

Chapter 8

Intra-Node Tuning

FUJITSU LIMITED
April 2016

■ CPU Tuning

- What Is CPU Tuning (Intra-Node Tuning)?
- Positioning of CPU Tuning

■ How to Effectively Use PA Information and Tuning Flows

■ Navigation from PA Information to Tuning Techniques

- Tuning Map
- Tuning Technique List

■ Scalar Tuning

- Improvement in Data Access Wait (Improvement in Thrashing)
- Improvement in Data Access Wait (Increase in Data Locality)
- Improvement in Data Access Wait (Latency Concealment)
- Improvement in Data Access Wait (Reduced Amount of Access)
- Improvement in Operation Wait (Instruction Scheduling Improvement)

■ Thread Parallelization Processing Tuning

- Thread Parallelization ratio Improvement
- Execution Efficiency Improvement of Thread Parallelization Processing

CPU Tuning

- What Is CPU Tuning (Intra-Node Tuning)?
- Positioning of CPU Tuning

What Is CPU Tuning (Intra-Node Tuning)?

CPU tuning (Intra-Node tuning) improves execution efficiency on a multi-core CPU.

The types of CPU tuning are scalar tuning and thread parallelization processing tuning.

Their various approaches to improvement include source tuning, optimization control line tuning, and compiler options tuning.

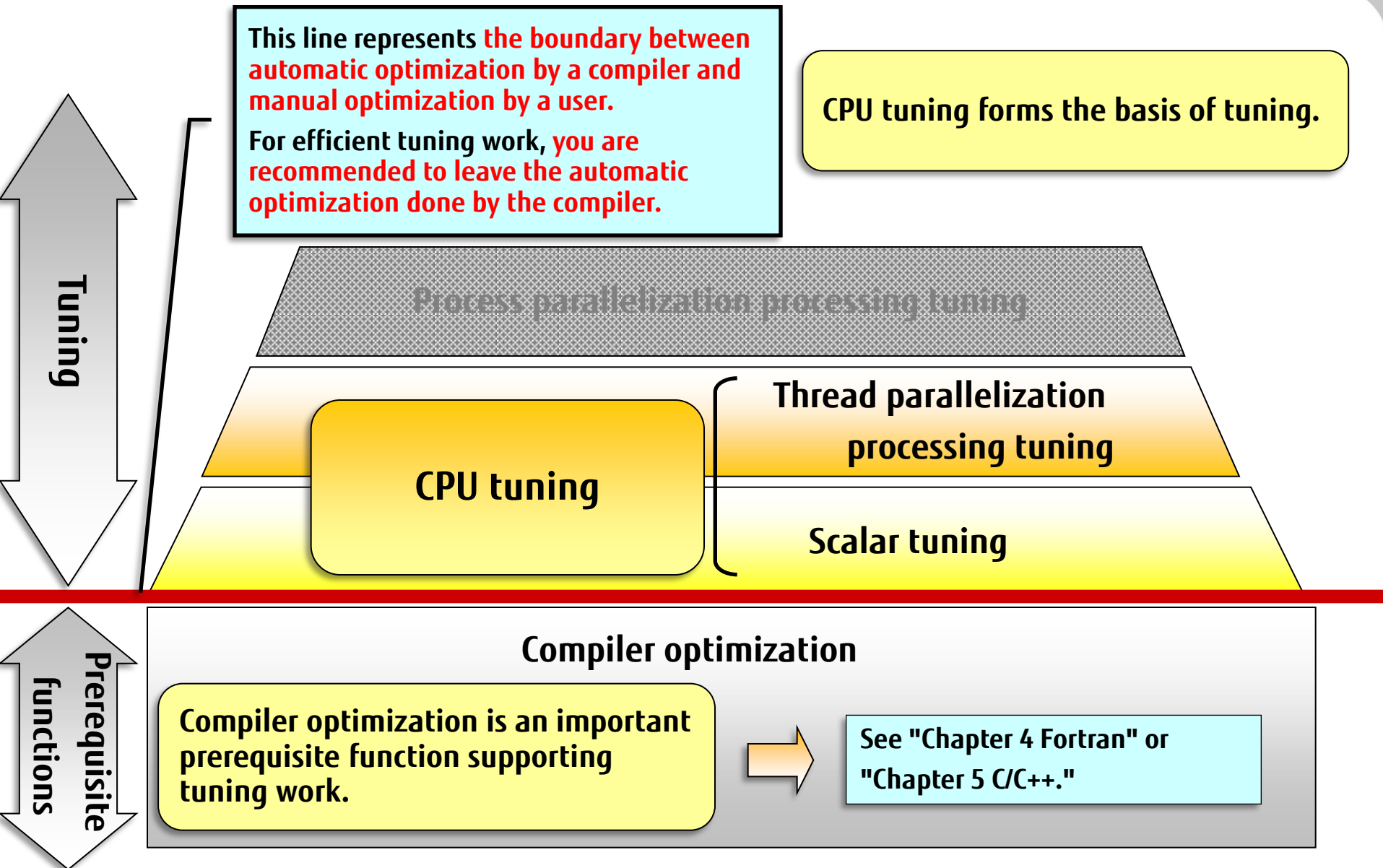
■ Scalar tuning

This tuning improves execution efficiency on a multi-core CPU by focusing attention on the cores.

■ Thread parallelization processing tuning

This tuning improves the thread parallelization ratio and execution efficiency of thread parallelization processing on a multi-core CPU.

Positioning of CPU Tuning



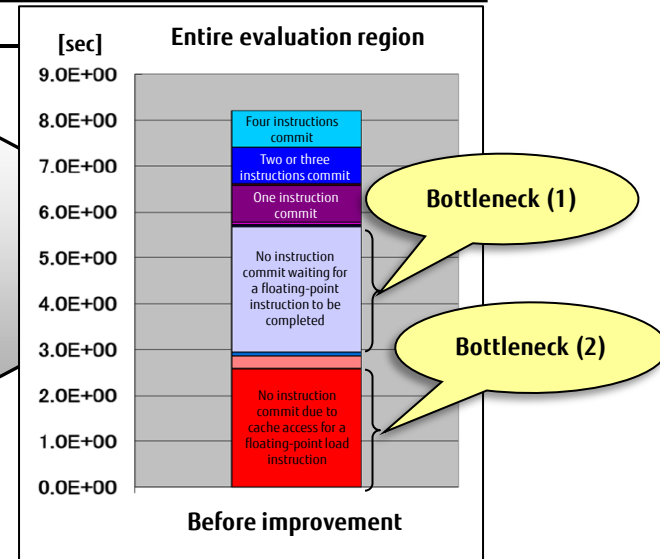
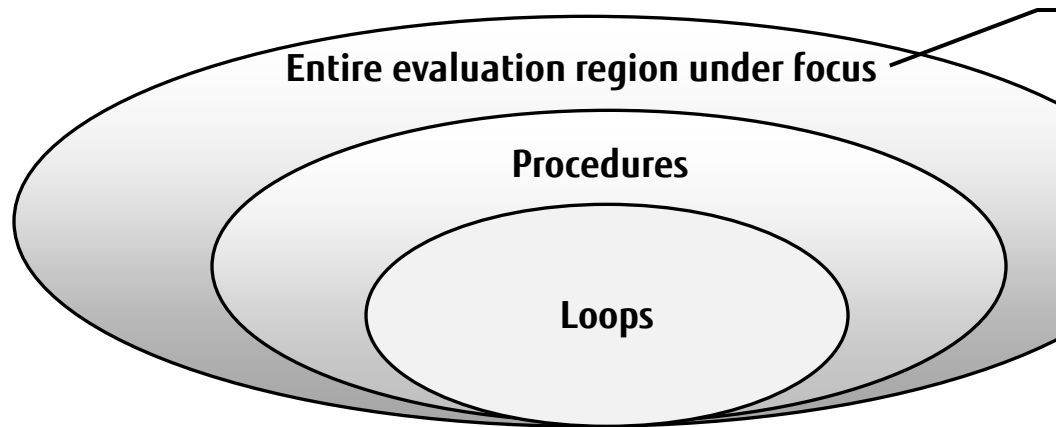
How to Effectively Use PA Information and Tuning Flows

- How to Effectively Use PA Information
- Tuning Flow
 1. Hot Spot Detection
 2. PA Information Collection
 3. Breakdown to the Level of Hot Spots
 4. Analysis and Diagnosis: Hot Spot (1)
 5. Measures and Effects: Hot Spot (1)

How to Effectively Use PA Information

■ Understanding bottlenecks

You can determine bottlenecks in the entire evaluation region under focus (except input/output and communication), from PA information for the entire evaluation region.



■ Effective use for tuning

What measures are required for improvement in bottlenecks?
To what extent can bottlenecks be improved? To answer to these questions,
PA information must be broken down to the level of loops and analyzed.

1. Hot Spot Detection

First, detect hot spots in the evaluation region under focus.

To detect hot spots, use the sampling region specification function of fipp.

■ What is the sampling region specification function?

You can collect cost information for the specified region by using the sampling region specification function. To specify a measurement section in the source code, insert C or C++ functions or Fortran subroutines at the start and end points of cost information measurement.

Function name	Function
fipp_start	Measurement start
fipp_stop	Measurement end

■ Insertion diagram



- * If the evaluation region under focus is the entire program, the sampling region specification function is not needed.
- * For details on the sampling region specification function, see the tutorial in "Chapter 7 Tuning Tool."

2. PA Information Collection

Here, collect PA information for detected hot spots. Use the advanced profiler routines of fapp (precision PA) because analysis requires highly precise PA information.

■ Advanced profiler routines (precision PA)

The routines are C and C++ functions and Fortran subroutines for specifying a measurement section for PA information. By specifying a collection section in the source code, you can collect highly precise information.

■ Insertion diagram

Function name	Function
start_collection	Information measurement start
stop_collection	Information measurement end

Entire evaluation region

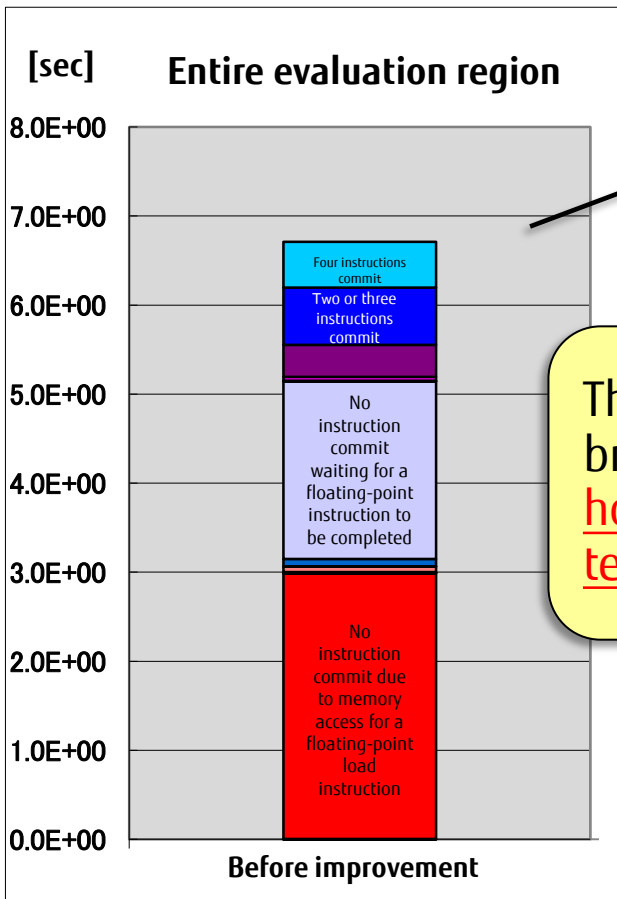
```
call start_collection("region_all")
call start_collection("region_1")
Hot spot (1)
call stop_collection("region_1")
call start_collection("region_2")
Hot spot (2)
call stop_collection("region_2")
call start_collection("region_3")
Hot spot (3)
call stop_collection("region_3")
call start_collection("region_4")
Hot spot (4)
call stop_collection("region_4")
call stop_collection("region_all")
```

"Each hot spot" enclosed by advanced profiler routine functions

3. Breakdown to the Level of Hot Spots

The graph makes it possible to understand the degree of a bottleneck.

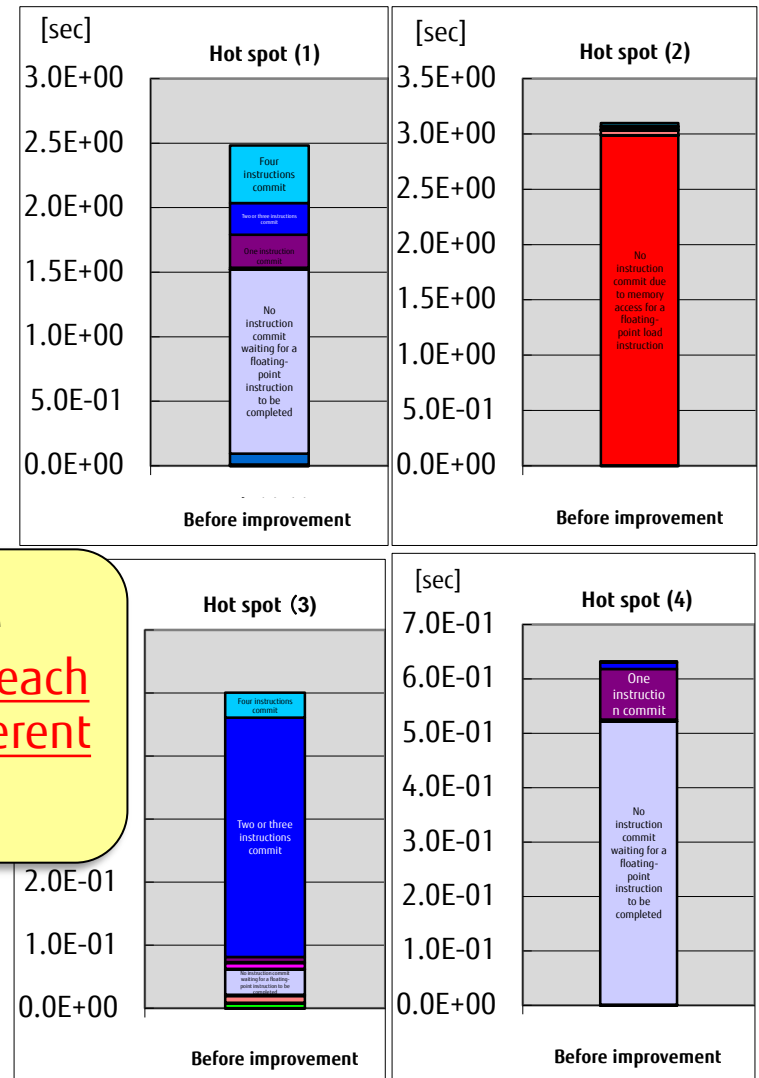
■ PA graph of the entire evaluation region



Breakdown to level of hot spots

The CPU status after the breakdown proves that each hot spot (loop) has different tendencies!

■ PA graphs of hot spots (1) to (4)



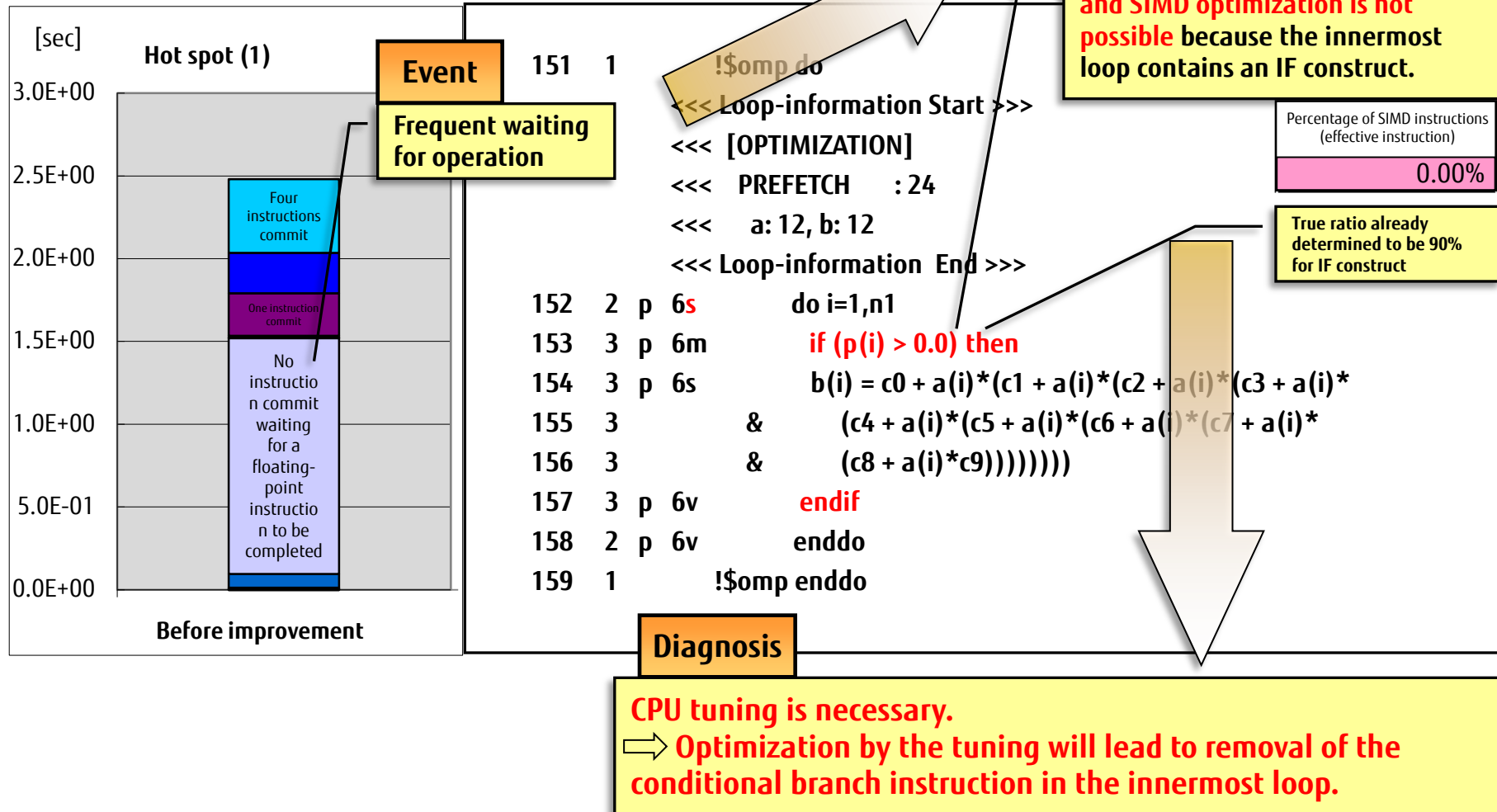
From entire evaluation region to level of hot spots

