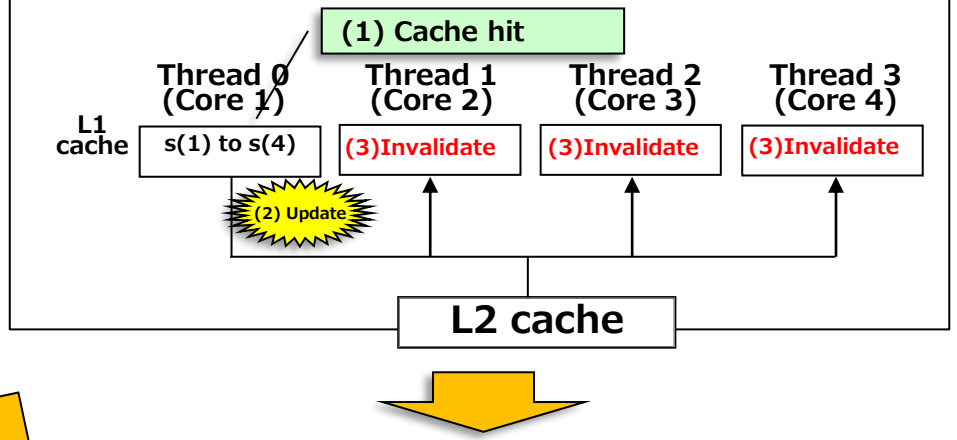
■ Initial state

Each thread reads same cache line containing s(1) to s(4)



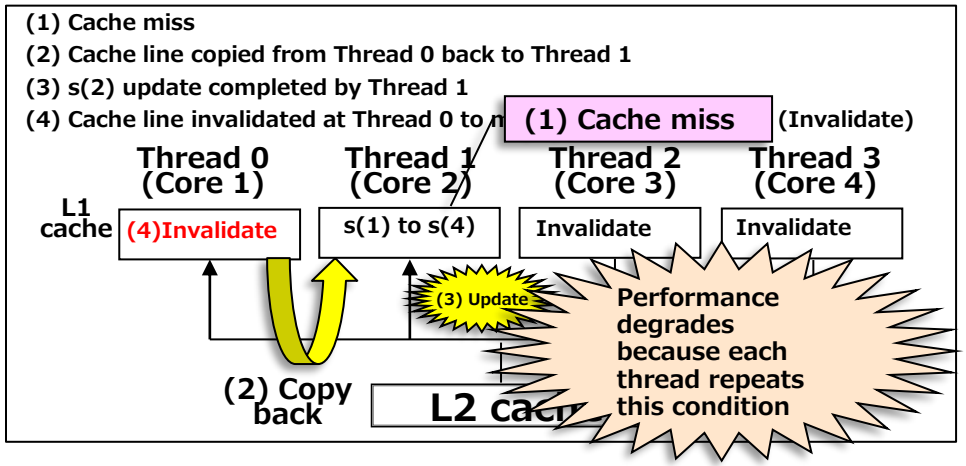
■ Thread 0 specifies s(1) update

(1) Cache hit
(2) s(1) update completed by Thread 0
(3) Cache lines invalidated at Threads 1 to 3 to maintain data coherency (Invalidate)



■ Thread 1 specifies s(2) update

(1) Cache miss
(2) Cache line copied from Thread 0 back to Thread 1
(3) s(2) update completed by Thread 1
(4) Cache line invalidated at Thread 0 to n (Invalidate)



Fortran False Sharing (Before Improvement)

FUJITSU

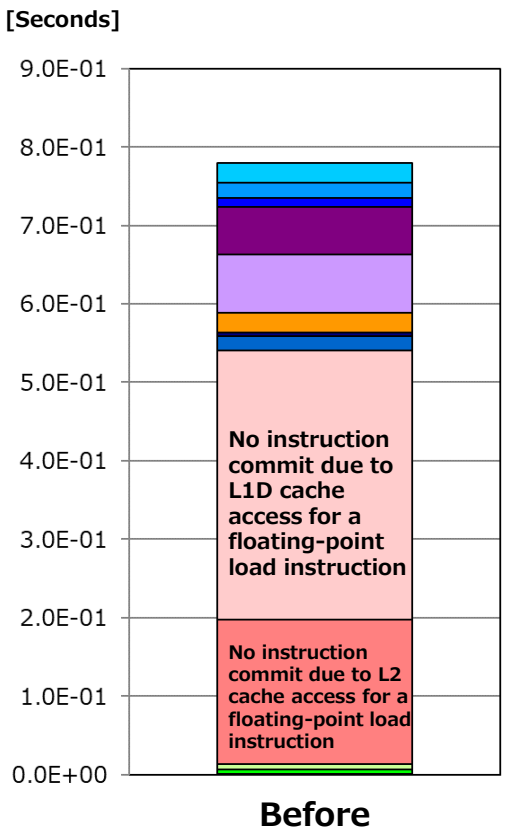
False sharing occurs because the number of iterations of the parallelization dimension j is small (16 iterations) and Array a data shares a cache line between threads. Consequently, the data access wait time is long.

Source Before Improvement

```
22                    subroutine sub(flag)
23                    integer*8 i,j,n
24                    parameter(n=60000)
25                    parameter(m=16)
26                    real*8 a(m,n),b(m,n)
27                    integer flag(m,n)
28                    common /com/a,b
29
:
32    1   p             do i=1,m
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 2.28, ITR: 128,
                                                 MVE: 3, POL: S)
                  <<< Loop-information  End >>>
33    2   p   2v          do j=1,n
34    3   p   2v            if(flag(i,j).eq.1)th
35    3   p   2v              a(i,j)=b(j,i)
36    3   p   2v            endif
37    2   p   2v          enddo
38    1   p             enddo
```

Number of parallelization dimensions: 16

False sharing occurs



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.57E+09	7.27E+08	0.20	9.39%	90.85%	-0.23%	1.75E+04	0.00	21.82%	100.00%	0.00%

Fortran False Sharing (Source Tuning)

FUJITSU

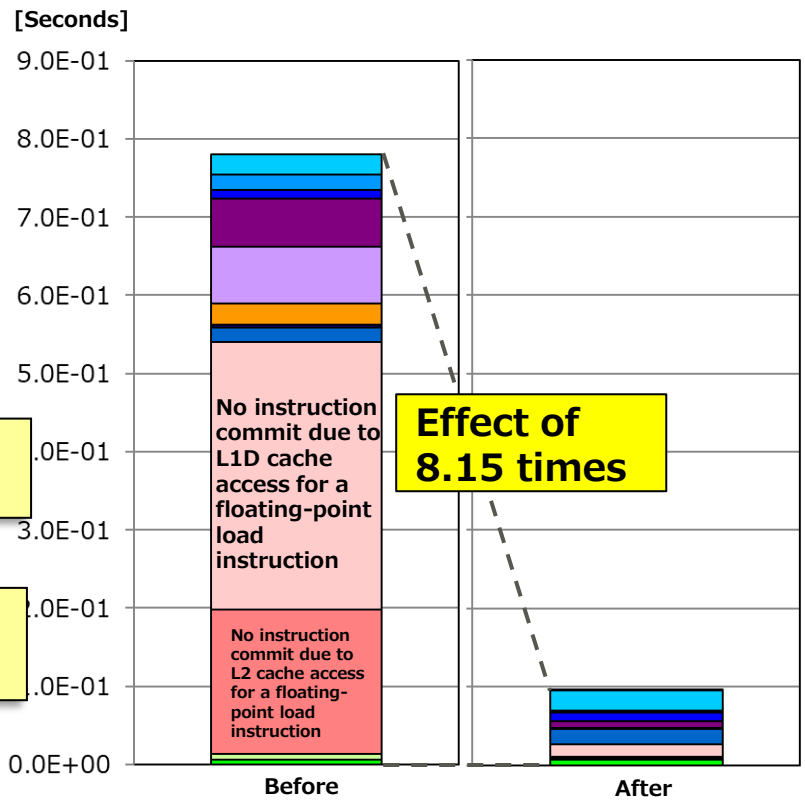
False sharing can be avoided through loop interchange and outer parallelization. This results in fewer L1 cache misses and an improved data access wait time.