4. Analysis and Diagnosis: Hot Spot (1)

■ IF construct in the innermost loop

■ PA graph

■ Source list



5. Measures and Effects: Hot Spot (1)

■ IF construct in the innermost loop

Measure

Source list

```

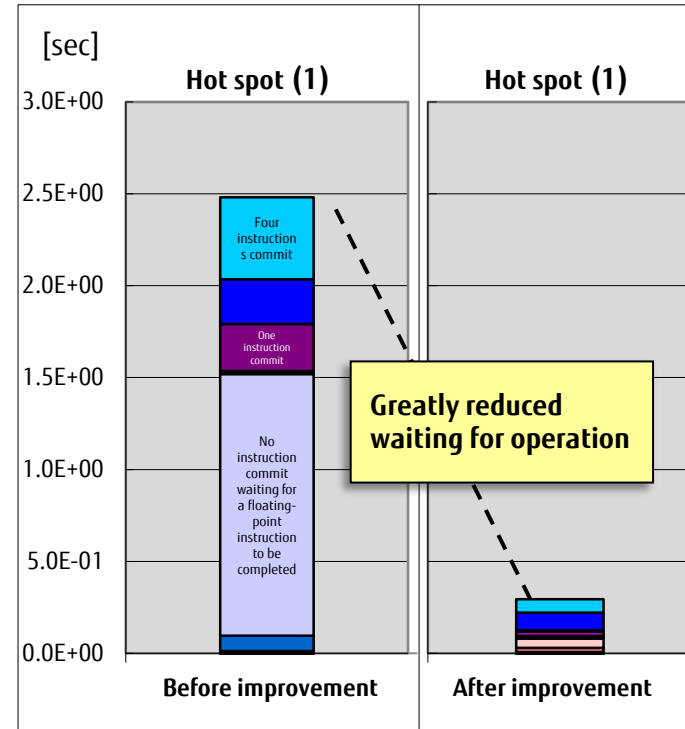
151  1  !$omp do
152  1  !ocl simd
      <<< Loop-information Start
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< SOFTWARE PIPELINING
      <<< PREFETCH    : 12
      <<< b: 12
      <<< Loop-information End >>>
153  2  p 6v      do i=1,n1
154  3  p 6v      if (p(i) > 0.0) then
155  3  p 6v          b(i) = c0 + a(i)*(c1 + a(i)*(c2 + a(i)*(c3 + a(i)*
156  3          &      (c4 + a(i)*(c5 + a(i)*(c6 + a(i)*(c7 + a(i)*
157  3          &      (c8 + a(i)*c9)))))))))
158  3  p 6v      endif
159  2  p 6v      enddo
160  1  !$omp enddo
    
```

Utilization of masked instructions
Apply SIMD optimization by using the mask method since the true ratio of the IF construct is high at 90%. This measure is also intended to facilitate software pipelining.

True ratio of 90%
for IF construct

The specification of !ocl simd is equivalent to specifying the compiler options -Ksimd=2. The SIMD optimization uses the mask method.

PA graph



Effect

Performance increased by 8.55 times

	Execution time (sec)	Floating-point operation peak rate	SIMD instruction rate (effective instruction)	Total number of valid operations
Before improvement	2.48	5.93%	0.00%	9.46E+10
After improvement	0.29	55.58%	87.71%	1.89E+10

Analysis and Tuning of Each Hot Spot

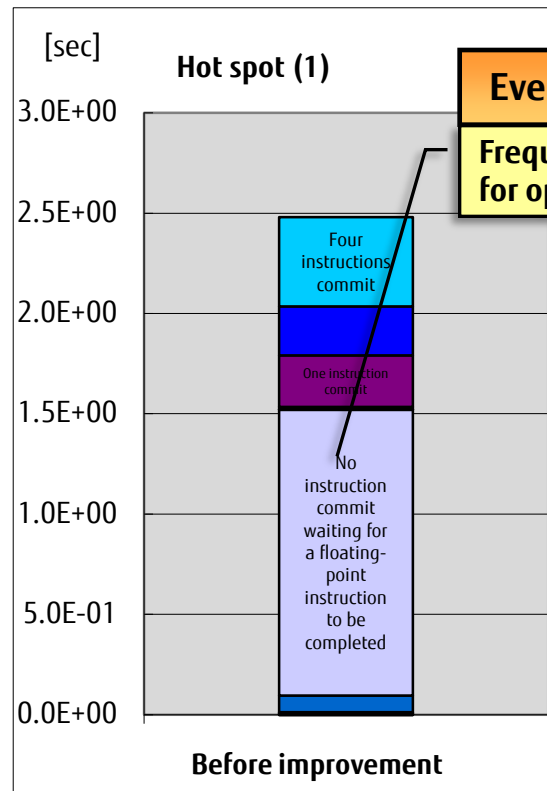
- (Duplicate) Hot Spot (1): IF Construct in the Innermost Loop (Analysis and Diagnosis)
- (Duplicate) Hot Spot (1): IF Construct in the Innermost Loop (Measures and Effects)
- Hot Spot (2): Stride Access (Analysis and Diagnosis)
- Hot Spot (2): Stride Access (Measures and Effects)
- Hot Spot (3): Ideal Operation (Analysis and Diagnosis)
- Hot Spot (4): Data Dependency (Analysis and Diagnosis)
- Entire Evaluation Region (Measures and Effects)
- Summary

Hot Spot (1): IF Construct in the Innermost Loop (Analysis and Diagnosis)

IF construct in the innermost loop

PA graph

Source list



Event 151 1

Frequent waiting for operation

```

!$omp do
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< PREFETCH :24
<<< a: 12, b: 12
<<< Loop-information End >>>
152 2 p 6s      do i=1,n1
153 3 p 6m      if (p(i) > 0.0) then
154 3 p 6s      b(i) = c0 + a(i)*(c1 + a(i)*(c2 + a(i)*(c3 + a(i)*
155 3           & (c4 + a(i)*(c5 + a(i)*(c6 + a(i)*(c7 + a(i)*
156 3           & (c8 + a(i)*c9)))))))))
157 3 p 6v      endif
158 2 p 6v      enddo
159 1           !$omp enddo
    
```

Analysis

Facilitation of software pipelining and SIMD optimization is not possible because the innermost loop contains an IF construct.

SIMD instruction rate
(effective instruction)

0.00%

True ratio of 90%
for IF construct

Diagnosis

CPU tuning is necessary.

⇒ Optimization by the tuning will lead to removal of the conditional branch instruction in the innermost loop.

Hot Spot (1): IF Construct in the Innermost Loop (Measures and Effects)

IF construct in the innermost loop

Measure

Source list

```

151  1  !$omp do
152  1  !ocl simd
      <<< Loop-information Start
      <<< [OPTIMIZATION]
      <<< SIMD (VL: 4)
      <<< SOFTWARE PIPELINING
      <<< PREFETCH    : 12
      <<< b: 12
      <<< Loop-information End >>>
153  2  p  6v    do i=1,n1
154  3  p  6v    if (p(i) > 0.0) then
155  3  p  6v      b(i) = c0 + a(i)*(c1 + a(i)*(c2 + a(i)*(c3 + a(i)*
156  3          &    (c4 + a(i)*(c5 + a(i)*(c6 + a(i)*(c7 + a(i)*
157  3          &    (c8 + a(i)*c9))))))
158  3  p  6v    endif
159  2  p  6v    enddo
160  1  !$omp enddo
    
```

True ratio of 90%
for IF construct

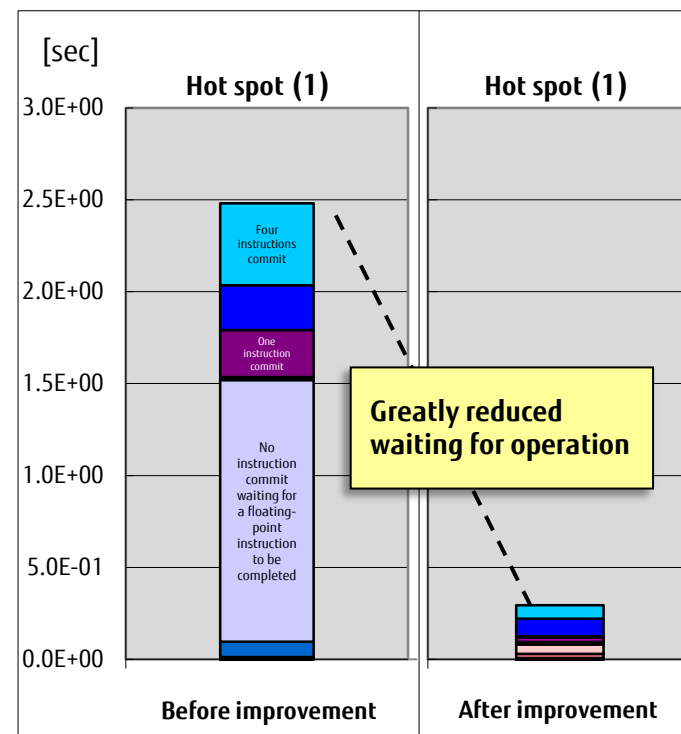
Utilization of masked instructions
Apply SIMD optimization by using the mask method since the true ratio of the IF construct is high at 90%. This measure is also intended to facilitate software pipelining.

The specification of !ocl simd is equivalent to specifying the compiler options -Ksimd=2. The SIMD optimization uses the mask method.

Effect

**Performance increased
by 8.55 times**

PA graph



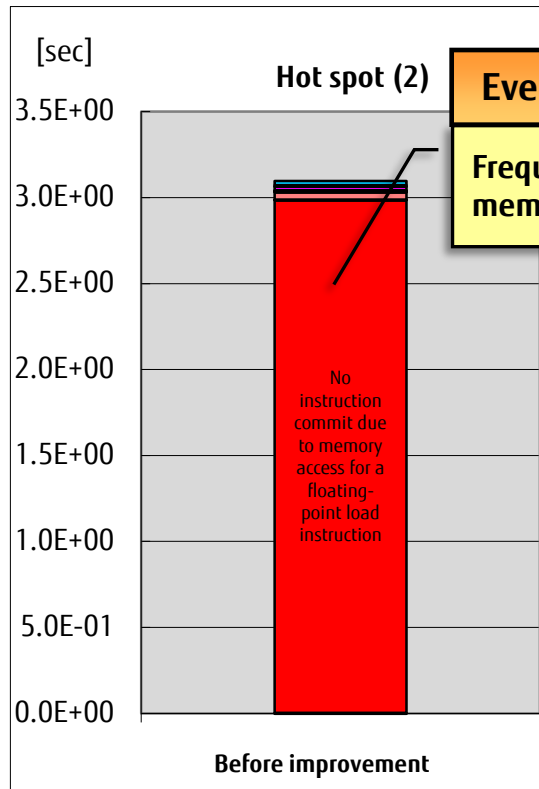
	Execution time (sec)	Floating-point operation peak rate	SIMD instruction rate (effective instruction)	Total number of valid operations
Before improvement	2.48	5.93%	0.00%	9.46E+10
After improvement	0.29	55.58%	87.71%	1.89E+10

Hot Spot (2): Stride Access (Analysis and Diagnosis)

Stride access

PA graph

Source list



Event

Frequent waiting for memory access

176 1

177 2

178 3

179 3

180 3

181 3

182 3

183 2

184 1

!\$omp do

do j=1,n2

<<< Loop-information Start >>>

<<< [OPTIMIZATION]

<<< SIMD(VL: 4)

<<< SOFTWARE PIPELINING

<<< Loop-information End >>>

do i=1,n1

$b(i,j) = c0 + a(j,i) * (c1 + a(j,i) * (c2 + a(j,i) * (c3 + a(j,i) * (c4 + a(j,i) * (c5 + a(j,i) * (c6 + a(j,i) * (c7 + a(j,i) * (c8 + a(j,i) * c9))))))))$

&

&

enddo

enddo

!\$omp enddo

Analysis

The cache use efficiency is low since array b is accessed contiguously and **access to array a is stride access**. The result is a throughput bottleneck.

L1D miss rate (/Load-store instruction)	L2 miss rate (/Load-store instruction)
51.31%	51.34%

Diagnosis

CPU tuning is necessary.

⇒ **Improve the cache use efficiency of array a.**

Hot Spot (2): Stride Access (Measures and Effects)

Stride access

Source list

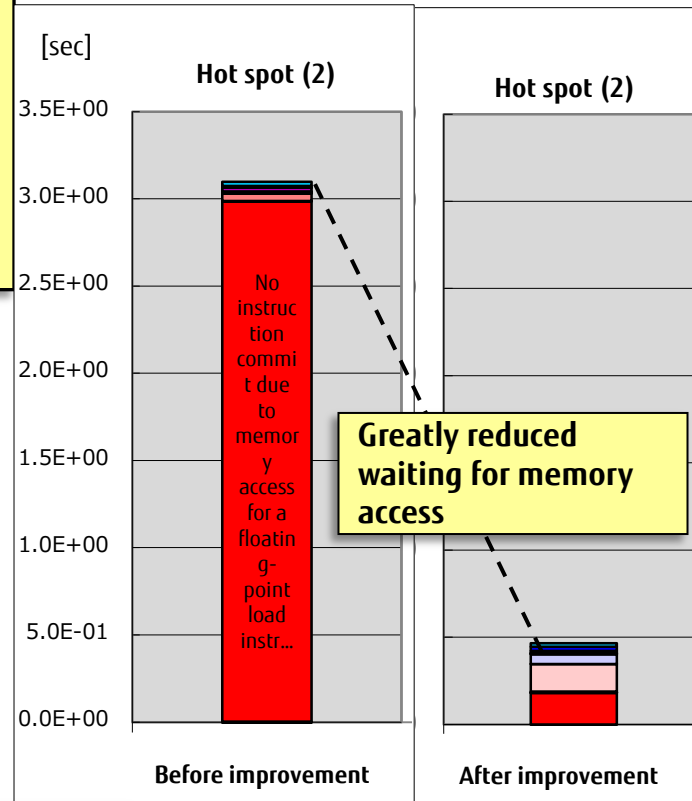
```

177 1      !$omp do
178 2 p      do jj=1,n2,16
179 3 p      do ii=1,n1,96
180 4 p      do j=jj,min(jj+16-1,
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< SIMD(VL: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End >>>
181 5 p 6v      do i=ii,min(ii+96-1,n1)
182 5 p 6v      b(i,j) = c0 + a(j,i)*(c1 + a(j,i)*(c2 + a(j,i)*(c3 + a(j,i)*
183 5          & (c4 + a(j,i)*(c5 + a(j,i)*(c6 + a(j,i)*(c7 + a(j,i)*
184 5          & (c8 + a(j,i)*c9))))))
185 5 p 6v      enddo
186 4 p      enddo
187 3 p      enddo
188 2 p      enddo
189 1      !$omp enddo
    
```

Measure

Application of loop blocking
 The size of one block is 12 KB (96 x 16 x 8). The memory area size required for processing one block is 24 KB (12 x 2).
 The measure is intended to improve the L1D and L2 cache use efficiency.

PA graph



Effect

Performance increased by 6.6 times

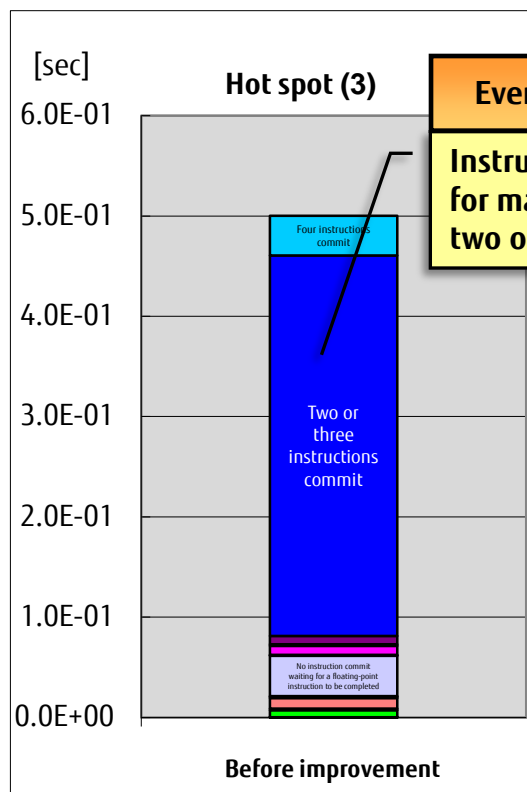
	Execution time (sec)	Floating-point operation peak rate	L1D miss rate(/Load-store instruction)	L2 miss rate(/Load-store instruction)
Before improvement	3.10	1.28%	51.31%	51.34%
After improvement	0.47	8.54%	6.66%	6.12%

Hot Spot (3): Ideal Operation (Analysis and Diagnosis)

■ Ideal operation

■ PA graph

■ Source list



Event 201 1 !\$omp do

Instruction commit accounting for majority, with many having two or more instructions

Information Start >>>

IMIZATION]

D(VL: 4)

<<< SOFTWARE PIPELINING

<<< Loop-information End >>>

```

202 2 p 6v do i=1,n1
203 2 p 6v b(i) = c0 + a(i)*(c1 + a(i)*(c2 + a(i)*(c3 + a(i)*
204 2      & (c4 + a(i)*(c5 + a(i)*(c6 + a(i)*(c7 + a(i)*
205 2      & (c8 + a(i)*c9))))))
206 2 p 6v enddo
207 1 !$omp enddo
    
```

Analysis

Both the SIMD instruction rate and percentage of SIMD multiply-add operation instructions are high. The **operation peak ratio** also shows a very high performance value of **78%**.