Source After Improvement

```
23                    integer*8 i,j,n
24                    parameter(n=60000)
25                    parameter(m=16)
26                    real*8 a(m,n),b(m,n)
27                    integer flag(m,n)
:
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    SOFTWARE PIPELINING(IPC: 1.32, ITR: 48,
                                         MVE: 2, POL: S)
              <<<    PREFETCH(SOFT) : 4
              <<<     SEQUENTIAL : 4
              <<<      b: 4
              <<< Loop-information  End >>>
32    1  p  2          do j=1,n
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    SIMD(VL: 8)
              <<<    FULL UNROLLING
              <<< Loop-information  End >>>
33    2  p  fv           do i=1,m
34    3  p  fv             if(flag(i,j).eq.1)then
35    3  p  fv               a(i,j)=b(j,i)
36    3  p  fv             endif
37    2  p  fv           enddo
38    1  p  2          enddo
```

Loop interchange and outer parallelization

False sharing prevented



L1D misses reduced and performance raised because false sharing prevented

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	3.57E+09	7.27E+08	0.20	9.39%	90.85%	-0.23%	1.75E+04	0.00	21.82%	100.00%	0.00%
After	0.00	1.19E+09	5.29E+07	0.04	4.40%	84.91%	10.70%	1.61E+04	0.00	7.89%	85.83%	6.28%

False Sharing (Before Improvement)

False sharing occurs because the number of iterations of the parallelization dimension j is small (16 iterations) and Array a data shares a cache line between threads. Consequently, the data access wait time is long.

Source Before Improvement

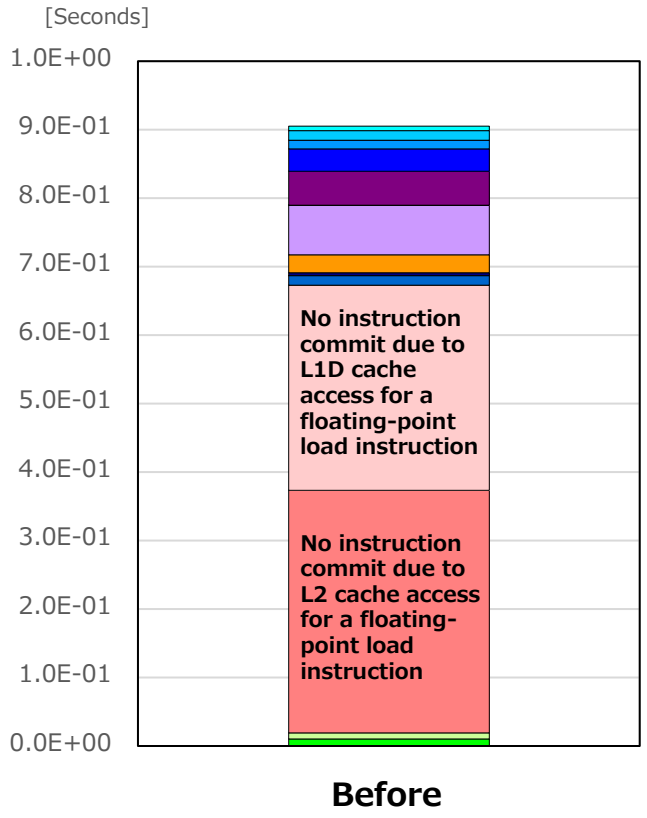
```
42              #pragma omp for
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expect
            <<<      b, a, (unknown)
            <<< Loop-information  End >>>
43  p           for (i=0;i<M;i++)
44  p           {
            <<< Loop-information Sta
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.00, ITR: 128,
                                MVE: 3, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      b, a, (unknown)
            <<< Loop-information  End >>>
45  p   2v      for(j=0;j<N;j++)
46  p   2v      {
47  p   2v        if(flag[j][i]==1)
48  p   2v        {
49  p   2v          a[j][i]=b[i][j];
50  p   2v        }
51  p   2v      }
52  p           }
```

M=16
N=60000

Array declaration double a[N][M], b[N][M];

Number of parallelization dimensions: 16

False sharing occurs



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	4.04E+09	7.28E+08	0.18	30.45%	69.61%	0.00%	3.94E+04	0.00	48.46%	58.66%	0.00%

False Sharing (Source Tuning)

FUJITSU

False sharing can be avoided through loop interchange and outer parallelization. This results in fewer L1 cache misses and an improved data access wait time.

Source After Improvement

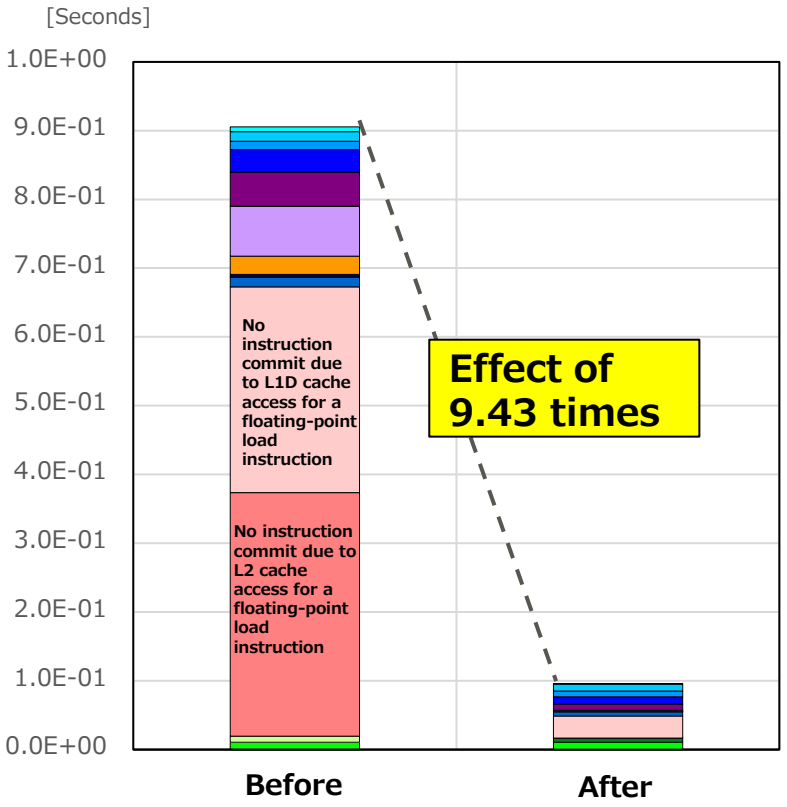
```
42              #pragma omp for
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SOFTWARE PIPELINING(
                           MVE: 3, POL: S)
            <<<    PREFETCH(HARD) Expect
            <<<      a, (unknown)
            <<< Loop-information  End >>>
43  p           for(j=0;j<N;j++)
44  p           {
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    FULL UNROLLING
            <<< Loop-information  End >>>
45  p   fv      for (i=0;i<M;i++)
46  p   fv      {
47  p   fv        if(flag[j][i]==1)
48  p   fv        {
49  p   fv          a[j][i]=b[i][j];
50  p   fv        }
51  p   fv      }
52  p         }
```

M=16
N=60000

Array declaration
double a[N][M],
b[N][M];

Loop interchange and outer parallelization

False sharing prevented



L1D misses reduced and performance raised because false sharing prevented

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before	0.00	4.04E+09	7.28E+08	0.18	30.45%	69.61%	0.00%	3.94E+04	0.00	48.46%	58.66%	0.00%
After	0.00	1.27E+09	4.81E+07	0.04	6.15%	93.86%	0.00%	1.93E+04	0.00	15.65%	89.19%	0.00%

FUJITSU

Loops With Irregular Throughput

- Loops With Irregular Throughput (Before Improvement)
- Loops With Irregular Throughput (OpenMP Tuning)

Fortran Loops With Irregular Throughput (Before Improvement)

FUJITSU

If throughput is irregular, cyclic division with a static scheduling method causes a load imbalance. In the following example, the "Synchronous waiting time between threads" event occurs many times due to the load imbalance.

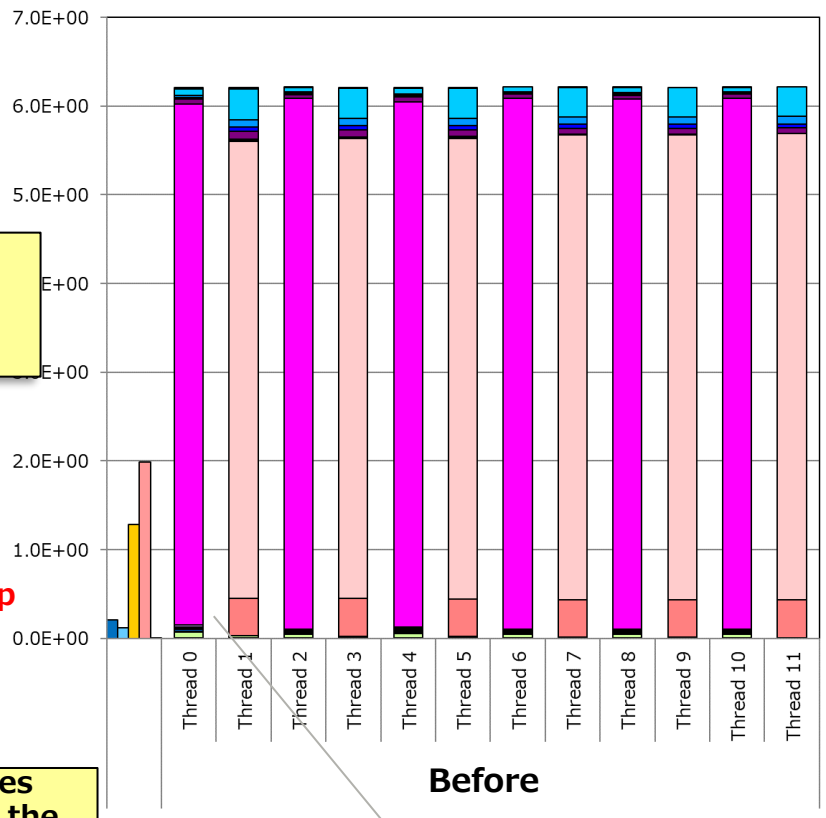
Source Before Improvement

```
1                 subroutine init(a,b,ie,n)
:
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    FUSED
                  <<< Loop-information  End >>>
8    1              do i=1,n
9    2                if (mod(i,2).eq.0) then
10   2                  ie(i)=100000
11   2                endif
12   1              enddo
:
16                subroutine sub(a,b,s,ie,n)
17                 real a(n),b(n),s
18                 integer ie(n)
19                 !$omp parallel do schedule(static,1)
20   1  p            do j=1,n
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 16)
                  <<<    SOFTWARE PIPELINING(IPC: 3.50,
                                 ITR: 448, MVE: 7, POL: S)
                  <<< Loop-information  End >>>
21   2  p  2v         do i=1,ie(j)
22   2  p  2v           a(i) = a(i)*b(i)*s
23   2  p  2v         enddo
24   1  p            enddo
25                end subroutine sub
 :
27                program main
28                 parameter(n=1000000)
31                 call init(a,b,ie,n)
 :
33                 call sub(a,b,2.0,ie,n)
 :
35                end program main
```

Sets value in array ie, which has ending value of evaluation loop, only at even-numbered iterations

Evaluation loop

The innermost loop iterates 100,000 times only when the control variable j is an even number.



Synchronous waiting time between threads occurs due to load imbalance

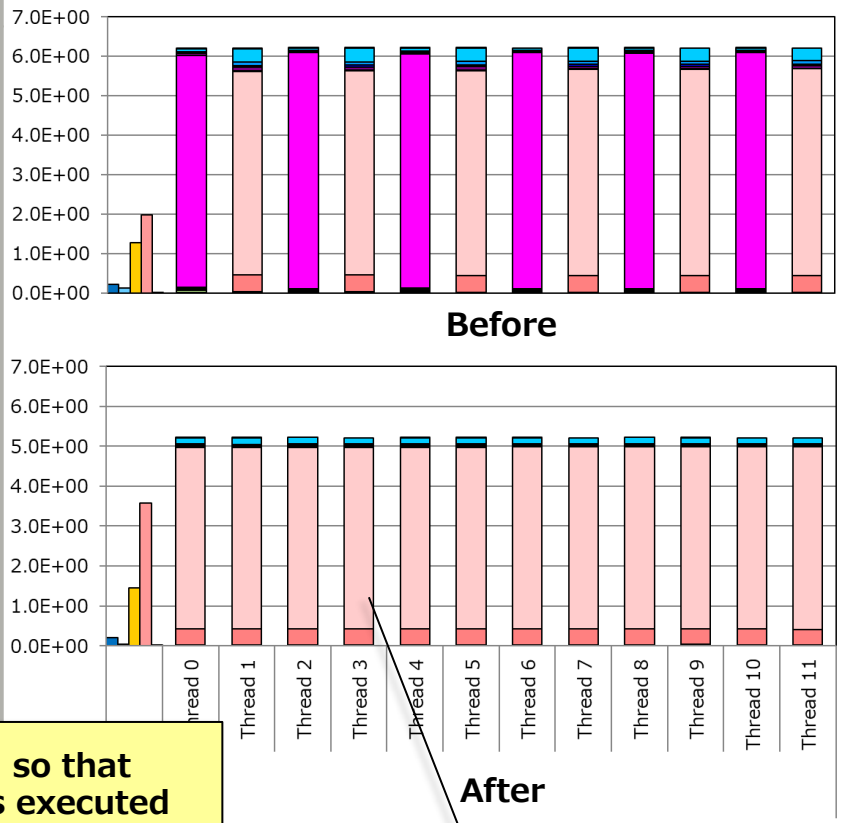
Fortran Loops With Irregular Throughput (OpenMP Tuning)

FUJITSU

Change to a dynamic scheduling method so that the next processing can be done by a thread that finishes processing earlier than other threads. The result is improvement in the load imbalance.

Source After Improvement

```
1                 subroutine init(a,b,ie,n)
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    FUSED
                 <<< Loop-information  End >>>
8     1             do i=1,n
9     2               if (mod(i,2).eq.0) then
10    2                 ie(i)=100000
11    2               endif
12    1             enddo
 :
16                 subroutine sub(a,b,s,ie,n)
17                   real a(n),b(n),s
18                   integer ie(n)
19                   !$omp parallel do schedule(dynamic,1)
20    1   p           do j=1,n
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 16)
                 <<<    SOFTWARE PIPELINING(IPC: 3.50, ITR: 448,
                                            MVE: 7, POL: S)
                 <<< Loop-information  End >>>
21    2   p   2v          do i=1,ie(j)
22    2   p   2v            a(i) = a(i)*b(i)*s
23    2   p   2v          enddo
24    1   p           enddo
25                 end subroutine sub
:
27                 program main
28                   parameter(n=1000000)
31                   call init(a,b,ie,n)
:
33                   call sub(a,b,2.0,ie,n)
```



dynamic specified so that next processing is executed by thread that completes processing earlier

Load imbalance improved

C/C++ Loops With Irregular Throughput (Before Improvement)

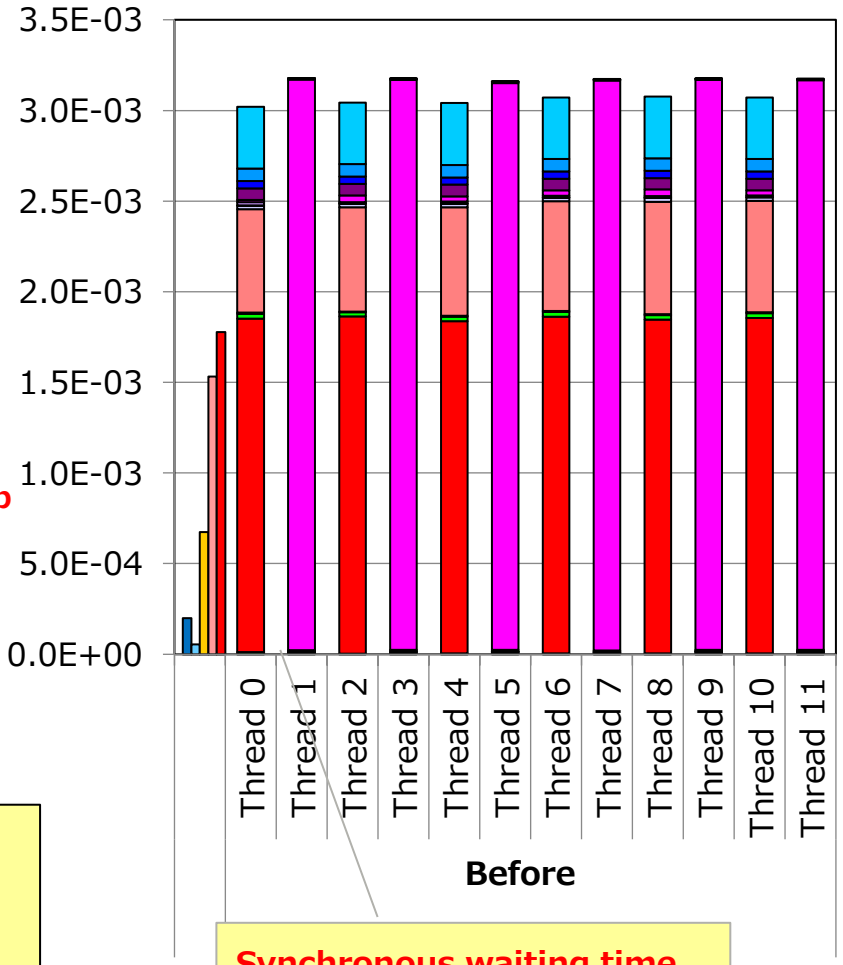
FUJITSU

If throughput is irregular, cyclic division with a static scheduling method causes a load imbalance. In the following example, the "Synchronous waiting time between threads" event occurs many times due to the load imbalance.