SIMD instruction rate (effective instruction)	SIMD floating-point multiply-and-add instruction rate (effective instruction)	Floating-point operation peak rate
95.31%	77.95%	78.33%

Diagnosis

CPU tuning is not necessary.

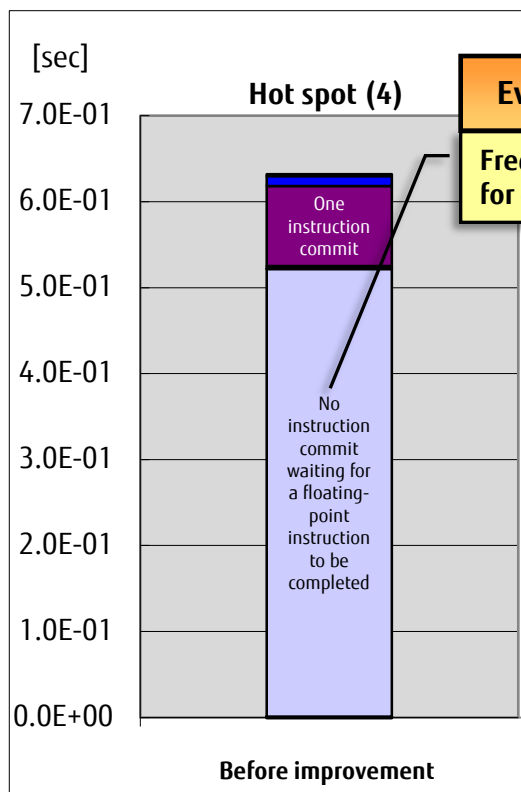
⇒ **Instruction-level parallelization can reach very high levels.**

Hot Spot (4): Data Dependency (Analysis and Diagnosis)

■ Data Dependency

■ PA graph

■ Source list



Event

Frequent waiting for operation

```
225 2 p 6s
226 2 p 6s
227 2
228 2
229 2 p 6s
```

```
do i=2,n1
  a(i) = c0 + a(i-1)*(c1 + a(i-1)*(c2 + a(i-1)*(c3 + a(i-1)*
    & (c4 + a(i-1)*(c5 + a(i-1)*(c6 + a(i-1)*(c7 + a(i-1)*
    & (c8 + a(i-1)*c9))))))))
enddo
```

Analysis

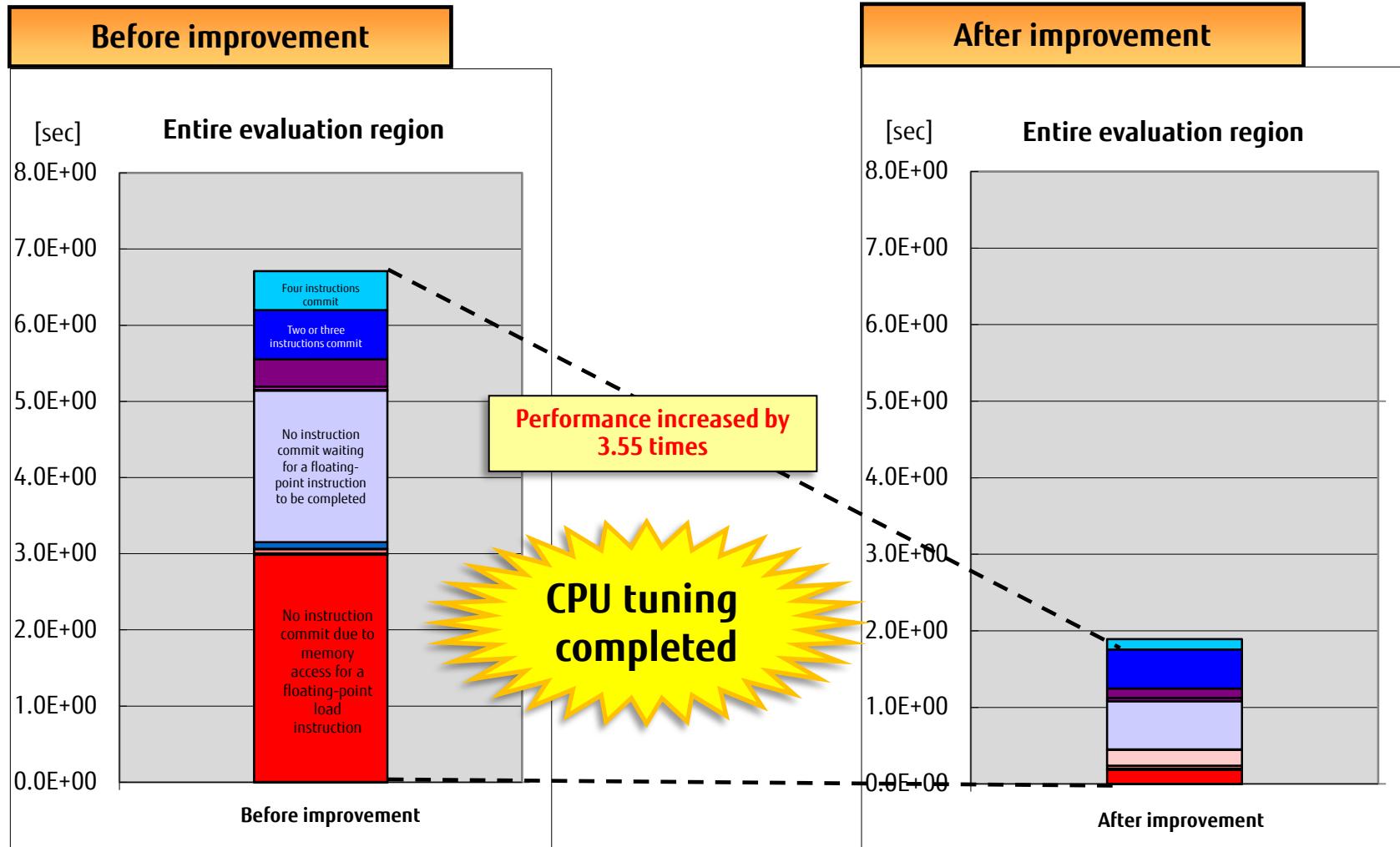
Neither facilitation of software pipelining and SIMD optimization nor parallelization is possible because the processing shown as $a(i)=a(i-1)$ for array a **causes data dependency between iterations.**

Diagnosis

CPU tuning is impossible.
⇒ The operation algorithm must be reviewed.

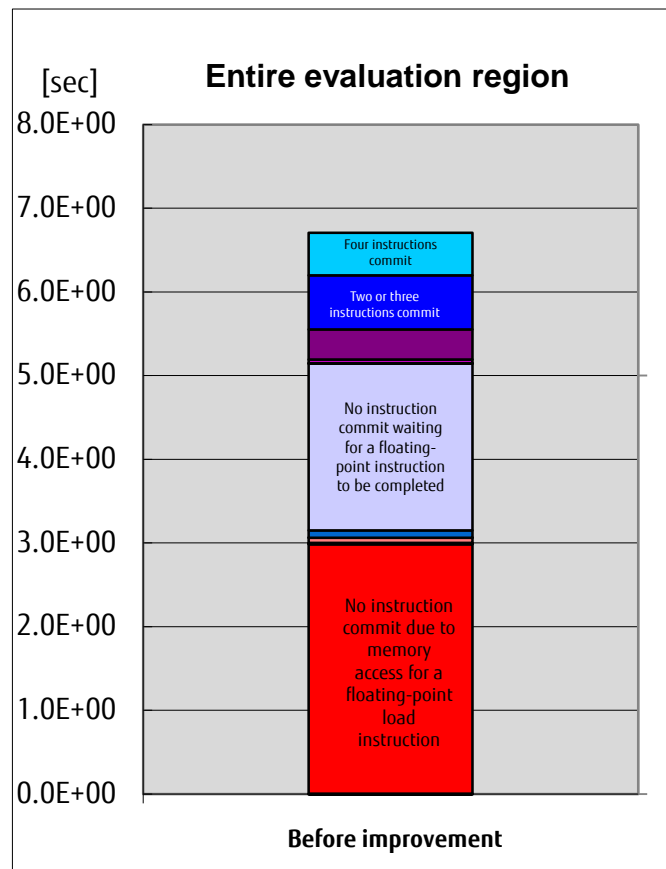
Entire Evaluation Region (Measures and Effects)

PA graph

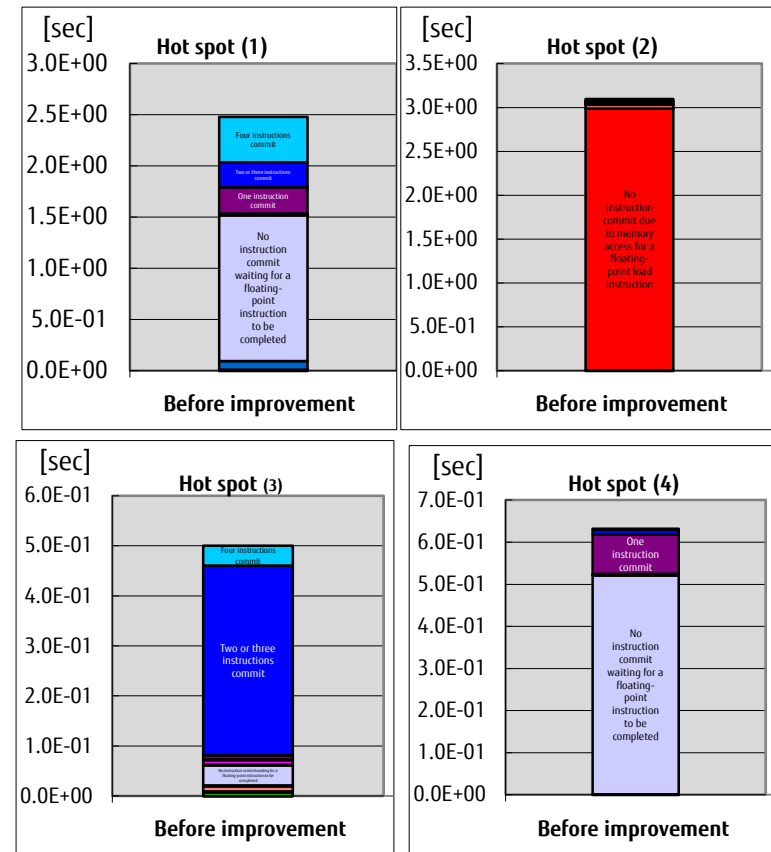


Summary

- You can determine bottlenecks from the PA graph of an entire evaluation region.
- The bottleneck factors are often different for every loop. For this reason, a breakdown to the level of loops is necessary to analyzing and determining whether CPU tuning is possible and how to take measures for problems.



Breakdown



Navigation from PA Information to Tuning Techniques

- Tuning Map
- Tuning Technique List

- The **tuning map** is useful for determining a specific **tuning method** from PA information.
- Tuning map
 - The tuning map is a list showing tuning proposals by bottleneck type. The list clearly shows what PA information to check and bottleneck factors (conditions) that occur by bottleneck classification, and summarizes the measures (tuning proposals: what to improve) for solving them.
 - ⇒ 1. Identify bottleneck factors from PA information.
 - ⇒ 2. Present measures (tuning proposals) for removing bottlenecks.
- Tuning technique list
 - This list summarizes various tuning techniques by tuning proposal.
 - ⇒ Select an effective tuning technique for improvement.
- For examples of actual measures, see the scalar tuning examples.

Tuning Map (1/12)

■ Entire tuning map

Bottleneck classification	High cost as seen from PA graph		High cost as seen from PA information	Condition	Tuning proposal
Memory bottleneck	No instruction commit due to memory access for a floating-point load instruction		-	Memory latency is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Prefetch-related improvement
	No instruction commit due to memory access for an integer load instruction		-	Memory latency is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Prefetch-related improvement
	No instruction commit because SP (store port) is full		-	The store instruction cost is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Prefetch-related improvement - High-speed store (XFILL)
	No instruction commit due to memory and cache busy		-	Memory throughput is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Loop blocking - High-speed store (XFILL)
	-		High memory busy rate	Memory throughput is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Loop blocking - High-speed store (XFILL)
	-		High percentage of L2 misses High percentage of L2 misses due to dm	Memory latency is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Loop blocking - Prefetch-related improvement - Thrashing
L2 cache bottleneck	No instruction commit due to L2 access for a floating-point load instruction		-	L2 cache latency is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Prefetch-related improvement
	No instruction commit due to L2 access for an integer load instruction		-	L2 cache latency is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Prefetch-related improvement
	-		High L2 busy rate	L2 cache throughput is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Loop blocking
	-		High percentage of L1D misses High percentage of L1D misses due to dm	L2 cache latency is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Thrashing
L1 cache bottleneck	No instruction commit due to L1D access for a floating-point load instruction		-	L1 cache latency is a bottleneck.	Instruction scheduling improvement
	No instruction commit due to L1D access for an integer load instruction		-	L1 cache latency is a bottleneck.	Instruction scheduling improvement
	-		High L1 busy rate	L1 cache throughput is a bottleneck.	Improvement in data access wait - Algorithm review
Scheduling bottleneck	No instruction commit waiting for a floating-point instruction to be completed		-	Operation instruction latency is a bottleneck.	Instruction scheduling improvement
	No instruction commit waiting for an integer instruction to be completed		-	Operation instruction latency is a bottleneck.	Instruction scheduling improvement
	No instruction commit waiting for a branch instruction to be completed		-	A branch instruction is a bottleneck.	Instruction scheduling improvement - IF statement removal - Masked SIMD
Parallelization bottleneck	Synchronous waiting time between threads		-	A part that is not thread parallelization is a bottleneck.	Thread parallelization ratio improvement
Load imbalance bottleneck	Synchronous waiting time between threads		Large difference in the instruction balance between max and min	A load imbalance between threads is a bottleneck.	Execution efficiency improvement of thread parallelization processing
TLB bottleneck	-		High percentage of mDTLB misses	TLB misses and TLB thrashing are a bottleneck.	Improvement in the TLB bottleneck - Elimination of thrashing - Change of areas used - Optimization using large page options
	-		High percentage of uDTLB misses	TLB misses are a bottleneck.	Improvement in the TLB bottleneck - Page size expansion
Instruction fetch	No instruction commit waiting for an instruction to be fetched		-	Instruction cache misses and thrashing are a bottleneck.	Improvement in instruction fetch - Reduction in the loop body - Algorithm review - Elimination of thrashing
Instruction count bottleneck	Instruction commit	Four instructions commit	-	The number of instructions is a bottleneck.	Improvement in the instruction count bottleneck - Facilitation of SIMD optimization - Prefetch-related improvement - Inline expansion
		Two or three instructions commit			
		One instruction commit			
Other	No instruction commit for other reasons		-	PA may have not been collected correctly.	PA re-collection

Tuning Map (2/12)

■ Bottleneck classifications

Left: Bottleneck classifications
Right: Costs as seen from PA graph

Memory bottleneck	No instruction commit due to memory access for a floating-point load instruction	
	No instruction commit due to memory access for an integer load instruction	
	No instruction commit because SP (store port) is full	
	No instruction commit due to memory and cache busy	
L2 cache bottleneck	No instruction commit due to L2 access for a floating-point load instruction	
	No instruction commit due to L2 access for an integer load instruction	
L1 cache bottleneck	No instruction commit due to L1D access for a floating-point load instruction	
	No instruction commit due to L1D access for an integer load instruction	
Scheduling bottleneck	No instruction commit waiting for a floating-point instruction to be completed	
	No instruction commit waiting for an integer instruction to be completed	
	No instruction commit waiting for a branch instruction to be completed	
Parallelization bottleneck	Synchronous waiting time between threads	
Load imbalance bottleneck	Synchronous waiting time between threads	
TLB bottleneck	-	
Instruction fetch	No instruction commit waiting for an instruction to be fetched	
Instruction count bottleneck	Instruction commit	Four instructions commit
		Two or three instructions commit
		One instruction commit
Other	No instruction commit for other reasons	

■ Memory bottleneck

High cost as seen from PA graph	High cost as seen from PA information	Condition	Tuning proposal
No instruction commit due to memory access for a floating-point load instruction	-	Memory latency is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Prefetch-related improvement
No instruction commit due to memory access for an integer load instruction	-	Memory latency is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Prefetch-related improvement
No instruction commit because SP (store port) is full	-	The store instruction cost is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Prefetch-related improvement - High-speed store (XFILL)
No instruction commit due to memory and cache busy	-	Memory throughput is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Loop blocking - High-speed store (XFILL)
-	High memory busy rate	Memory throughput is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Loop blocking - High-speed store (XFILL)
-	High percentage of L2 misses High percentage of L2 misses due to dm	Memory latency is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Loop blocking - Prefetch-related improvement - Thrashing

■ L2 cache bottleneck

High cost as seen from PA graph	High cost as seen from PA information	Condition	Tuning proposal
No instruction commit due to L2 access for a floating-point load instruction	-	L2 cache latency is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Prefetch-related improvement
No instruction commit due to L2 access for an integer load instruction	-	L2 cache latency is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Prefetch-related improvement
-	High L2 busy rate	L2 cache throughput is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Loop blocking
-	High percentage of L1D misses High percentage of L1D misses due to dm	L2 cache latency is a bottleneck.	Improvement in data access wait - Dimensional displacement of an array - Thrashing