Source Before Improvement

```
28  void sub(int n, float (* restrict a)[n], float * restrict b,
                            float s, int * restrict ie){
29            int i, j;
30
31           #pragma omp parallel for schedule(static, 1)
             <<< Loop-information Start >>>
             <<<  [OPTIMIZATION]
             <<<    PREFETCH(HARD) Expected by compiler :
             <<<      (unknown)
             <<< Loop-information  End >>>
32   p         for (j = 0; j < n; j++){
             <<< Loop-information Start >>>
             <<<  [OPTIMIZATION]
             <<<    SIMD(VL: 16)
             <<<    SOFTWARE PIPELINING(IPC: 3.50,
                        ITR: 448, MVE: 7, POL: S)
             <<<    PREFETCH(HARD) Expected by compiler :
             <<<      (unknown)
             <<< Loop-information  End >>>
33   p    2v     for (i = 0; i < ie[j]; i++){
34   p    2v       a[j][i] = a[j][i]*b[i]*s;
35   p    2v     }
36   p         }
37           }
```

Evaluation loop



The innermost loop iterates 100,000 times when the control variable j is an even number. The innermost loop iterates 1,000,000 times when the control variable j is an odd number.

Synchronous waiting time between threads occurs due to load imbalance

C/C++ Loops With Irregular Throughput (OpenMP Tuning)

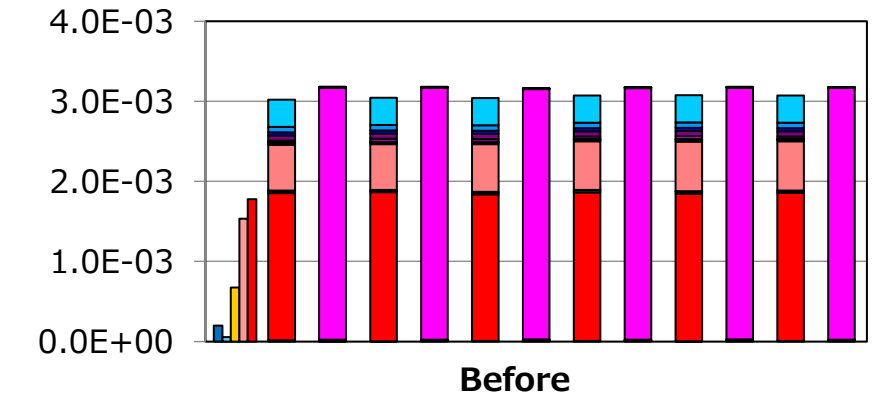
FUJITSU

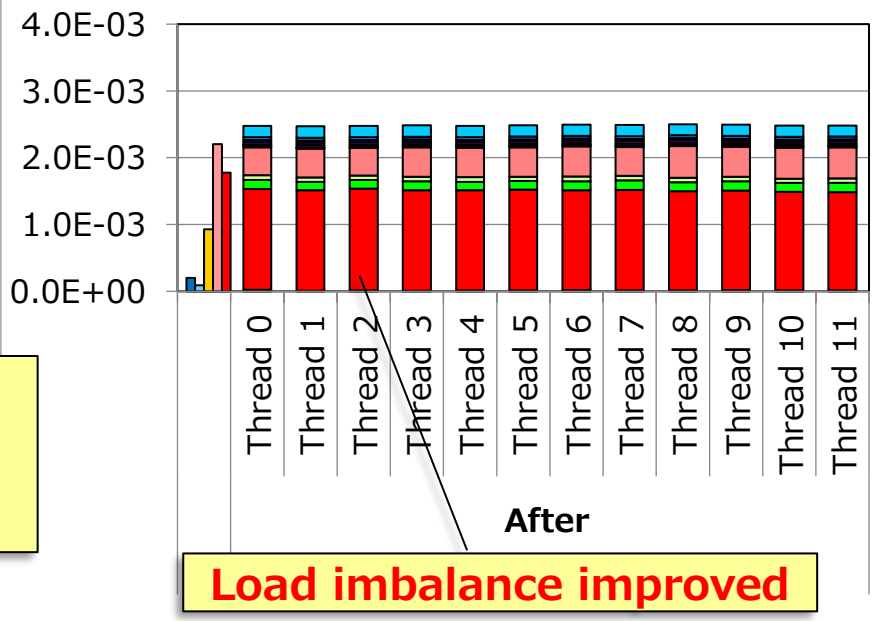
Change to a dynamic scheduling method so that the next processing can be done by a thread that finishes processing earlier than other threads. The result is improvement in the load imbalance.

Source After Improvement

```
28 void sub(int n, float (* restrict a)[n],
              float * restrict b, float s, int * restrict ie){
29           int i, j;
30
31          #pragma omp parallel for schedule(dynamic, 1)
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
32  p        for (j = 0; j < n; j++){
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 16)
           <<<    SOFTWARE PIPELINING(IPC: 3.50, ITR: 448,
                                MVE: 7, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
33  p    2v     for (i = 0; i < ie[j]; i++){
34  p    2v       a[j][i] = a[j][i]*b[i]*s;
35  p    2v     }
36  p         }
37          }
```

dynamic specified so that next processing is executed by thread that completes processing earlier





Load imbalance improved

FUJITSU

Parallelization in the Appropriate Parallelization Dimension

- Parallelization in the Appropriate Parallelization Dimension (Before Improvement)
- Parallelization in the Appropriate Parallelization Dimension (Source Tuning)
- Parallelization in the Appropriate Parallelization Dimension (Compiler Option Tuning)
- Parallelization in the Appropriate Parallelization Dimension (OpenMP Source Tuning)

Fortran Parallelization in the Appropriate Parallelization Dimension (Before Improvement)

FUJITSU

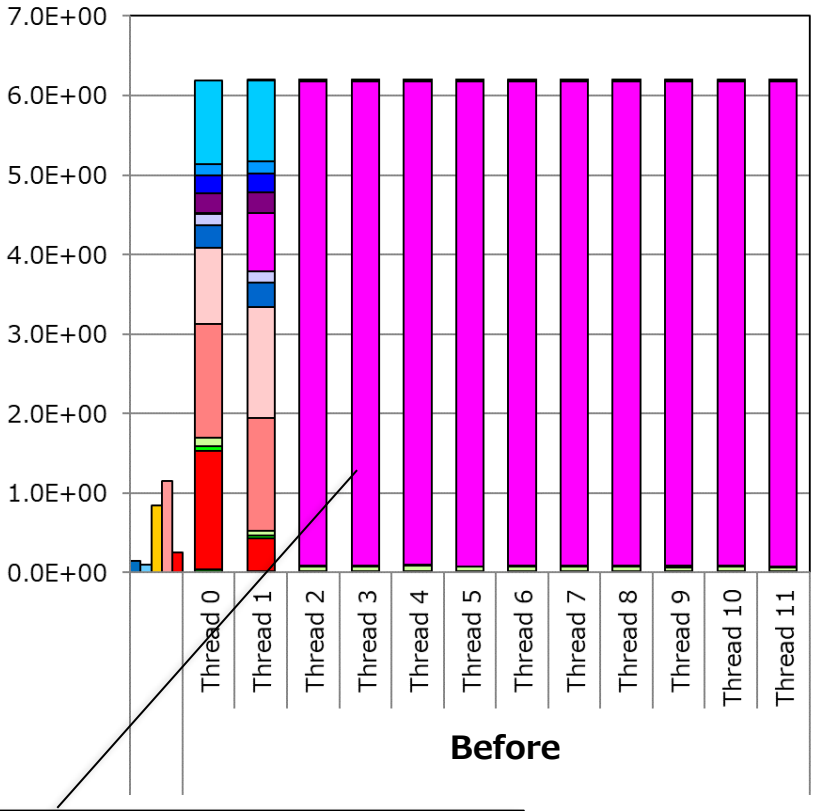
If the number of loop iterations in a parallelization dimension is small and unknown at the compile time, a load imbalance occurs when there are fewer iterations than parallel threads (12 in the example). Consequently, the "Synchronous waiting time between threads" event occurs many times.

Source Before Improvement

```
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 2
                <<< Loop-information  End >>>
32    1  pp          do k=1,l
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, b, a
                <<< Loop-information  End >>>
33    2  p           do j=1,m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.25,
                                    ITR: 144, MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, b, a
                <<< Loop-information  End >>>
34    3  p   2v            do i=1,n
35    3  p   2v              a(i,j,k)=b(i,j,k)+c(i,j,k)
36    3  p   2v            enddo
37    2  p             enddo
38    1  p           enddo
```

l = 2
m = 512
n = 256

Poor load balance among threads



Load imbalance occurs because number of iterations in parallelization dimension k is 2

Parallelization in the Appropriate Parallelization Dimension (Source Tuning)

FUJITSU

Specify the SERIAL and PARALLEL specifiers to perform parallelization in the appropriate dimension. The result is improvement in the load imbalance.

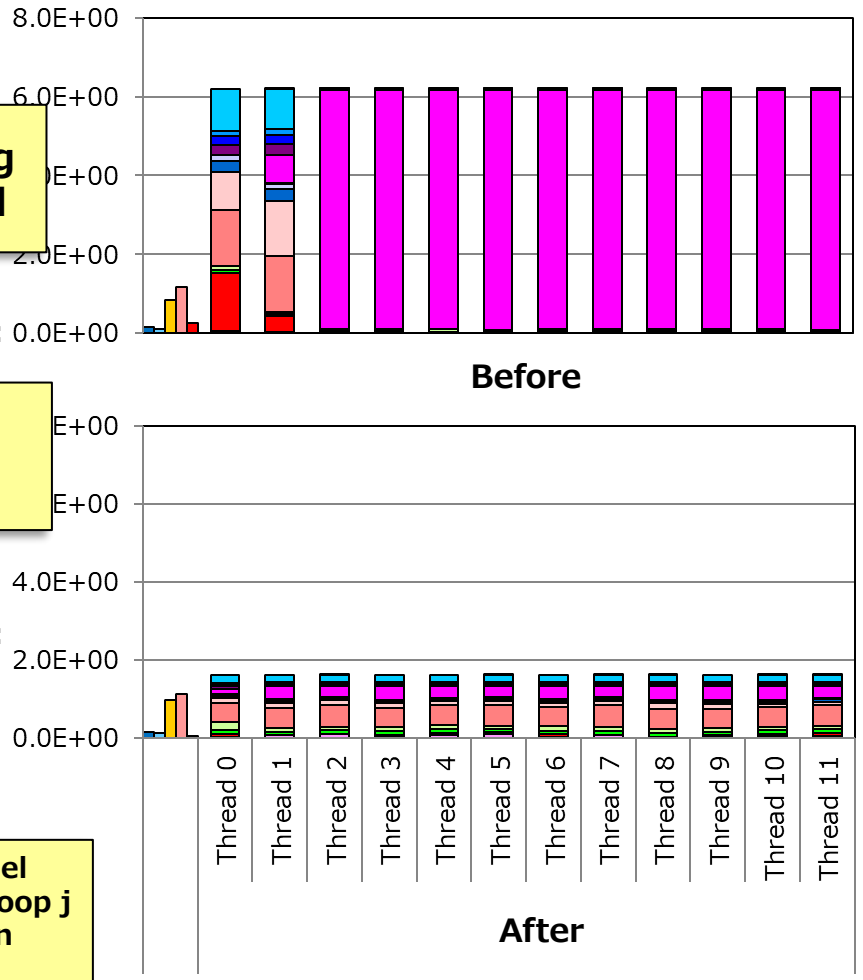
Source After Improvement

```
31                !ocl serial
32    1             do k=1,l
33    1            !ocl parallel
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 4
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, b, a
                <<< Loop-information  End >>>
34    2  pp           do j=1,m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.25,
                                  ITR: 144, MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, b, a
                <<< Loop-information  End >>>
35    3   p   2v          do i=1,n
36    3   p   2v            a(i,j,k)=b(i,j,k)+c(i,j,k)
37    3   p   2v          enddo
38    2   p           enddo
39    1             enddo
```

Loop slicing suppressed

l = 2
m = 512
n = 256

Executed in parallel when number of loop j iterations is 512 in parallelization dimension



Fortran Parallelization in the Appropriate Parallelization Dimension (Compiler Option Tuning)

FUJITSU

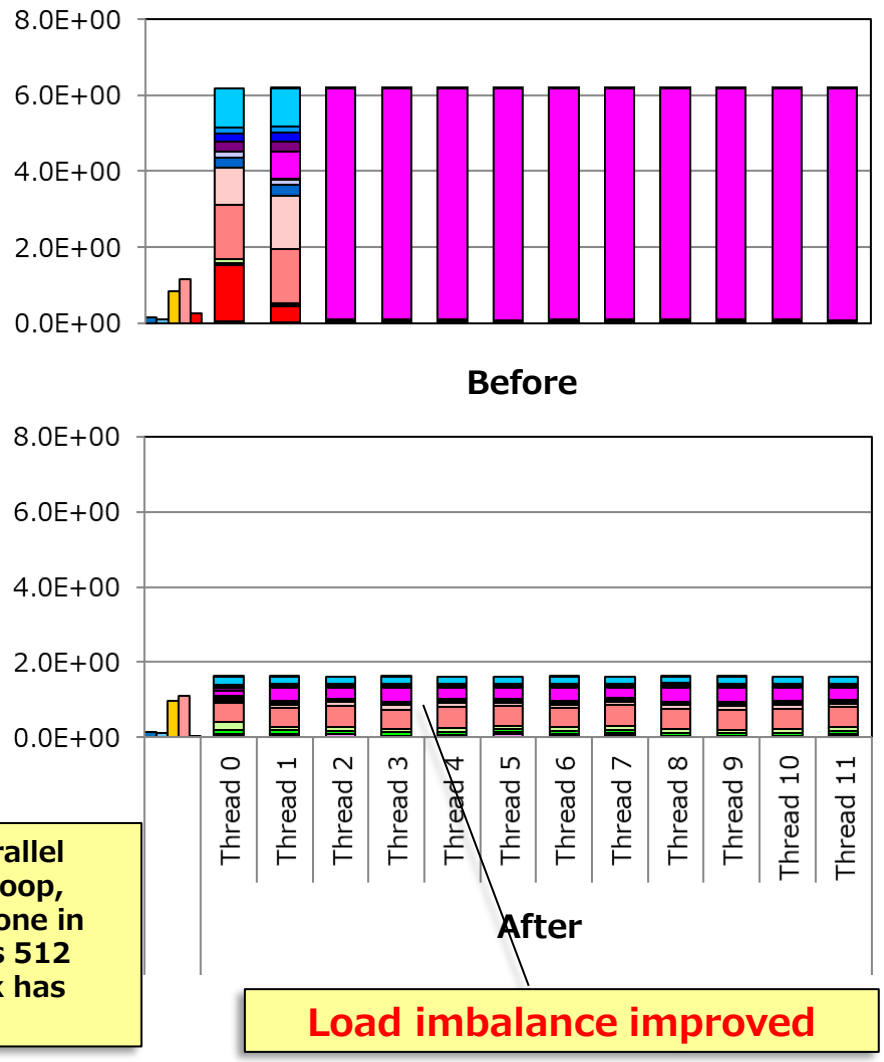
Specify the compiler option -Kdynamic_iteration to automatically select the appropriate parallelization dimension at execution and improve the load imbalance.