■ L1 cache bottleneck

High cost as seen from PA graph	High cost as seen from PA information	Condition	Tuning proposal
No instruction commit due to L1D access for a floating-point load instruction	-	L1 cache latency is a bottleneck.	Instruction scheduling improvement
No instruction commit due to L1D access for an integer load instruction	-	L1 cache latency is a bottleneck.	Instruction scheduling improvement
-	High L1 busy rate	L1 cache throughput is a bottleneck.	Improvement in data access wait - Algorithm review

■ Scheduling bottleneck

High cost as seen from PA graph	High cost as seen from PA information	Condition	Tuning proposal
No instruction commit waiting for a floating-point instruction to be completed	-	Operation instruction latency is a bottleneck.	Instruction scheduling improvement
No instruction commit waiting for an integer instruction to be completed	-	Operation instruction latency is a bottleneck.	Instruction scheduling improvement
No instruction commit waiting for a branch instruction to be completed	-	A branch instruction is a bottleneck.	Instruction scheduling improvement - IF statement removal - Masked SIMD

■ Parallelization bottleneck

High cost as seen from PA graph	High cost as seen from PA information	Condition	Tuning proposal
Synchronous waiting time between threads	-	A part that is not thread parallelization is a bottleneck.	Thread parallelization ratio improvement

■ Load imbalance bottleneck

High cost as seen from PA graph	High cost as seen from PA information	Condition	Tuning proposal
Synchronous waiting time between threads	Large difference in the instruction balance between max and min	A load imbalance between threads is a bottleneck.	Execution efficiency improvement of thread parallelization processing

■ TLB bottleneck

High cost as seen from PA graph	High cost as seen from PA information	Condition	Tuning proposal
-	High percentage of mDTLB misses	TLB misses and TLB thrashing are a bottleneck.	Improvement in the TLB bottleneck <ul style="list-style-type: none">- Elimination of thrashing- Change of areas used- Optimization using large page options
-	High percentage of uDTLB misses	TLB misses are a bottleneck.	Improvement in the TLB bottleneck <ul style="list-style-type: none">- Page size expansion

■ Instruction fetch

High cost as seen from PA graph	High cost as seen from PA information	Condition	Tuning proposal
No instruction commit waiting for an instruction to be fetched	-	Instruction cache misses and thrashing are a bottleneck.	Improvement in instruction fetch <ul style="list-style-type: none">- Reduction in the loop body- Algorithm review- Elimination of thrashing

■ Instruction count bottleneck

High cost as seen from PA graph		High cost as seen from PA information	Condition	Tuning proposal
Instruction commit	Four instructions commit	-	The number of instructions is a bottleneck.	Improvement in the instruction count bottleneck <ul style="list-style-type: none">- Facilitation of SIMD optimization- Prefetch-related improvement- Inline expansion
	Two or three instructions commit			
	One instruction commit			

■ Other

High cost as seen from PA graph	High cost as seen from PA information	Condition	Tuning proposal
No instruction commit for other reasons	-	PA may have not been collected correctly.	PA re-collection

Tuning Technique List (1/2)

■ Major classifications

Thread parallelization ratio improvement	Execution efficiency improvement of thread parallelization processing	Improvement in data access wait	Instruction scheduling improvement	Improvement in the TLB bottleneck	Improvement in instruction fetch	Improvement in the instruction count bottleneck
NORECURRENCE specifier (Facilitation of automatic parallelization)	False sharing	Dimensional displacement of an array	Software pipelining	Page size expansion	Reduction in the loop body	Facilitation of SIMD optimization
NOALIAS specifier (Facilitation of automatic parallelization)	Parallelized dimension change	Prefetch-related improvement	Unrolling	Elimination of thrashing	Algorithm review (Expansion of the problem scale)	Prefetch-related improvement
Peeling (Facilitation of automatic parallelization)	Division method change (Cyclic)	High-speed store (XFILL)	Facilitation of SIMD optimization	Change of areas used	Elimination of thrashing	Inline expansion
OpenMP parallelization	Division method change (Dynamic)	Loop blocking	IF statement removal	Optimization using large page options		
	Parallelization algorithm review	Elimination of thrashing	Masked SIMD			
		Algorithm review (Reducing the memory access instruction ratio)	Outer unrolling			
		Array division	Suppression of software pipelining & specification of the number of unrollings (Loop with a few iterations)			
		Loop fission	Rerolling			
		Strip mining	Peeling			
		Sector cache	NORECURRENCE specifier			
		Loop interchange	NOALIAS specifier			
		Loop fusion				
		Array merging				

* The colored items represent medium classifications. If applicable, go to the next page.

Tuning Technique List (2/2)

■ Medium classifications

Prefetch-related improvement	Facilitation of SIMD optimization	Elimination of thrashing	Reduction in the loop body
Addition of prefetching	Changing arrays to simple variables	Padding	Suppression of software pipelining
Deletion of unnecessary prefetching	Loop unswitching	Dimensional displacement of an array	Suppression of unrolling
Prefetching toward the outer loop	IF statement removal	Array merging	Suppression of loop fusion
Indirect access prefetching	Rerolling	Reduction in the loop body	Loop fission
	Inline expansion		
	Loop fission (separating dependent accesses)		
	Loop fission (loop extraction) for a part with a high true ratio		
	Cloning		
	NORECURRENCE specifier		
	NOALIAS specifier		

Scalar Tuning

- Improvement in Data Access Wait (Improvement in Thrashing)
- Improvement in Data Access Wait (Increase in Data Locality)
- Improvement in Data Access Wait (Latency Concealment)
- Improvement in Data Access Wait (Reduced Amount of Access)
- Improvement in Operation Wait (Instruction Scheduling Improvement)

Improvement in Data Access Wait (Improvement in Thrashing)

- Improvement in Cache Thrashing
- Improvement in TLB Thrashing

Improvement in Cache Thrashing

- What Is Cache Thrashing?
- Tuning Approach to Cache Thrashing (Basics)
- Tuning Approach to Cache Thrashing (Application)

What Is Cache Thrashing?

Cache thrashing is a phenomenon in which only data of specific indexes of cache memory (location information in cache memory) is frequently overwritten. This phenomenon is due to the size of 1 WAY being 16 KB. It is likely to occur when the array size is a power of 2 (multiple of 16 KB) or when a loop contains many streams.

Note: A stream is a series of data defined in references corresponding to loop iterations.

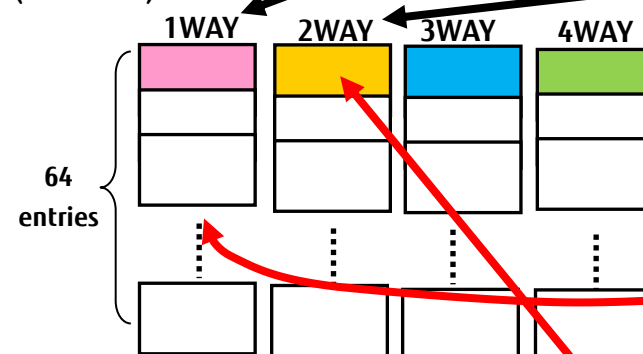
Source code example
<pre> subroutine sub(a, n, m) ※n=256, m=256 real*8 a(n,m,8) do j= 1, m do i= 1, n a(i,j,8)=a(i,j,1)+a(i,j,2)+a(i,j,3)+a(i,j,4)+ a(i,j,5)+a(i,j,6)+a(i,j,7) enddo enddo end </pre>

Rough standard for L1D cache thrashing

L1D miss rate(/Load-store instruction)	Percentage of L1D misses due to dm (relative to number of L1D misses)
Single-precision: 1.5625% or higher Double-precision: 3.125% or higher	20% or higher

Single-precision: 1/64 (one miss for every 64 times)
 Double-precision: 1/32 (one miss for every 32 times)

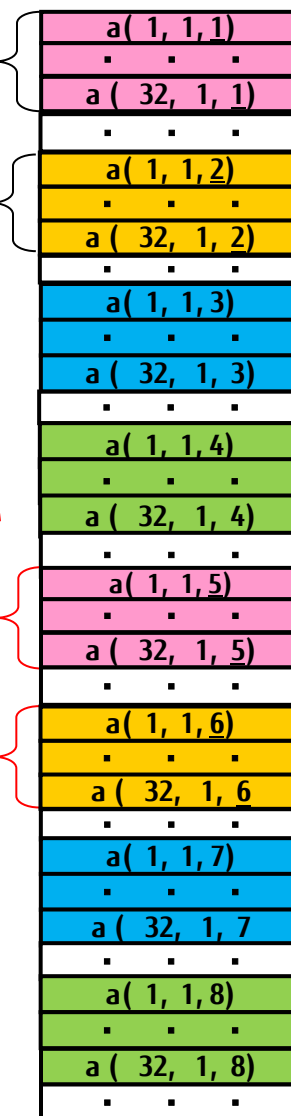
(L1D cache)



—————> Storing data in cache
 —————> Storing data in cache (conflict)
 - - - - -> Execution order (1) to (7)

In this example, $a(1,1,1)$ to $a(1,1,8)$ are placed at an interval of 32×16 KB (on a 16-KB boundary), so the eight of them are assigned to the same index. Therefore, the first and second points of data are overwritten by the fifth and sixth points of data, respectively.

(Data alignment in memory)



Execution order

- (1)
 $256 \times 256 \times 8 \text{ B}$
 $= 32 \times 16 \text{ KB}$
 Discrete access
- (2)
 $256 \times 256 \times 8 \text{ B}$
 $= 32 \times 16 \text{ KB}$
 Discrete access
- (3)
 $256 \times 256 \times 8 \text{ B}$
 $= 32 \times 16 \text{ KB}$
 Discrete access
- (4)
 $256 \times 256 \times 8 \text{ B}$
 $= 32 \times 16 \text{ KB}$
 Discrete access
- (5)
 $256 \times 256 \times 8 \text{ B}$
 $= 32 \times 16 \text{ KB}$
 Discrete access
- (6)
 $256 \times 256 \times 8 \text{ B}$
 $= 32 \times 16 \text{ KB}$
 Discrete access
- (7)
 $256 \times 256 \times 8 \text{ B}$
 $= 32 \times 16 \text{ KB}$
 Discrete access


Tuning Approach to Cache Thrashing (Basics)

- Tuning Approach (Basics)
- Array Merging
- Dimensional Displacement of an Array
- Loop Fission
- Padding

Tuning Approach (Basics)

■ Array merging


```
do i=1,n
  ... = a(i) + b(i)
enddo
```



```
do i=1,n
  ... = z(1,i) + z(2,i)
enddo
```

■ Dimensional displacement of an array


```
do i=1,n
  ... = a(l,1) + a(l,2)
enddo
```



```
do i=1,n
  ... = a(1,i) + a(2,i)
enddo
```

■ Loop fission


```
do i=1,n
  ... = a(i) + b(i)
  ... = c(i) + d(i)
enddo
```



```
do i=1,n
  ... = a(i) + b(i)
enddo
do i=1,n
  ... = c(i) + d(i)
enddo
```

■ Padding

```
common //a(n),b(n)
do i=1,n
  ... = a(i) + b(i)
enddo
```



```
common //a(n),p(64),b(n)
do i=1,n
  ... = a(i) + b(i)
enddo
```

A resolution approach improves thrashing risks themselves by reducing the number of streams.

Advantage: Can accommodate shifts in the problem scale such as parameter changes.

Disadvantage: Requires many corrections. SIMD optimization is complex.
Note: This does not pertain to loop fission.

Advantage: Corrections are few, and SIMD optimization is easy.

Disadvantage: Adjustments are necessary for every shift in the problem scale such as a parameter change.

An avoidance approach for improvement shifts array addresses through padding.

Array Merging

- What Is Array Merging?
- Array Merging (Before Improvement)
- Effects of Array Merging (Source Tuning)
- Array Merging (in C Language) (Before Improvement)
- Effects of Array Merging (in C Language) (Source Tuning)
- Effects of Array Merging (Compiler Options Tuning)

What Is Array Merging?

Array merging is tuning that merges multiple arrays into one array.

■ Use conditions

- Each array to be merged has the same number of elements.

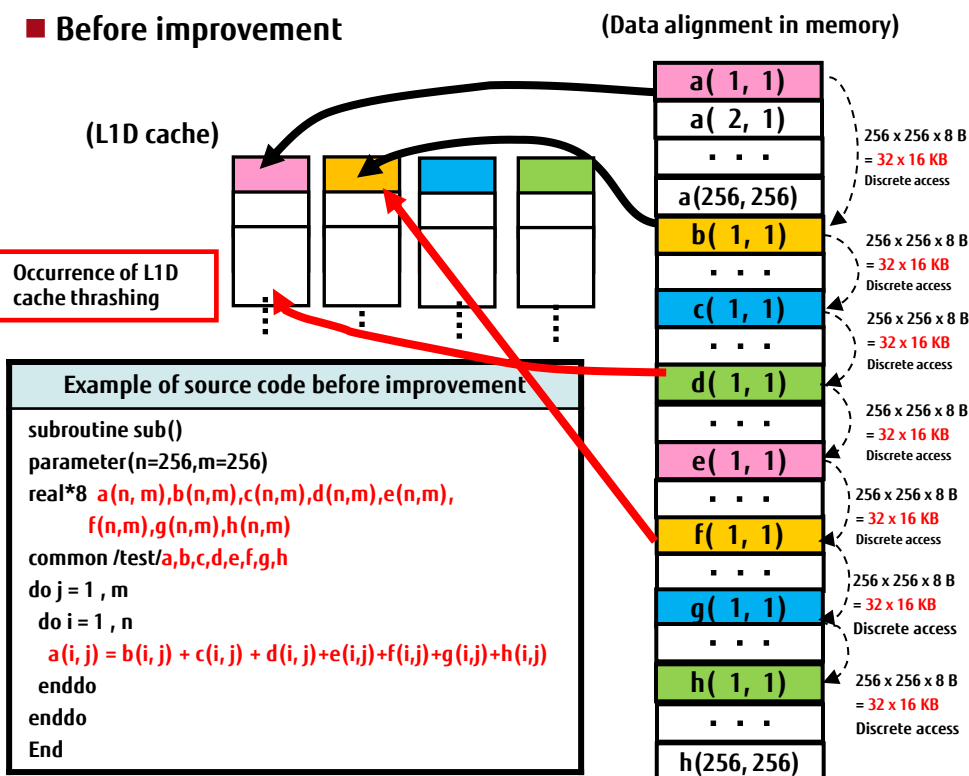
■ Purpose

- The purpose is to reduce the number of streams.

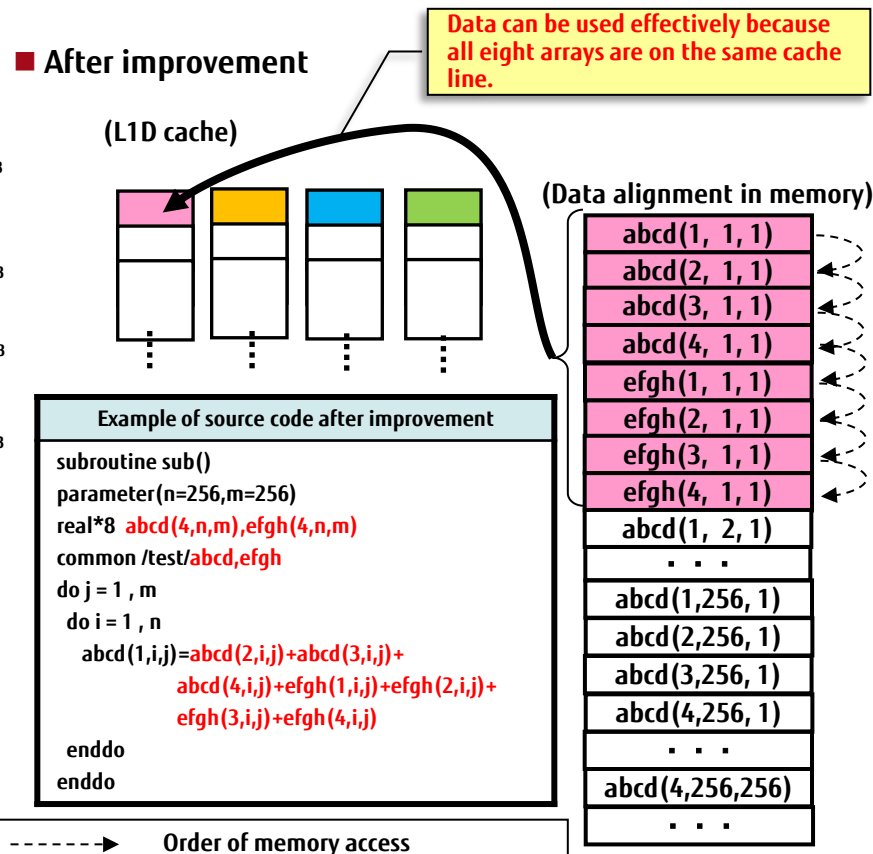
■ Adverse effect

- Load and store instructions become stride or indirect instructions.