Source After Improvement

```
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 2
                <<< Loop-information  End >>>
31    1  pp          do k=1,l
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 4
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, b, a
                <<< Loop-information  End >>>
32    2  pp            do j=1,m
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 728
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.25,
                                  ITR: 144, MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, b, a
                <<< Loop-information  End >>>
33    3  pp  2v            do i=1,n
34    3   p  2v              a(i,j,k)=b(i,j
35    3   p  2v            enddo
36    2   p              enddo
37    1   p            enddo
```

l = 2
m = 512
n = 256

Despite attempt at parallel execution from outer loop, parallel execution is done in inner loop j, which has 512 iterations, since loop k has few iterations (2)



Fortran Parallelization in the Appropriate Parallelization Dimension (OpenMP Source Tuning)

FUJITSU

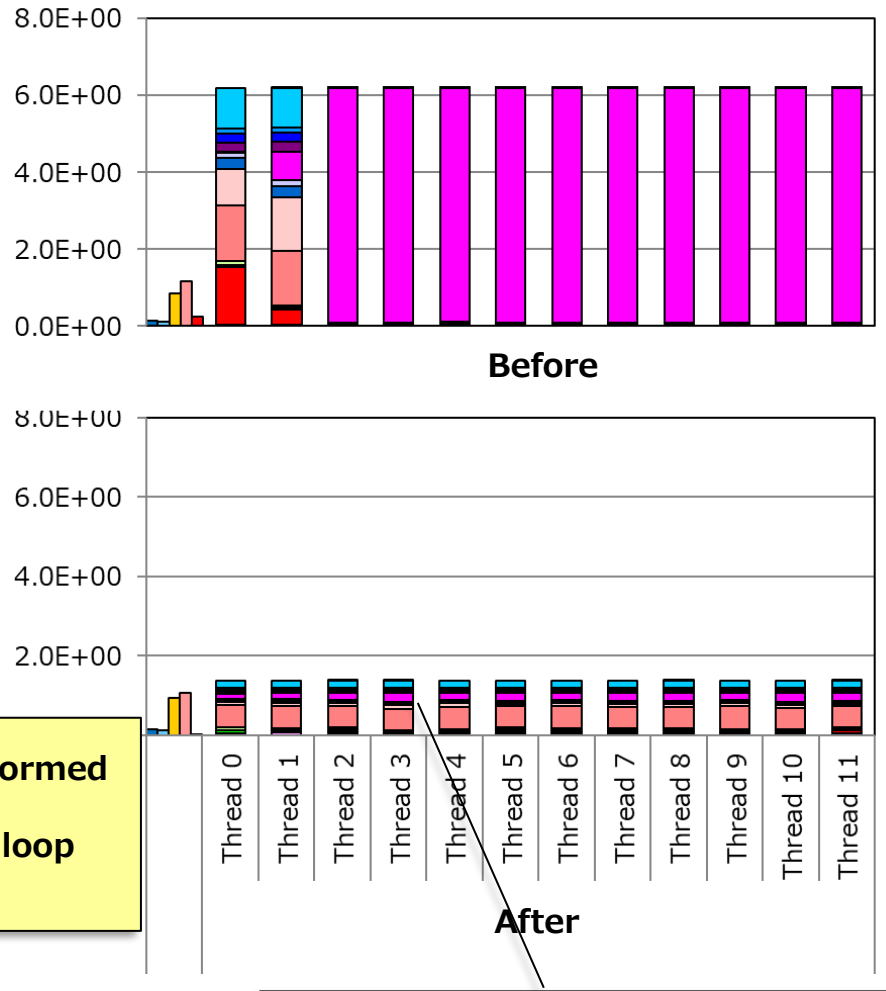
Specify the OpenMP COLLAPSE clause to transform outer loops into a single loop. The result is improvement in the load imbalance.

Source After Improvement

```
32            !$omp parallel do private(k,j,i) collapse(2)
        <<< Loop-information Start >>>
        <<<  [OPTIMIZATION]
        <<<    PREFETCH(HARD) Expected by compiler :
        <<<      c, b, a
        <<< Loop-information  End >>>
33    1   p            do k=1,l
34    2   p              do j=1,m
        <<< Loop-information Start >>>
        <<<  [OPTIMIZATION]
        <<<    SIMD(VL: 8)
        <<<    SOFTWARE PIPELINING(IPC: 3.25,
                                   ITR: 144, MVE: 4, POL: S)
        <<<    PREFETCH(HARD) Expected by compiler :
        <<<      c, b, a
        <<< Loop-information  End >>>
35    3   p   2v             do i=1,n
36    3   p   2v               a(i,j,k)=b(i,j,k)+c(i,j,k)
37    3   p   2v             enddo
38    2   p              enddo
39    1   p            enddo
40                  !$omp end pa
```

l = 2
m = 512
n = 256

COLLAPSE clause transformed loop k, which has few iterations, and its inner loop into single loop



Load imbalance improved

- Note
  - Be careful not to COLLAPSE the SIMD transformation axis (innermost loop).

C/C++ Parallelization in the Appropriate Parallelization Dimension (Before Improvement)

FUJITSU

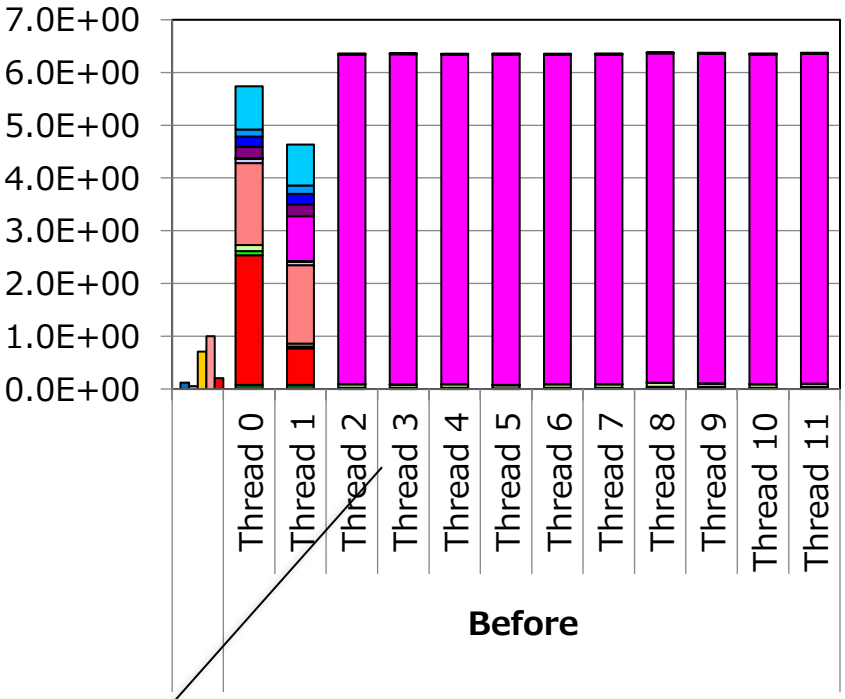
If the number of loop iterations in a parallelization dimension is small and unknown at the compile time, a load imbalance occurs when there are fewer iterations than parallel threads (12 in the example). Consequently, the "Synchronous waiting time between threads" event occurs many times.

Source Before Improvement

```
42          #pragma omp parallel for
43   p        for (k = 0; k < l; k++){
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    PREFETCH(HARD) Expected by compiler :
              <<<      (unknown)
              <<< Loop-information  End >>>
44   p          for (j = 0; j < m; j++){
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    SIMD(VL: 8)
              <<<    SOFTWARE PIPELINING(IPC: 3.00,
                       ITR: 144, MVE: 5, POL: S)
              <<<    PREFETCH(HARD) Expected by compiler :
              <<<      (unknown)
              <<< Loop-information  End >>>
45   p    2v       for (i = 0; i < n; i++){
46   p    2v         a[k][j][i] = b[k][j][i] + c[k][j][i];
47   p    2v       }
48   p          }
49   p        }
```

l=2
m=512
n=256

Poor load balance among threads



Load imbalance occurs because number of iterations in parallelization dimension k is 2

Parallelization in the Appropriate Parallelization Dimension (Source Tuning)

FUJITSU

Specify the parallel for or parallel specifiers to perform parallelization in the appropriate dimension. The result is improvement in the load imbalance.