■ Before improvement



■ After improvement



Array Merging (Before Improvement)

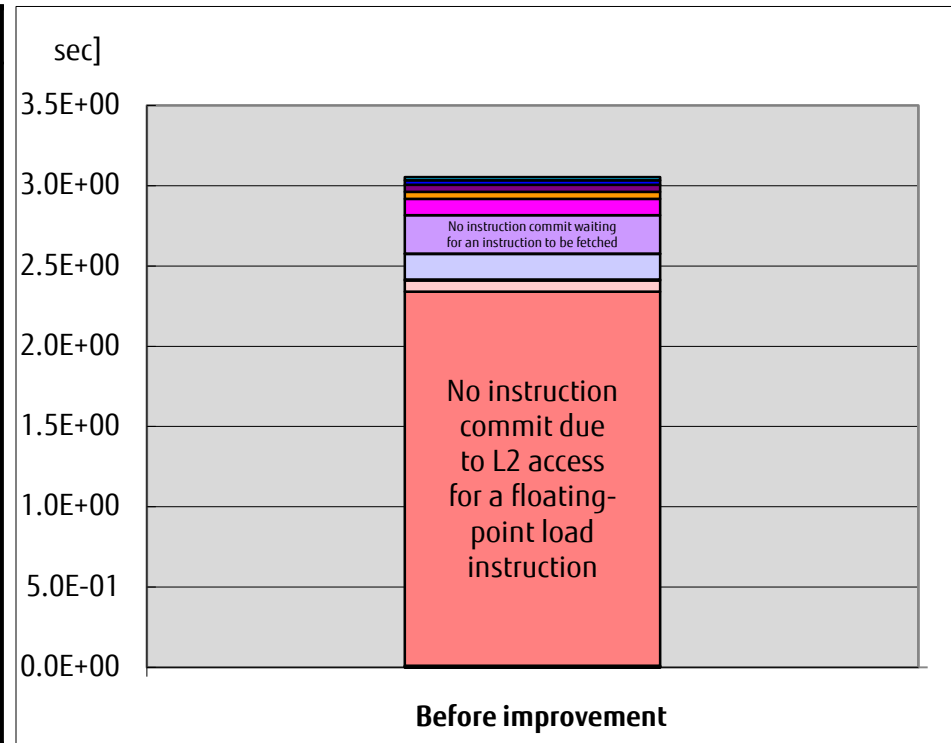
L1D cache thrashing occurs because each array is located on a 16-KB boundary. Consequently, the following is a frequent event: No instruction commit due to L2 access for a floating-point load instruction.

Source code 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: 422
    <<< [OPTIMIZATION]
    <<< COLLAPSED
    <<< SIMD(VL: 4)
    <<< SOFTWARE PIPELINING
    <<< Loop-information End >>>
43  1 pp 6v do j = 1, m
    <<< Loop-information Start >>>
    <<< [OPTIMIZATION]
    <<< COLLAPSED
    <<< Loop-information End >>>
44  2 p 6 do i = 1, n
45  2 p 6v 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 6v enddo
47  1 p enddo
    
```

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



Cache

	L1I miss rate(/Effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	Memory throughput (GB/sec)	L2 throughput (GB/sec)
Before improvement	0.00%	23.21%	3.12E+09	91.66%	8.34%	0.00%	0.00%	0.00	261.73

The percentage of L1D misses is high and the L1 miss dm percentage is high, despite the fact that the array is accessed sequentially.

⇒ **L1D cache thrashing has occurred.**

Effects of Array Merging (Source Tuning)

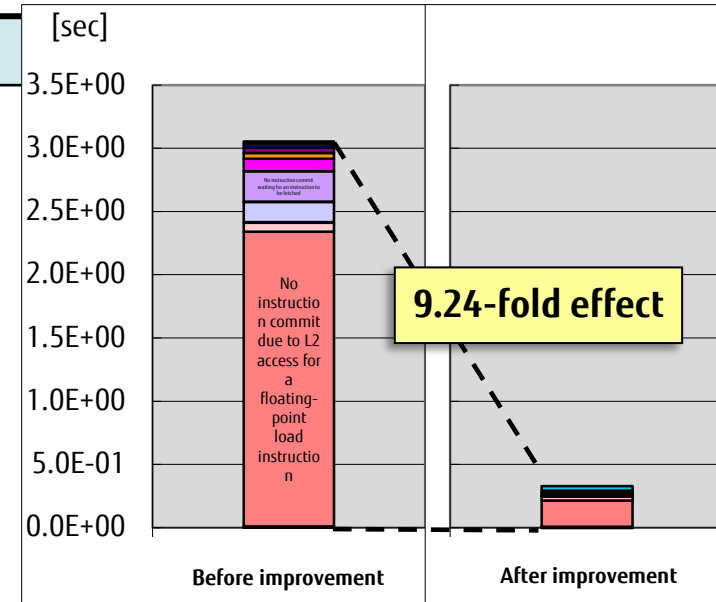
Array merging reduced the number of streams from eight to two, so L1D cache thrashing was avoided. This results in improvement of the following event: No instruction commit due to L2 access for a floating-point load instruction.

Source code after improvement (source tuning)

```

44     parameter(n=256,m=256)
45     real*8 abcd(4,n,m),efgh(4,n,m)
46     common /test/abcd,efgh
    <<< Loop-information Start >>>
    <<< [PARALLELIZATION]
    <<< Standard iteration count: 2
    <<< Loop-information End >>>
47  1 pp    do j = 1 , m
    <<< Loop-information Start >>>
    <<< [OPTIMIZATION]
    <<< SIMD (VL: 4)
    <<< SOFTWARE PIPELINING
    <<< Loop-information End >>>
48  2 p 4v    do i = 1 , n
49  2 p 4v      abcd(1,i,j)=abcd(2,i,j)+abcd(3,i,j)+abcd(4,i,j)+efgh(1,i,j)+efgh(2,i,j)+efgh(3,i,j)+efgh(4,i,j)
50  2 p 4v    enddo
51  1 p      enddo
    
```

Merging 8 arrays in units of 4



Cache

	L1I miss rate(/Effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	Memory throughput (GB/sec)	L2 throughput (GB/sec)
Before improvement	0.00%	23.21%	3.12E+09	91.66%	8.34%	0.00%	0.00%	0.00	261.73
After improvement	0.00%	3.19%	4.29E+08	25.52%	74.48%	0.00%	0.00%	0.01	335.65

The percentage of L1D cache misses decreased from 23.21% to 3.19%, and the L1D miss dm percentage decreased too from 91.66% to 25.52%.

Array Merging (in C Language) (Before Improvement) FUJITSU

L1D cache thrashing occurs because each array is located on a 16-KB boundary. Consequently, the following is a frequent event: No instruction commit due to L2 access for a floating-point load instruction.

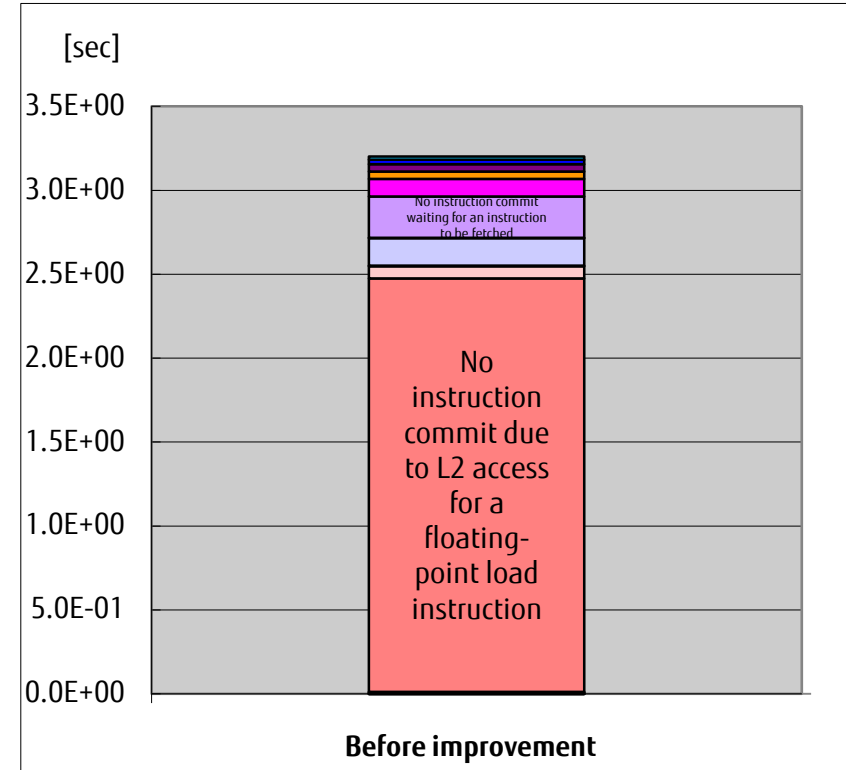
Source code before improvement

```

5  #define N 256
6  #define M 256
7
8  double a[M][N],b[M][N],c[M][N],d[M][N],e[M][N],f[M][N],g[M][N],h[M][N];
   :
   <<< Loop-information Start >>>
   <<< [PARALLELIZATION]
   <<< Standard iteration count: 433
   <<< [OPTIMIZATION]
   <<< COLLAPSED
   <<< SIMD (VL: 4)
   <<< SOFTWARE PIPELINING
   <<< Loop-information End >>>
47 pp 6v for(j=0;j<M;j++){
   <<< Loop-information Start >>>
   <<< [OPTIMIZATION]
   <<< COLLAPSED
   <<< Loop-information End >>>
48 p 6v for(i=0;i<N;i++){
49 p 6v  a[j][i] = b[j][i] + c[j][i] + d[j][i] + e[j][i] + f[j][i] + g[j][i] + h[j][i];
50 p 6v  }
   :

```

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



Cache

	L1I miss rate(/Effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwfp rate(/L1D miss)	L1D miss swfp rate(/L1D miss)	L2 miss rate(/Load-store instruction)	Memory throughput (GB/sec)	L2 throughput (GB/sec)
Before improvement	0.00%	21.95%	2.95E+09	91.96%	8.04%	0.00%	0.00%	0.00	235.97

The percentage of L1D misses is high and the L1 miss dm percentage is high, despite the fact that the array is accessed sequentially.
 ⇒ **L1D cache thrashing has occurred.**

Effects of Array Merging (in C Language) (Source Tuning)

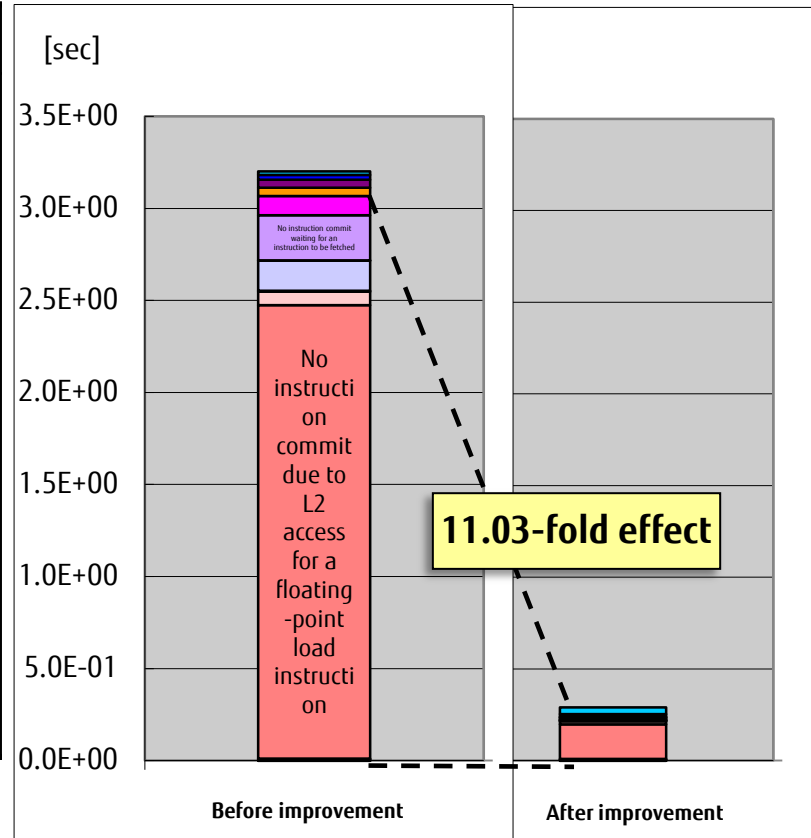
Array merging reduced the number of streams from eight to two, so L1D cache thrashing was avoided. This results in improvement of the following event: No instruction commit due to L2 access for a floating-point load instruction.

Source code after improvement (source tuning)

```

5  #define N 256
6  #define M 256
7
8  double abcd[M][N][4],efgh[M][N][4];
   :
<<< Loop-information Start >>>
<<< [PARALLELIZATION]
<<< Standard iteration count: 2
<<< Loop-information End >>>
43 pp   for(j=0;j<M;j++){
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< SIMD(VL: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End >>>
44 p 4v   for(i=0;i<N;i++){
45 p 4v   abcd[j][i][0] = abcd[j][i][1] + abcd[j][i][2] + abcd[j][i][3] + efgh[j][i][0] +
         efgh[j][i][1] + efgh[j][i][2] + efgh[j][i][3];
46 p 4v   }
47       }
   :
    
```

Merging 8 arrays in units of 4



Cache

	L1I miss rate/(Effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate/(L1D miss)	L1D miss hwpf rate/(L1D miss)	L1D miss swpf rate/(L1D miss)	L2 miss rate/(Load-store instruction)	Memory throughput (GB/sec)	L2 throughput (GB/sec)
Before improvement	0.00%	21.95%	2.95E+09	91.96%	8.04%	0.00%	0.00%	0.00	235.97
After improvement	0.00%	3.18%	4.27E+08	27.49%	72.51%	0.00%	0.00%	0.01	377.36

The percentage of L1D cache misses decreased from 21.95% to 3.18%, and the L1D miss dm percentage decreased too from 91.96% to 27.49%.

You can achieve effects similar to source tuning by specifying the following compiler options.

Compiler options	Description of function
-Karray_merge_common [=name]	Gives an instruction to merge multiple arrays in a common block. You can specify a common block name for name. If name is omitted, the arrays in all the named common blocks are targets.
-Karray_merge_local	Gives an instruction to merge multiple local arrays. -Karray_merge_local_size=1000000 is also valid at the same time.
-Karray_merge	This option is equivalent to specifying the -Karray_merge_local and -Karray_merge_common options.

■ Use example (source code before improvement)

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

◆ Notes

- Options must be specified for all source code that uses the target arrays.
- The effects of array merging vary depending on the program.
- Incorrect use may result in different computational results.
- These options cannot be used with debug options (-g and -Haesux).

Dimensional Displacement of an Array

- What Is Dimensional Displacement of an Array?
- Dimensional Displacement of an Array (Before Improvement)
- Effects of Dimensional Displacement of an Array (Source Tuning)
- Dimensional Displacement of an Array (in C Language) (Before Improvement)
- Effects of Dimensional Displacement of an Array (in C Language) (Source Tuning)
- Effects of Dimensional Displacement of an Array (Compiler Options Tuning)

What Is Dimensional Displacement of an Array?

Dimensional displacement of an array is a tuning method where multiple streams of the same array become one stream.

■ Use conditions

- Multiple streams exist in the same array.

* $a(1,1,1)$ to $a(1,1,8)$ are shown as multiple streams.

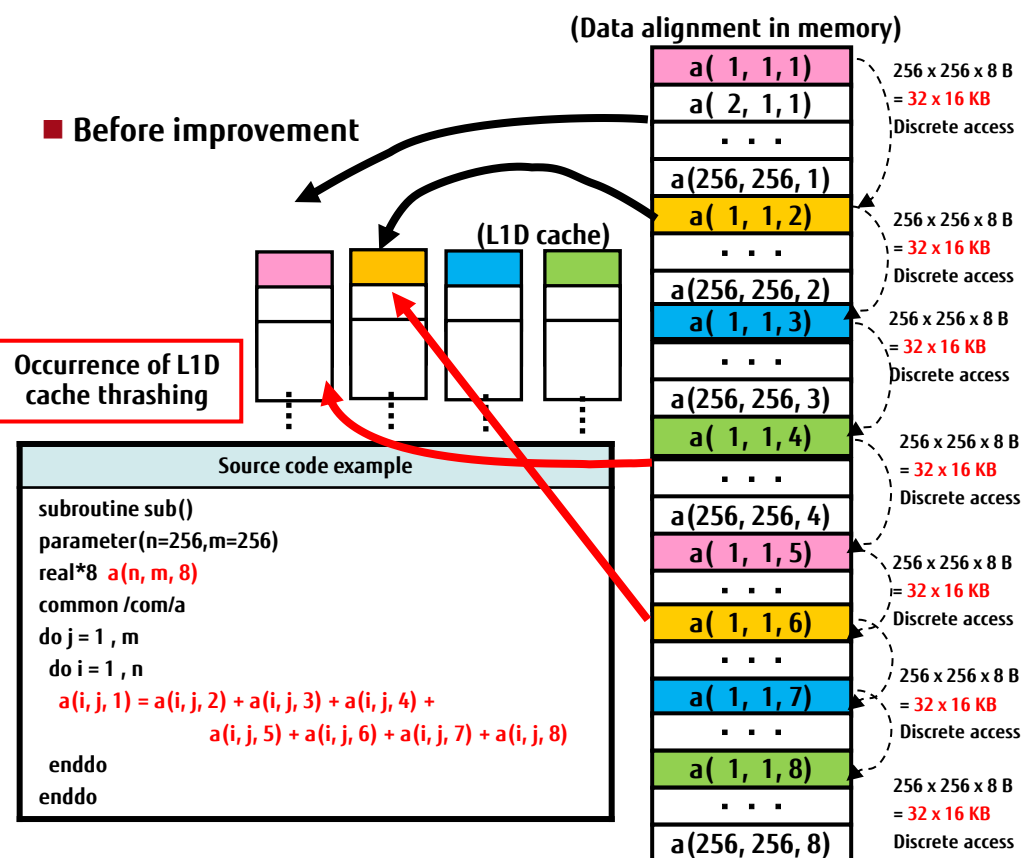
■ Purpose

- The purpose is to reduce the number of streams.

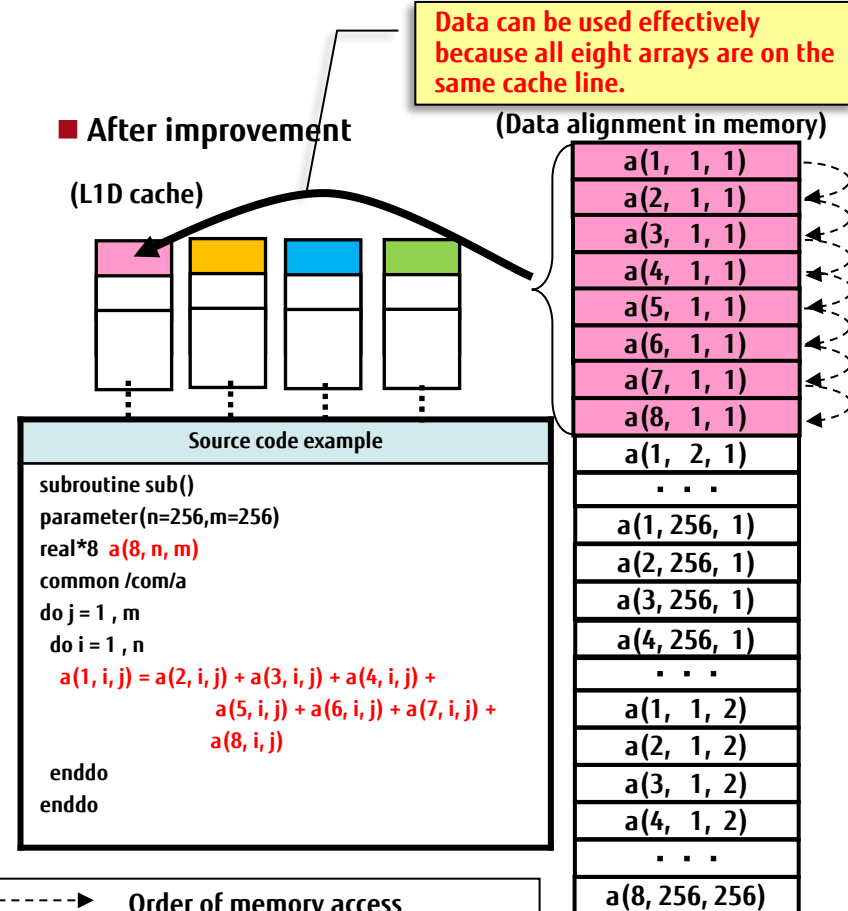
■ Adverse effect

- SIMD optimization of load and store instructions is more difficult.

■ Before improvement



■ After improvement



→ Storing data in cache → Storing data in cache (conflict) → Order of memory access

Dimensional Displacement of an Array (Before Improvement)

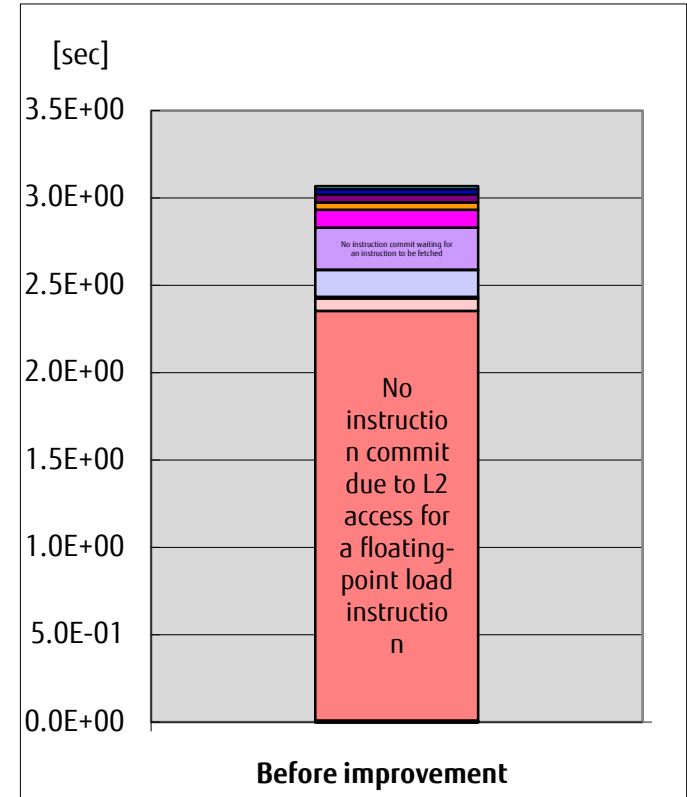
L1D cache thrashing occurs because each stream of array a is located on a 16-KB boundary. Consequently, the following is a frequent event: No instruction commit due to L2 access for a floating-point load instruction.

Source code before improvement

```

33     parameter(n=256,m=256)
34     real*8  a(n, m, 8)
35     common /com/a
    <<< Loop-information Start >>>
    <<< [PARALLELIZATION]
    <<< Standard iteration count: 422
    <<< [OPTIMIZATION]
    <<< COLLAPSED
    <<< SIMD(VL: 4)
    <<< SOFTWARE PIPELINING
    <<< Loop-information End >>>
36  1 pp  6v  do j = 1, m
    <<< Loop-information Start >>>
    <<< [OPTIMIZATION]
    <<< COLLAPSED
    <<< Loop-information End >>>
37  2 p  6    do i = 1, n
38  2 p  6v    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) + a(i, j, 8)
39  2 p  6v    enddo
40  1 p      enddo
    
```

Size of 1 stream
 $256 \times 256 \times 8 \text{ B} =$
 $32 \times 16 \text{ KB}$
(16-KB boundary)



Cache

Cache	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	23.26%	3.13E+09	91.57%	8.43%	0.00%	0.00%	261.00	0.00

The percentage of L1D misses is high and the L1 miss dm percentage is high, despite the fact that the array is accessed sequentially.

➡ L1D cache thrashing has occurred.

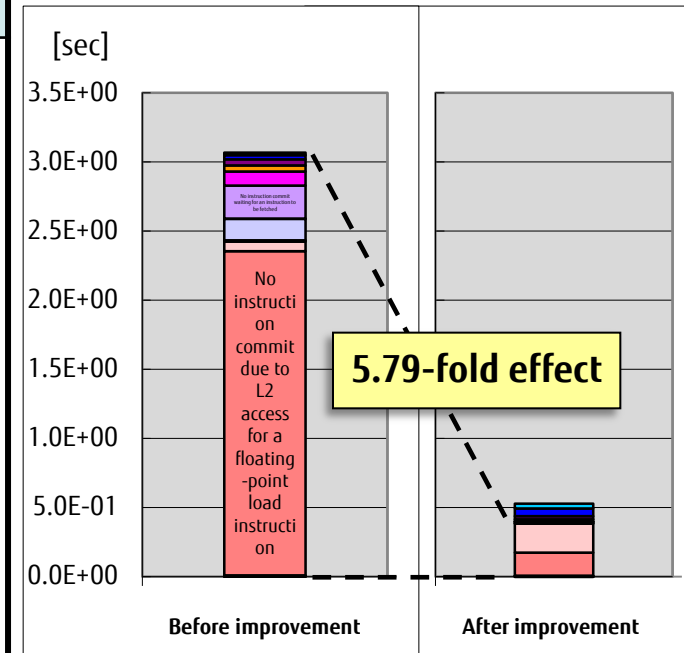
Effects of Dimensional Displacement of an Array (Source Tuning)

Dimensional displacement of an array reduced the number of streams from eight to one, so L1D cache thrashing was avoided. This results in improvement of the following event: No instruction commit due to L2 access for a floating-point load instruction.

Source code after improvement (source tuning)

```

33      parameter(n=256,m=256)
34      real*8 a(8, n, m)
35      common /com/a
      <<< Loop-information Start >>>
      <<< [PARALLELIZATION]
      <<< Standard iteration count: 2
      <<< Loop-information End >>>
36  1 pp      do j = 1, m
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< SOFTWARE PIPELINING
      <<< Loop-information End >>>
37  2 p 4v      do i = 1, n
38  2 p 4v      a(1, i, j) = a(2, i, j) + a(3, i, j) + a(4, i, j) + a(5, i, j) + a(6, i, j) + a(7, i, j) + a(8, i, j)
39  2 p 4v      enddo
40  1 p      enddo
  
```



Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate (/L1D miss)	L1D miss hwpf rate (/L1D miss)	L1D miss swpf rate (/L1D miss)	L2 miss rate (/Load-store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	23.26%	3.13E+09	91.57%	8.43%	0.00%	0.00%	261.00	0.00
After improvement	0.00%	3.16%	4.27E+08	20.03%	79.97%	0.00%	0.00%	207.01	0.00

The percentage of L1D misses decreased from 23.26% to 3.16%, and the L1D miss dm percentage decreased too from 91.57% to 20.03%.

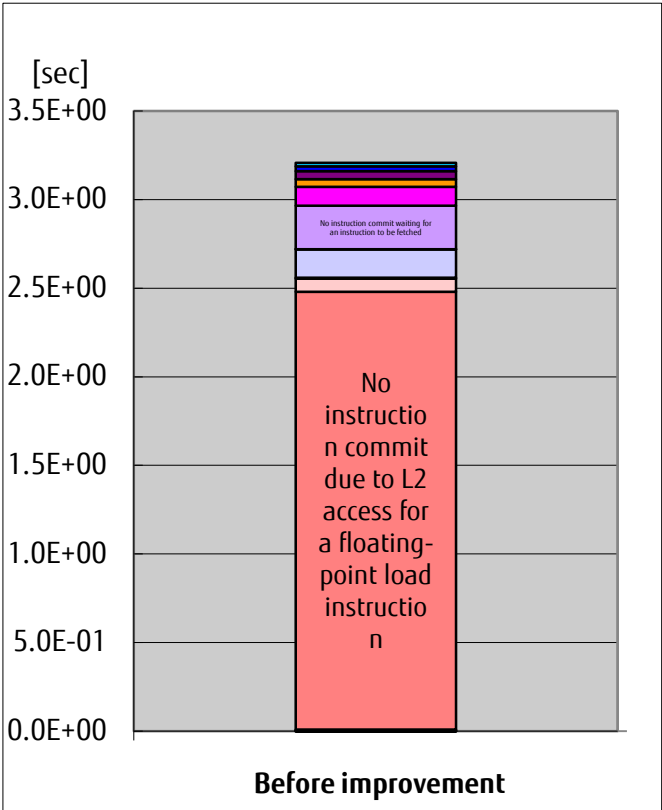
Dimensional Displacement of an Array (in C Language) (Before Improvement)

L1D cache thrashing occurs because each stream of array a is located on a 16-KB boundary. Consequently, the following is a frequent event: No instruction commit due to L2 access for a floating-point load instruction.

Source code before improvement

```
4  #define N 256
5  #define M 256
6  #define L 8
7
8  double a[L][M][N];
   :
   <<< Loop-information Start >>>
   <<< [PARALLELIZATION]
   <<< Standard iteration count: 433
   <<< [OPTIMIZATION]
   <<< COLLAPSED
   <<< SIMD (VL: 4)
   <<< SOFTWARE PIPELINING
   <<< Loop-information End >>>
39 pp 6v for(j=0;j<M;j++){
   <<< Loop-information Start >>>
   <<< [OPTIMIZATION]
   <<< COLLAPSED
   <<< Loop-information End >>>
40 p 6v for(i=0;i<N;i++){
41 p 6v 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] + a[7][j][i];
42 p 6v }
43 }
```

Size of 1 stream
256 x 256 x 8 B =
32 x 16 KB
(16-KB boundary)



Cache

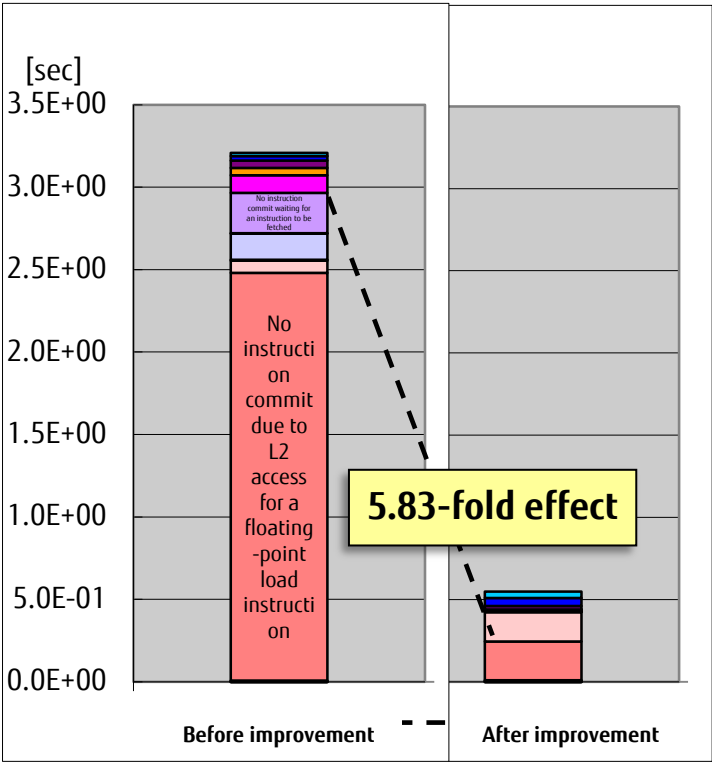
	L1 miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwfp rate(/L1D miss)	L1D miss swfp rate(/L1D miss)	L2 miss rate(/Load- store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	21.92%	2.95E+09	91.93%	8.07%	0.00%	0.00%	235.07	0.00

The percentage of L1D misses is high and the L1 miss dm percentage is high, despite the fact that the array is accessed sequentially.
⇒ L1D cache thrashing has occurred.

Dimensional displacement of an array reduced the number of streams from eight to one, so L1D cache thrashing was avoided. This results in improvement of the following event: No instruction commit due to L2 access for a floating-point load instruction.

Source code after improvement (source tuning)

```
4  #define N 256
5  #define M 256
6  #define L 8
7
8  double a[M][N][L];
9  :
10 <<< Loop-information Start >>>
11 <<< [PARALLELIZATION]
12 <<< Standard iteration count: 2
13 <<< Loop-information End >>>
39 pp   for(j=0;j<M;j++){
14 <<< Loop-information Start >>>
15 <<< [OPTIMIZATION]
16 <<< SIMD(VL: 4)
17 <<< SOFTWARE PIPELINING
18 <<< Loop-information End >>>
40 p 4v   for(i=0;i<N;i++){
41 p 4v     a[j][i][0] = a[j][i][1] + a[j][i][2] + a[j][i][3] + a[j][i][4] +
42 p 4v       a[j][i][5] + a[j][i][6] + a[j][i][7];
43 }
```



Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load- store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	21.92%	2.95E+09	91.93%	8.07%	0.00%	0.00%	235.07	0.00
After improvement	0.00%	3.16%	4.27E+08	22.84%	77.16%	0.00%	0.00%	199.05	0.00

The percentage of L1D misses decreased from 21.92% to 3.16%, and the L1D miss dm percentage decreased too from 91.93% to 22.84%.

You can achieve effects similar to source tuning by specifying the following compiler options.

Compiler options	Description of function
-Karray_subscript	Gives an instruction for dimensional displacement of allocatable arrays with 4 or more dimensions and arrays with 4 or more dimensions containing 10 or fewer elements in the final dimension and 100 or more elements in the other dimensions. -Karray_subscript_element=100,-Karray_subscript_elementlast=10, and -Karray_subscript_rank=4 are also valid at the same time.
-Karray_subscript_element=N ($2 \leq N \leq 2,147,483,647$)	Gives an instruction that the number of elements in a dimension other than the final dimension in an array subject to dimensional displacement be N or greater. This option has meaning in cases where the -Karray_subscript option is valid. However, the option has no meaning for an allocatable array.
-Karray_subscript_elementlast=N ($2 \leq N \leq 2,147,483,647$)	Gives an instruction that the number of elements in the final dimension of an array subject to dimensional displacement be N or less. This option has meaning in cases where the -Karray_subscript option is valid. However, the option has no meaning for an allocatable array.
-Karray_subscript_rank=N ($2 \leq N \leq 30$)	Gives an instruction that the number of dimensions of an array subject to dimensional displacement be N or greater. This option has meaning in cases where the -Karray_subscript option is valid.

■ Use example (source code before improvement)

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

```
-Karray_subscript,array_subscript_rank=2,array_subscript_element=2
```

◆ Notes

- Options must be specified for all source code that uses the target arrays.
- The effects of displacement vary depending on the program.
- Incorrect use may result in different computational results.

Loop Fission

- Loop Fission (Before Improvement)
- Effects of Loop Fission (Source Tuning)
- Loop Fission (in C Language) (Before Improvement)
- Effects of Loop Fission (in C Language) (Source Tuning)
- Effects of Loop Fission (Optimization Control Line Tuning)

Loop Fission (Before Improvement)

L1D cache thrashing occurs because each array is located on a 16-KB boundary. Consequently, the following is a frequent event: No instruction commit due to L2 access for a floating-point load instruction.

Source code 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: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End >>>

```

```

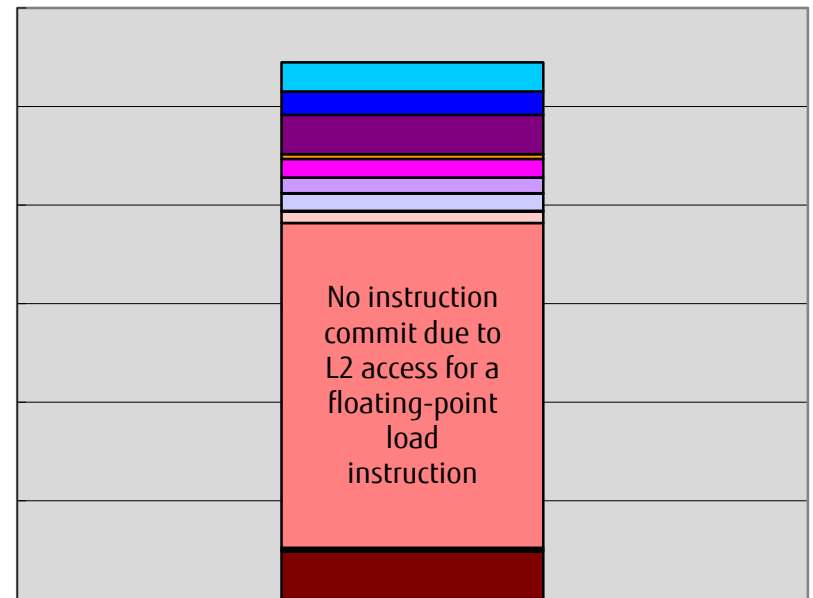
50  1 pp 2v  do i=1,n
51  1 p 2v   a(i) = s / b(i)
52  1 p 2v   c(i) = s / d(i)
53  1 p 2v   e(i) = s / f(i)
54  1 p 2v   g(i) = s / h(i)
55  1 p 2v  enddo

```

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

[sec]

6.0E-02
5.0E-02
4.0E-02
3.0E-02
2.0E-02
1.0E-02
0.0E+00



Before improvement

Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load- store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	9.03%	4.74E+07	73.50%	26.50%	0.00%	0.00%	222.75	0.18

The percentage of L1D misses is high and the L1 miss dm percentage is high, despite the fact that the array is accessed sequentially.
⇒ L1D cache thrashing has occurred.