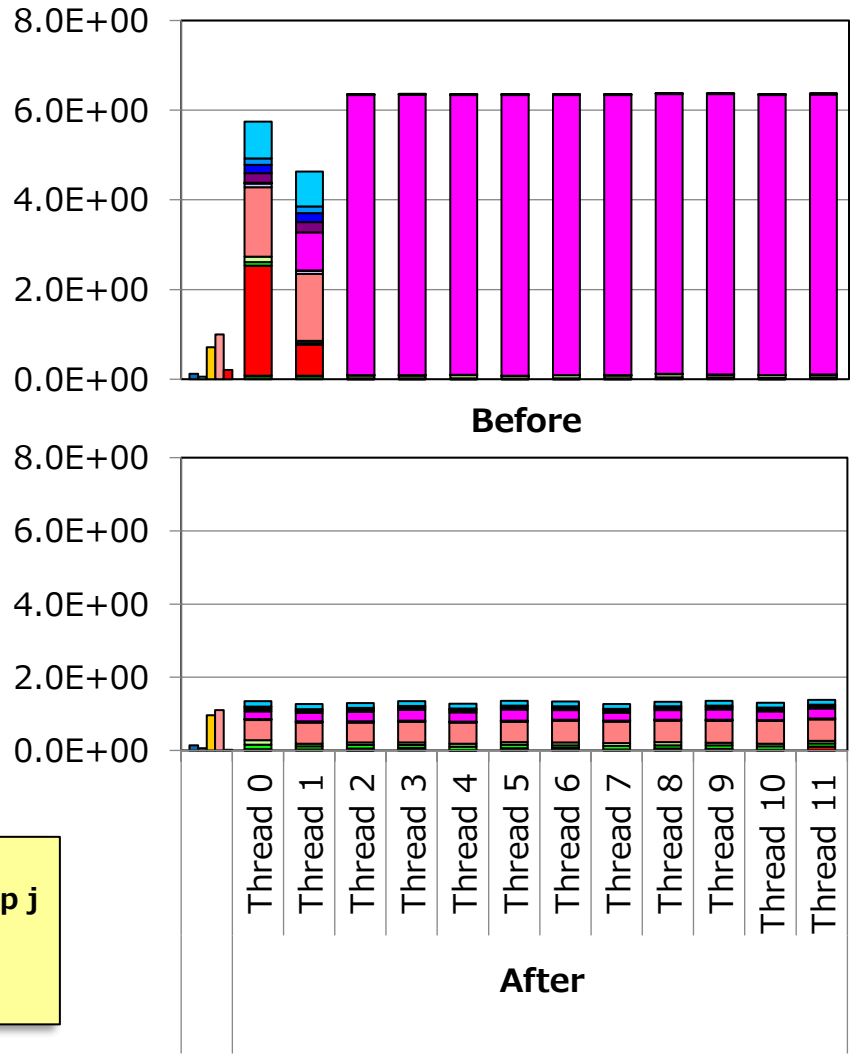
Source After Improvement

```
42          #pragma omp parallel private(i,j,k)
43          {
44            for (k = 0; k < l; k++){
45          #pragma omp for
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
46  p           for (j = 0; j < m; j++){
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 3.00,
                          ITR: 144, MVE: 5, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
47  p   2v        for (i = 0; i < n; i++){
48  p   2v          a[k][j][i] = b[k][j][i] + c[k][j][i];
49  p   2v        }
50  p           }
51            }
```

l=2
m=512
n=256

Loop slicing suppressed

Executed in parallel when number of loop j iterations is 512 in parallelization dimension



C/C++ Parallelization in the Appropriate Parallelization Dimension (Compiler Option Tuning)

FUJITSU

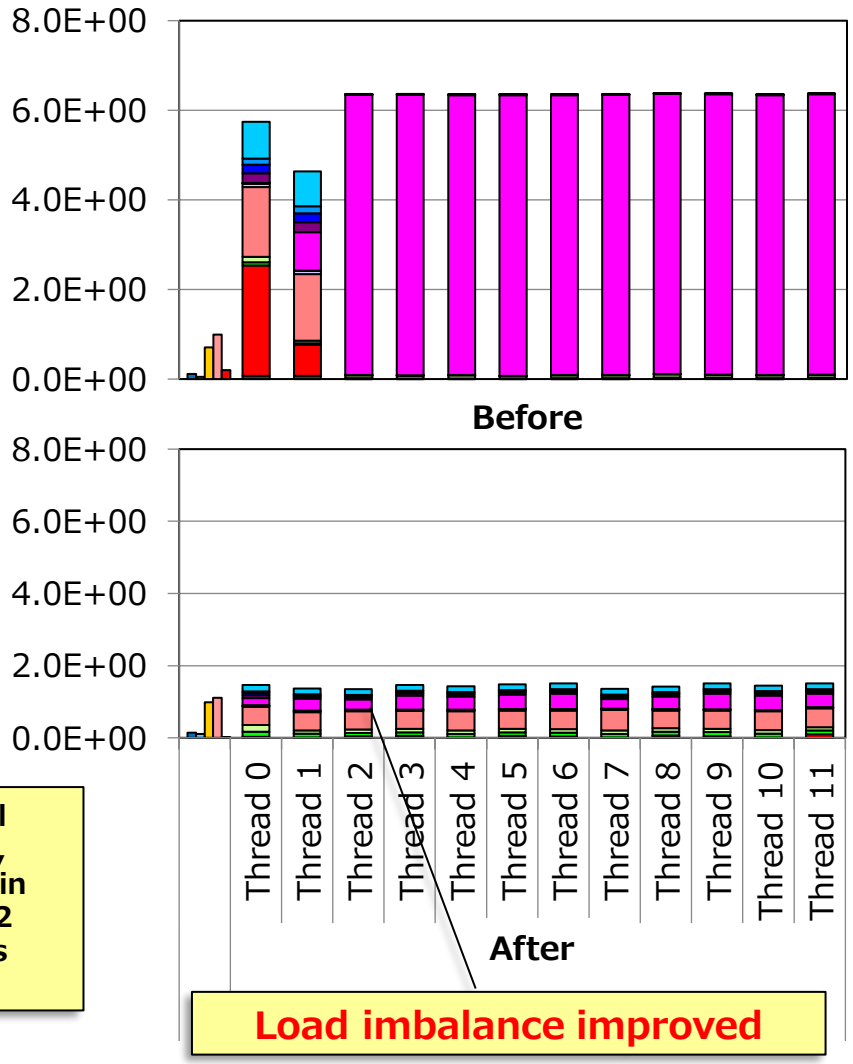
Specify the compiler option -Kdynamic_iteration to automatically select the appropriate parallelization dimension at execution and improve the load imbalance. The optimization is not available in Clang Mode.

Source After Improvement

```
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 3
                <<< Loop-information  End >>>
42  pp            for (k = 0; k < n3; k++){
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 37
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      (unknown)
                <<< Loop-information  End >>>
43  pp              for (j = 0; j < n2; j++){
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 552
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.25,
                                 ITR: 144, MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      (unknown)
                <<< Loop-information  E
44  pp    2v          for (i = 0; i < n1; i
45  p     2v            a[k][j][i] = b[k][j
46  p     2v          }
47  p               }
48  p             }
```

l=2
m=512
n=256

Despite attempt at parallel execution from outer loop, parallel execution is done in inner loop j, which has 512 iterations, since loop k has few iterations (2)



C/C++ Parallelization in the Appropriate Parallelization Dimension (OpenMP Source Tuning)

FUJITSU

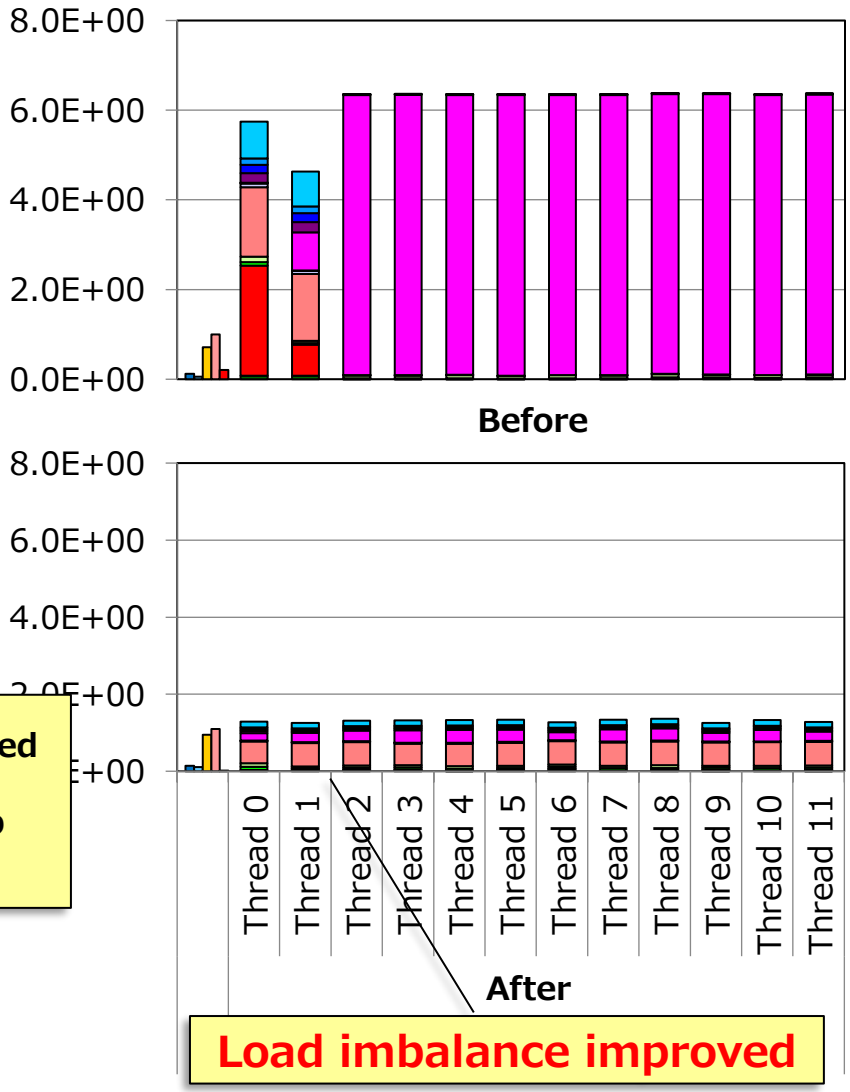
Specify the OpenMP collapse clause to transform outer loops into a single loop. The result is improvement in the load imbalance.