Effects of Loop Fission (Source Tuning)

Loop fission reduced the number of streams from eight to four, so L1D cache thrashing was avoided. This results in improvement of the following event: No instruction commit due to L2 access for a floating-point load instruction.

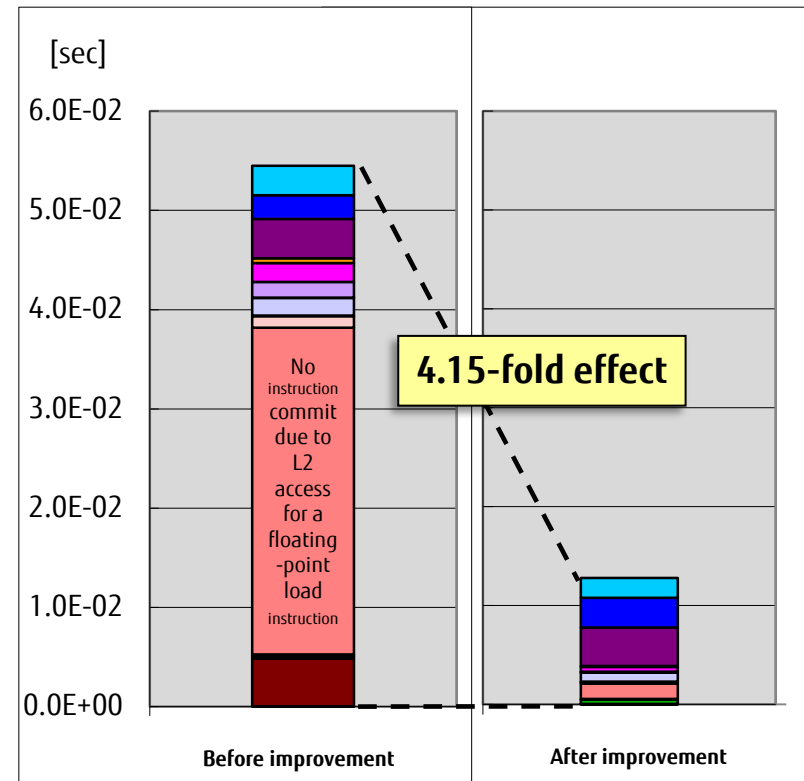
Source code after improvement (source tuning)

```

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
50      !OCL LOOP_NOFUSION
      <<< Loop-information Start >>>
      <<< [PARALLELIZATION]
      <<< Standard iteration count: 381
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< SOFTWARE PIPELINING
      <<< Loop-information End >>>
51  1 pp 4v  do i=1,n
52  1 p 4v   a(i) = s / b(i)
53  1 p 4v   c(i) = s / d(i)
54  1 p 4v  enddo
      <<< Loop-information Start >>>
      <<< [PARALLELIZATION]
      <<< Standard iteration count: 381
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< SOFTWARE PIPELINING
      <<< Loop-information End >>>
55  1 pp 4v  do i=1,n
56  1 p 4v   e(i) = s / f(i)
57  1 p 4v   g(i) = s / h(i)
58  1 p 4v  enddo
  
```

Suppressing loop fusion

Loop fission



The percentage of L1D misses decreased from 9.03% to 3.25%, and the L1D miss dm percentage decreased too from 73.50% to 15.93%.

Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	9.03%	4.74E+07	73.50%	26.50%	0.00%	0.00%	222.75	0.18
After improvement	0.00%	3.25%	1.71E+07	15.93%	84.07%	0.00%	0.00%	341.98	0.70

Loop Fission (in C Language) (Before Improvement)

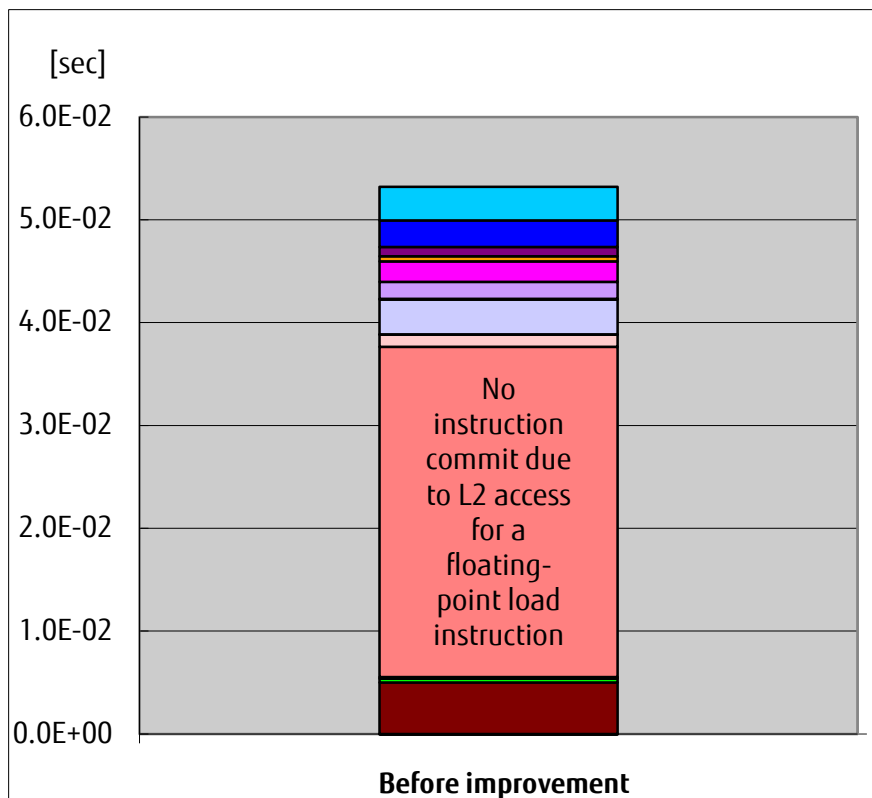
L1D cache thrashing occurs because each array is located on a 16-KB boundary. Consequently, the following is a frequent event: No instruction commit due to L2 access for a floating-point load instruction.

Source code before improvement

```

4  #define N 65536
5
6  double a[N],b[N],c[N],d[N],e[N],f[N],g[N],h[N];
    :
<<< Loop-information Start >>>
<<< [PARALLELIZATION]
<<< Standard iteration count: 206
<<< [OPTIMIZATION]
<<< SIMD(VL: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End >>>
46 pp 2v for(i=0;i<N;i++){
47 p 2v a[i] = s / b[i];
48 p 2v c[i] = s / d[i];
49 p 2v e[i] = s / f[i];
50 p 2v g[i] = s / h[i];
51 p 2v }
52 }
```

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



The percentage of L1D misses is high and the L1 miss dm percentage is high, despite the fact that the array is accessed sequentially.

⇒ L1D cache thrashing has occurred.

Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load- store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	8.76%	4.60E+07	73.69%	26.31%	0.00%	0.00%	221.43	0.03

Effects of Loop Fission (in C Language) (Source Tuning)

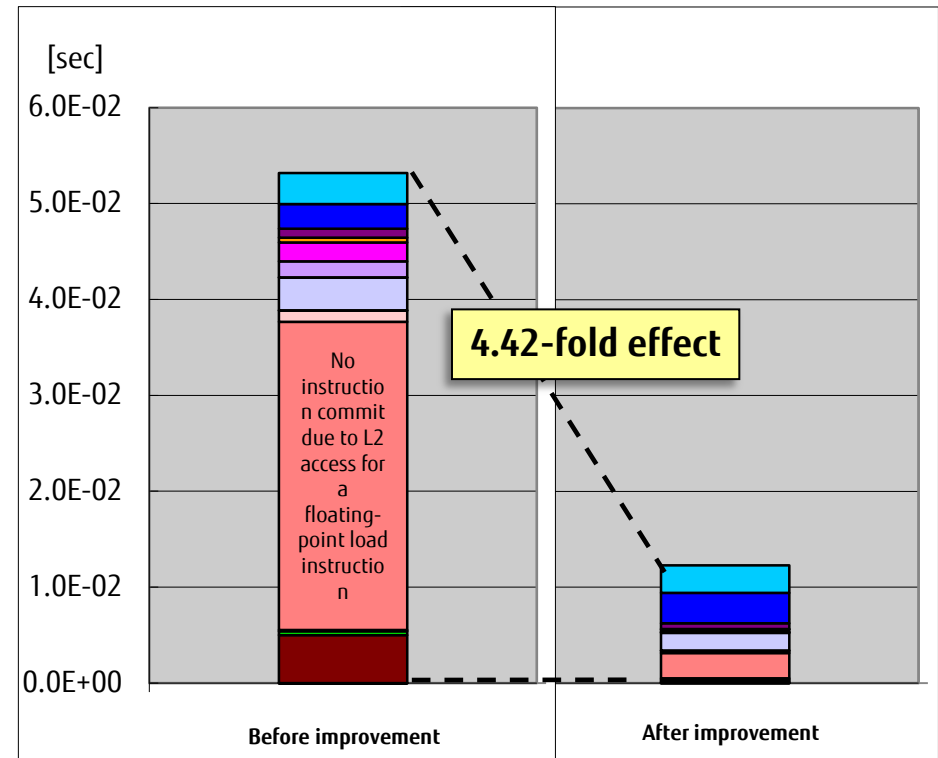
Loop fission reduced the number of streams from eight to four, so L1D cache thrashing was avoided. This results in improvement of the following event: No instruction commit due to L2 access for a floating-point load instruction.

```
Source code after improvement (source tuning)

4  #define N 65536
5
6  double a[N],b[N],c[N],d[N],e[N],f[N],g[N],h[N];
   :
46  #pragma loop loop_nofusion
   <<< Loop-information Start >>>
   <<< [PARALLELIZATION]
   <<< Standard iteration count: 381
   <<< [OPTIMIZATION]
   <<< SIMD(VL: 4)
   <<< SOFTWARE PIPELINING
   <<< Loop-information End >>>
47 pp 4v for(i=0;i<N;i++){
48 p 4v a[i] = s / b[i];
49 p 4v c[i] = s / d[i];
50 p 4v }
   <<< Loop-information Start >>>
   <<< [PARALLELIZATION]
   <<< Standard iteration count: 381
   <<< [OPTIMIZATION]
   <<< SIMD(VL: 4)
   <<< SOFTWARE PIPELINING
   <<< Loop-information End >>>
51 pp 4v for(i=0;i<N;i++){
52 p 4v e[i] = s / f[i];
53 p 4v g[i] = s / h[i];
54 p 4v }
```

Suppressing loop fusion

Loop fission



The percentage of L1D misses decreased from 8.76% to 3.23%, and the L1D miss dm percentage decreased too from 73.69% to 20.84%.

Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load- store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	8.76%	4.60E+07	73.69%	26.31%	0.00%	0.00%	221.43	0.03
After improvement	0.00%	3.23%	1.70E+07	20.84%	79.16%	0.00%	0.00%	353.87	0.02

Effects of Loop Fission (Optimization Control Line Tuning)

You can achieve effects similar to source tuning by specifying the following optimization control line.

Optimization control specifiers	Meaning	Optimization control line that can be specified			
		Program unit	DO loop unit	Statement unit	Array assignment statement unit
!OCL FISSION_POINT[(n1)] (where <i>n1</i> is decimal number from 1 to 6)	Gives an instruction for loop fission at the specified point inside a loop. The loop fission divides multiple loops that have loops nested to <i>n1</i> levels (counting from the innermost loop).	No	No	Yes	No

Source code 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: 381
    <<< [OPTIMIZATION]
    <<< SPLIT
    <<< SIMD(VL: 4)
    <<< SOFTWARE PIPELINING
    <<< Loop-information End >>>
50  1 pp 4v  do i=1,n
51  1 p 4v   a(i) = s / b(i)
52  1 p 4v   c(i) = s / d(i)
53  1      !ocl fission_point(1)
54  1 p 4v   e(i) = s / f(i)
55  1 p 4v   g(i) = s / h(i)
56  1 p 4v  enddo
    