Source After Improvement

```
42          #pragma omp parallel for private(i,j,k) collapse(2)
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
43  p         for (k = 0; k < l; k++){
44  p           for (j = 0; j < m; j++){
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 3.00,
                        ITR: 144, MVE: 5, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
45  p   2v      for (i = 0; i < n; i++){
46  p   2v        a[k][j][i] = b[k][j][i] + c[k][j][i];
47  p   2v      }
48  p         }
49  p       }
```

l=2
m=512
n=256

COLLAPSE clause transformed loop k, which has few iterations, and its inner loop into single loop



- Note
- Be careful not to COLLAPSE the SIMD transformation axis (innermost loop).

FUJITSU

Improving Execution Efficiency by Setting Large Pages

- Specifying a Large Page Paging Policy
- Changing the Lock Type

FUJITSU

Specifying a Large Page Paging Policy

- Large Page Paging Policy
- Effect of a Large Page Paging Policy (demand)

Large Page Paging Policy

FUJITSU

- XOS_MMM_L_PAGING_POLICY=prepage:demand:prepage
  - Memory allocation is as follows when multiple CMGs are running for thread parallelism:
    - In prepaging, data comes from CMG0 at the start of the load module.
    - In demand paging, data is put on the running CMG at the first access time.
  - Demand paging is recommended for processing across multiple CMGs.

Environment Variable Name	Specifiable Value (Default indicated by _)	Description
XOS_MMM_L_PAGING_POLICY	[demand | prepage]: [demand | prepage]: [demand | prepage]	This setting selects the paging method (page allocation trigger) for each memory area. "demand" means the demand paging method, and "prepage" means the prepaging method. This environment variable specifies paging methods for three memory areas by delimiting them with a colon (:). The 1st specification is for the .bss area for static data. ("prepage" is always used for the .data area for static data. No other paging method can be specified.) The 2nd specification is for the stack area and thread stack area. The 3rd specification is for the dynamic memory securing area. If any specified value is not a specifiable value, "prepage:demand:prepage" is assumed specified.

Effect of a Large Page Paging Policy (demand)

FUJITSU

Since data comes from CMG0 in prepaging, performance cannot reach that of 48-thread streams.

With the method changed to demand paging, data is put on the running CMG, and performance is significantly higher.

Source

```
14                Subroutine sub(n,iter,x1,x2,y1)
15                  real(8) :: x1(n), x2(n), y1(n),c0
16                  integer n,i,k
17                  c0=2.0
18
19                  call fapp_start("sub",0,0)
20    1              do k=1,iter
21    1             !$omp parallel do
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 2.45, ITR:
128, MVE: 2, POL: S)
                  <<<    PREFETCH(SOFT) : 10
                  <<<     SEQUENTIAL : 10
                  <<<      x2: 4, x1: 4, y1: 2
                  <<<    ZFILL         :
                  <<<      y1
                  <<< Loop-information  End >>>
22    2  p   v          do i=1,n
23    2  p   v            y1(i) = x1(i) + c0 * x2(i)
24    2  p   v          end do
25    1               enddo
 :                    ......
30                  parameter(N=45000000,ITER=100)
31                  real*8 x1(N),x2(N),y1(N)
32                  call init(N,ITER,x1,x2,y1)
33                  call sub(N,ITER,x1,x2,y1)
```

	Memory throughput (GB/s)
prepage (default)	93 GB/s
demand	804 GB/s

Compiler option: -Kfast,openmp -Kprefetch_sequential=soft -Kprefetch_line=9 -Kprefetch_line_L2=70 -Kzfill=18

Stream (Data size: About 1 GB)

FUJITSU

Changing the Lock Type

- What is the Lock Type (XOS_MMM_L_ARENA_LOCK_TYPE)?
- Effect of Changing the Lock Type

What is the Lock Type (XOS_MMM_L_ARENA_LOCK_TYPE)?

- XOS_MMM_L_ARENA_LOCK_TYPE
  - This is a setting related to the memory allocation policy.
  - "0" gives priority to memory acquisition performance. This is the recommended value when performing malloc in a parallel region. (This may improve performance because memory is acquired/released from an independent memory pool for each thread, reducing the cost of exclusive control as compared to the default setting.)
  - "1" (default) gives priority to memory use efficiency. This is the recommended value when memory usage is high.

Effect of Changing the Lock Type

FUJITSU

malloc performance is higher when XOS_MMM_L_ARENA_LOCK_TYPE=0 is specified. (Reduced execution time from 0.56 seconds to 0.35 seconds, a performance increase of 1.60 times)

Source

```
1                subroutine sub(n,m,iter,x1,x2,y2)
2                  integer(8) :: pZ1(iter)
3                  real(8) :: x1(n), x2(n), y2(n,m),c0
4                  c0=2.0
5
6                  !$omp parallel do shared(n,m,iter,x1,x2,c0,y2) private(pZ1,i,j,k) default(none)
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      x1, x2, y2
                 <<< Loop-information  End >>>
7    1  p           do k=1,m
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      (unknown)
                 <<< Loop-information  End >>>
8    2  p  s         do j=1,iter
9    2  p  m           pZ1(j) = malloc(8 * n)
10   2  p  v         end do
11   1
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELINING(IPC: 3.50, ITR: 144, MVE: 4, POL: S)
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      x1, x2, y2
                 <<< Loop-information  End >>>
12   2  p  2v        do i=1,n
13   2  p  2v          y2(i,k) = x1(i) + c0 * x2(i)
14   2  p  2v        end do
15   1
16   2  p  s         do j=1,iter
17   2  p  s           call free( pZ1(j))
18   2  p  s         end do
19   1  p           end do
20                 end subroutine sub
21
22                 program main
23                   parameter(N=1048512,ITER=80)
24                   real*8 x1(N),x2(N),y2(N,12)
25                   call sub(N,12,ITER,x1,x2,y2)
26                 end program main
```

FUJITSU

Reduced memory usage

- Rewriting OMP SINGLE to OMP MASTER

Rewriting OMP SINGLE to OMP MASTER

FUJITSU

- Reducing memory usage by rewriting
  - Rewriting omp single to omp master and barrier, fixes the execution thread and suppresses redundant use of memory space.

Source Before Improvement	Source After Improvement
!$omp parallel **!$omp single** **call loop(a,b,alpha,max);** **!$omp end single** **!$omp end parallel**	**!$omp parallel** **!$omp master** **call loop(a,b,alpha,max);** **!$omp end master** **!$omp barrier** **!$omp end parallel**

Revision History

FUJITSU

- Revision History

Version	Date	Details
1.0	Sep. 2020	- First published
1.3	Mar. 2021	- Correcting typographical errors and expressions by reviewing articles
1.4	Aug. 2021	- Fixing differences by increasing the number of software versions, and correcting typographical errors and expressions by reviewing articles - Added “What is Unroll-and-Jam?” page - Added “Rewriting OMP SINGLE to OMP MASTER” page
2.1	Jul. 2022	- Added tuning in C/C++ - Format changed - Correcting typographical errors and expressions by reviewing articles
2.2	Mar. 2023	- Correcting typographical errors and expressions by reviewing articles

FUJITSU