```

jwd8212o-i "a.f90", line 54: Loop is divided (loop fission).

Padding

- What Is Padding?
- Padding That Increases the Number of Array Elements in the First Dimension
- Padding That Increases the Number of Array Elements in the Second Dimension
- Padding Using a Dummy Array
- Padding Using a Dummy Array (for Arrays of Different Sizes)

What Is Padding?

Padding inserts a dummy area between arrays or inside an array.

■ Use conditions

- Multiple streams exist in the same array.
Alternatively,
- Multiple arrays exist.

■ Purpose

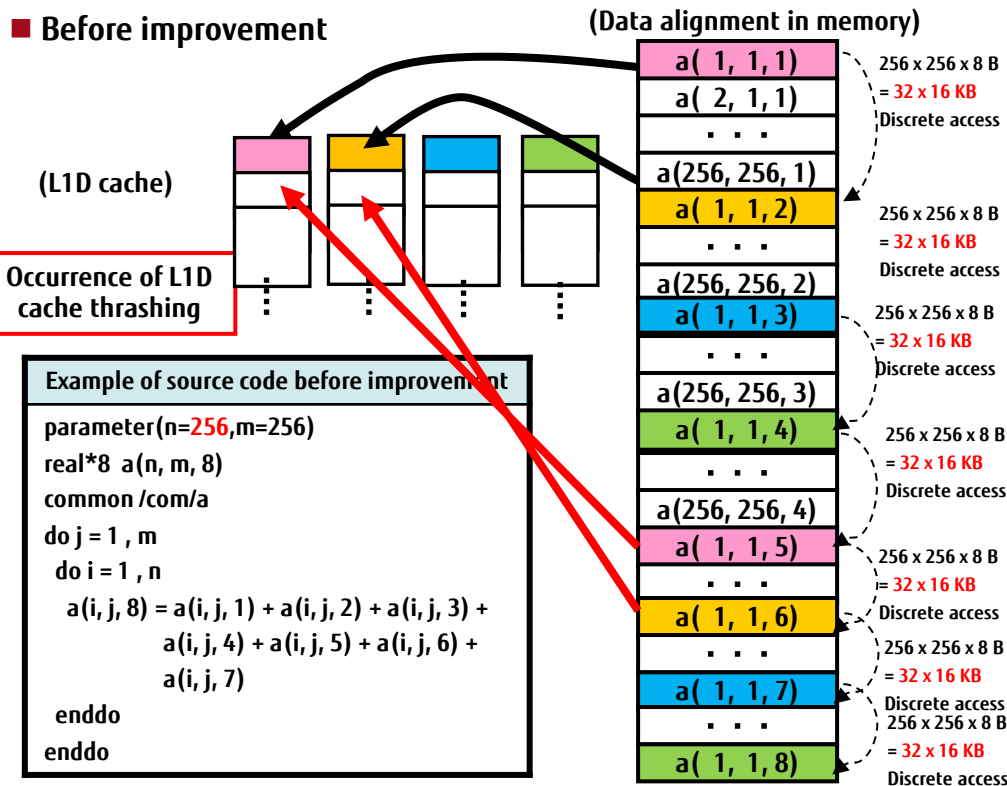
- The purpose is to create a temporary area to shift addresses.

■ Adverse effect

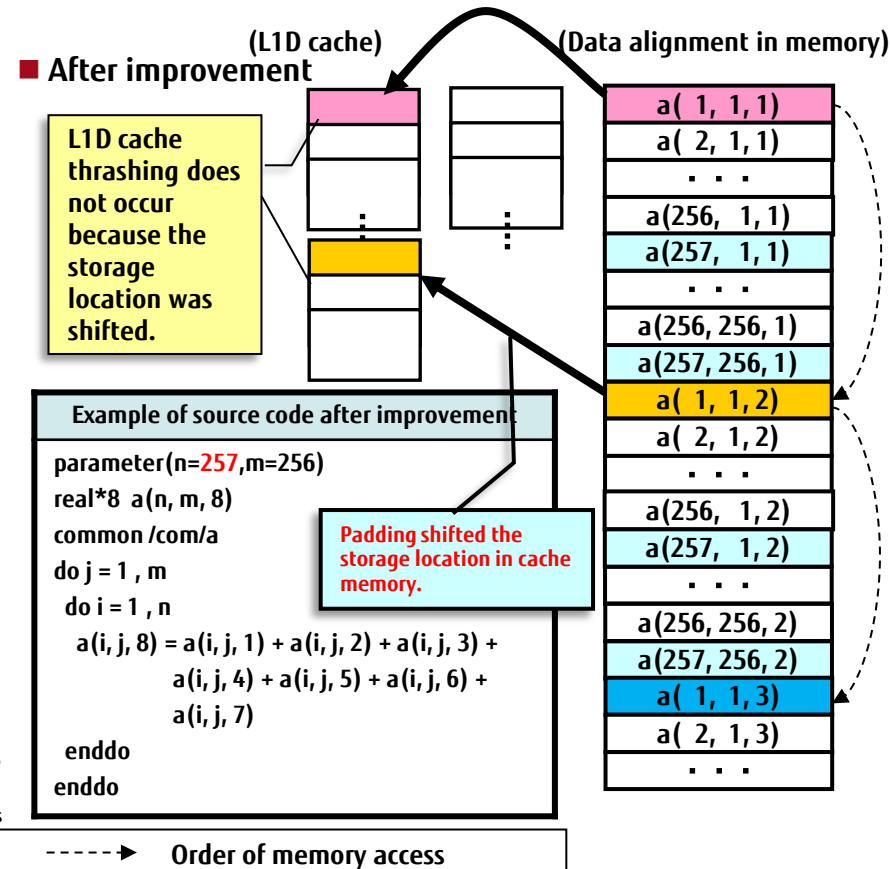
- The amount of padding must be changed every time that the problem scale changes.

Example where multiple streams exist in the same array

■ Before improvement



■ After improvement



→ Storing data in cache

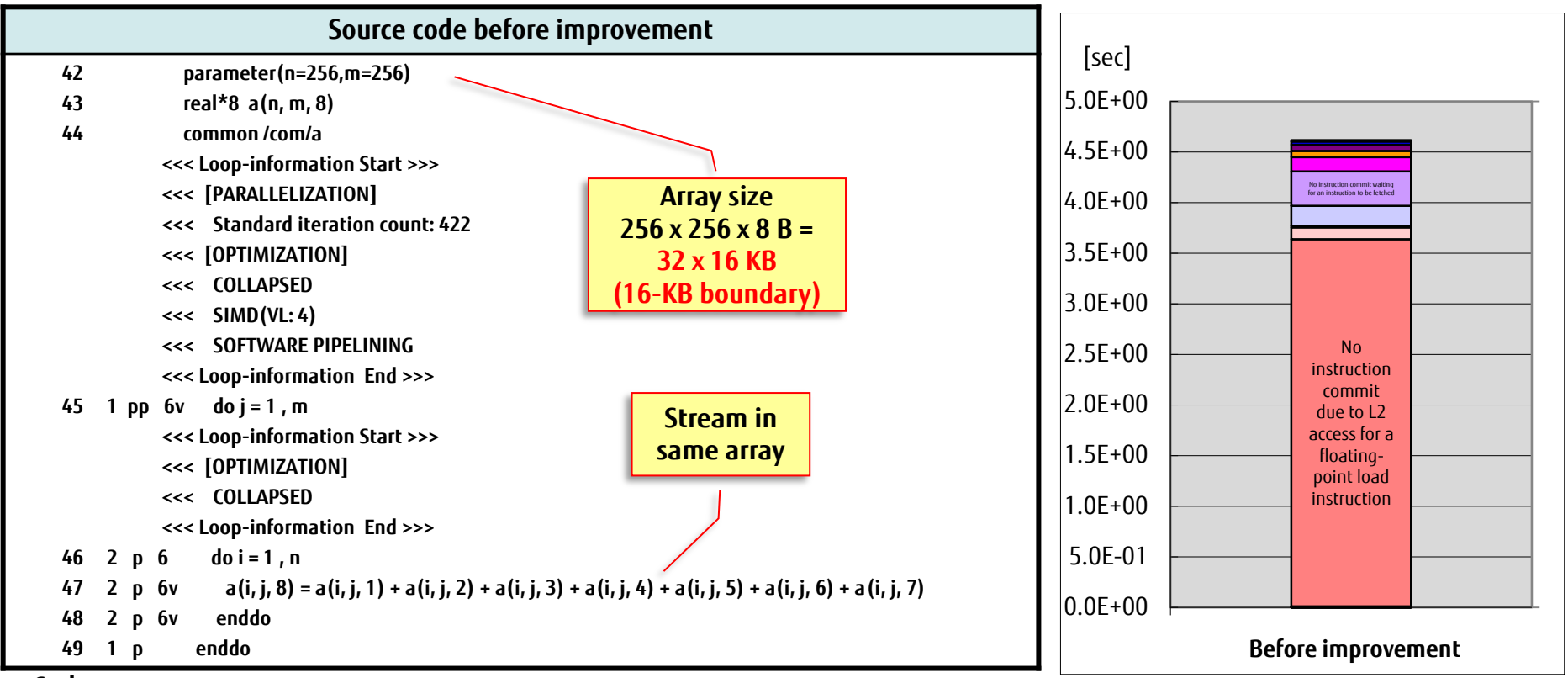
→ Storing data in cache (conflict)

-----> Order of memory access

Padding That Increases the Number of Array Elements in the First Dimension

- Padding That Increases the Number of Array Elements in the First Dimension (Before Improvement)
- Effects of Padding That Increases the Number of Array Elements in the First Dimension (Source Tuning)
- Padding That Increases the Number of Array Elements in the First Dimension (in C Language) (Before Improvement)
- Effects of Padding That Increases the Number of Array Elements in the First Dimension (in C Language) (Source Tuning)
- Effects of Padding That Increases the Number of Array Elements in the First Dimension (Compiler Options Tuning)

L1D cache thrashing occurs because each stream of array a is located on a 16-KB boundary. Consequently, the following is a frequent event: No instruction commit due to L2 access for a floating-point load instruction.



Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load- store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	33.19%	4.47E+09	95.12%	4.88%	0.00%	0.00%	247.59	0.00

The percentage of L1D cache misses is high and the demand percentage of L1D cache misses is high, despite the fact that the array is accessed sequentially.
⇒ L1D cache thrashing has occurred.

L1D cache thrashing was avoided because of padding (+1) of the first dimension of each stream of array a. This results in improvement of the following event: No instruction commit due to L2 access for a floating-point load instruction.

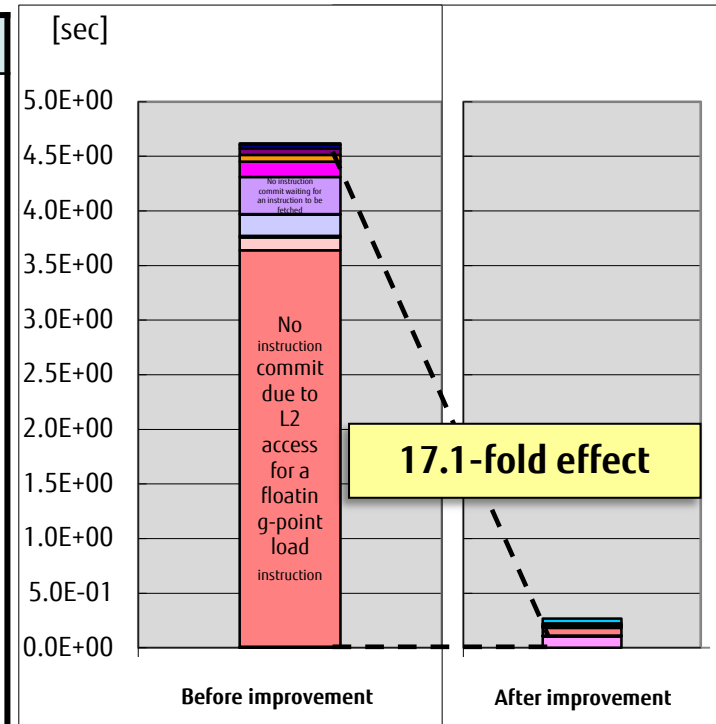
Source code after improvement (source tuning)

```

42  parameter(n=256,m=256)
43  real*8 a(n+1, m, 8)
44  common /com/a
    <<< Loop-information Start >>>
    <<< [PARALLELIZATION]
    <<< Standard iteration count: 2
    <<< Loop-information End >>>
45  1 pp  do j = 1, m
    <<< Loop-information Start >>>
    <<< [OPTIMIZATION]
    <<< SIMD(VL: 4)
    <<< SOFTWARE PIPELINING
    <<< Loop-information End >>>
46  2 p 4v do i = 1, n
47  2 p 4v  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)
48  2 p 4v  enddo
49  1 p  enddo
  
```

Shifting from 16-KB
boundary by **adding 1 to n**

Stream in
same array



Cache

The percentage of L1D misses decreased from 33.19% to 3.27%, and the L1D miss dm percentage decreased too from 95.12% to 9.46%.

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	33.19%	4.47E+09	95.12%	4.88%	0.00%	0.00%	247.59	0.00
After improvement	0.00%	3.27%	4.39E+08	9.46%	90.54%	0.00%	0.00%	421.35	0.01

Padding That Increases the Number of Array Elements in the First Dimension (in C Language) (Before Improvement)

L1D cache thrashing occurs because each stream of array a is located on a 16-KB boundary. Consequently, the following is a frequent event: No instruction commit due to L2 access for a floating-point load instruction.

Source code before improvement

```

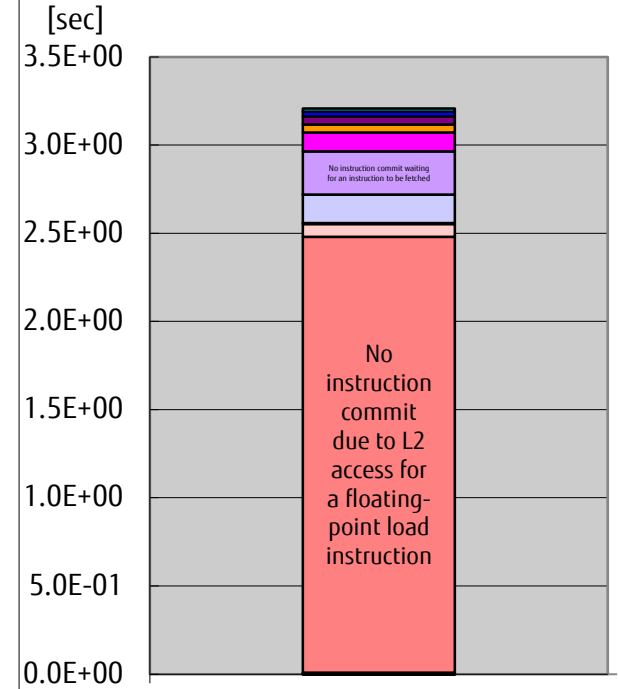
4  #define N 256
5  #define M 256
6  #define L 8
7
8  double a[L][M][N];
   :
<<< Loop-information Start >>>
<<< [PARALLELIZATION]
<<< Standard iteration count: 433
<<< [OPTIMIZATION]
<<< COLLAPSED
<<< SIMD(VL: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End >>>
39 pp 6v for(j=0;j<M;j++){
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< COLLAPSED
<<< Loop-information End >>>
40 p 6v for(i=0;i<N;i++){
41 p 6v  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] + a[7][j][i];
42 p 6v  }
43      }
    
```

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

Stream in
same array

The percentage of L1D misses is high and the demand percentage of L1D cache misses is high, despite the fact that the array is accessed sequentially.

⇒ L1D cache thrashing has occurred.



Before improvement

Cache

	L1 miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	21.92%	2.95E+09	91.92%	8.08%	0.00%	0.00%	235.08	0.00

Effects of Padding That Increases the Number of Array Elements in the First Dimension (in C Language) (Source Tuning)

L1D cache thrashing was avoided because of padding (+1) of the first dimension of each stream of array a. This results in improvement of the following event: No instruction commit due to L2 access for a floating-point load instruction.

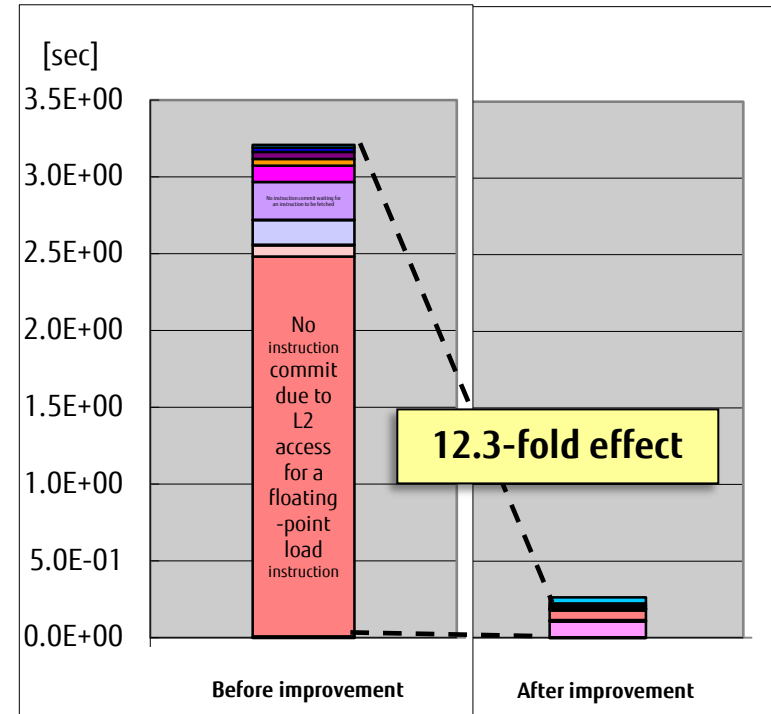
Source code after improvement (source tuning)

```

4  #define N 256
5  #define M 256
6  #define L 8
7
8  double a[L][M][N+1];
   :
<<< Loop-information Start >>>
<<< [PARALLELIZATION]
<<< Standard iteration count: 2
<<< Loop-information End >>>
38 pp for(j=0;j<M;j++){
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< SIMD(VL: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End >>>
39 p 4v for(i=0;i<N;i++){
40 p 4v 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] + a[7][j][i];
41 p 4v }
42 }
```

Shifting from 16-KB boundary by adding 1 to n

Stream in same array



Cache

The percentage of L1D misses decreased from 21.92% to 3.26%, and the L1D miss dm percentage decreased too from 91.92% to 8.79%.

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	21.92%	2.95E+09	91.92%	8.08%	0.00%	0.00%	235.08	0.00
After improvement	0.00%	3.26%	4.37E+08	8.79%	91.21%	0.00%	0.00%	425.99	0.01

You can achieve effects similar to source tuning by specifying the following compiler options.

Compiler options	Description of function
-Karraypad_const[=N] ($1 \leq N \leq 2,147,483,647$)	Pads N elements of an array whose first dimension is an explicit shape specification and shape specification expression is a constant expression. If N is omitted, the compiler determines the amount of padding for each target array. The padding creates a gap in the array.
-Karraypad_expr=N ($1 \leq N \leq 2,147,483,647$)	Pads N elements of an array whose first dimension is an explicit shape specification, regardless of whether its shape specification expression is a constant expression.

■ Use example (source code before improvement)

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

◆ Automatic selection of target arrays ➡ Application of padding

◆ Notes

- Options must be specified for all source code that uses the target arrays.
- The effects of padding vary depending on the program.
- Incorrect use may result in different computational results.
- The -Karraypad_const [=N] option and -Karraypad_expr=N option cannot be specified at the same time.

Padding That Increases the Number of Array Elements in the Second Dimension

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

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

Depending on the array size, there may be no improvement even with padding (+1) of the array elements of the first dimension.

Source code before improvement

```

33 parameter(k=64,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
        
```

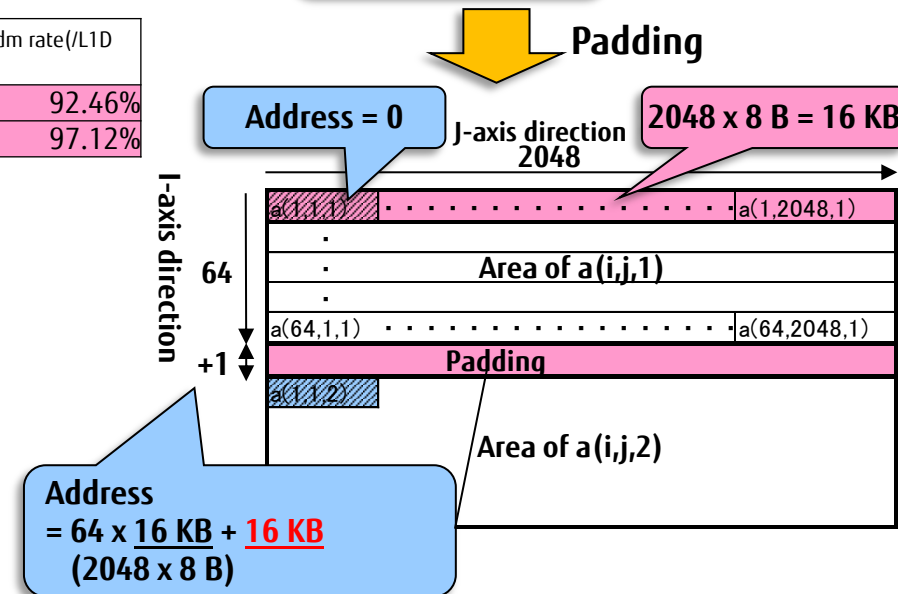
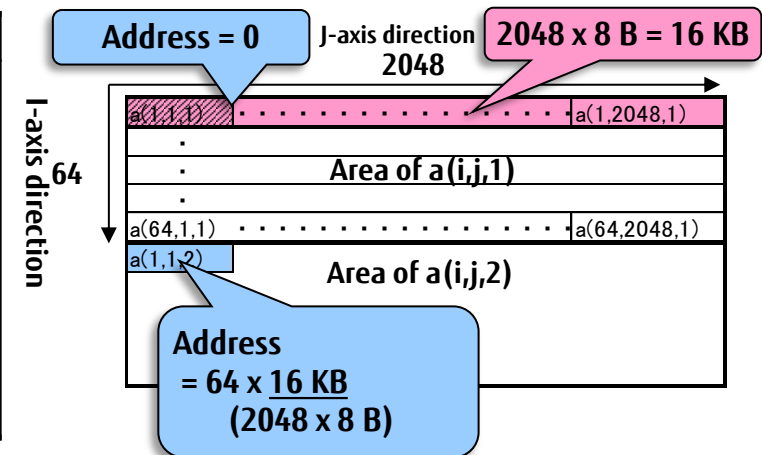
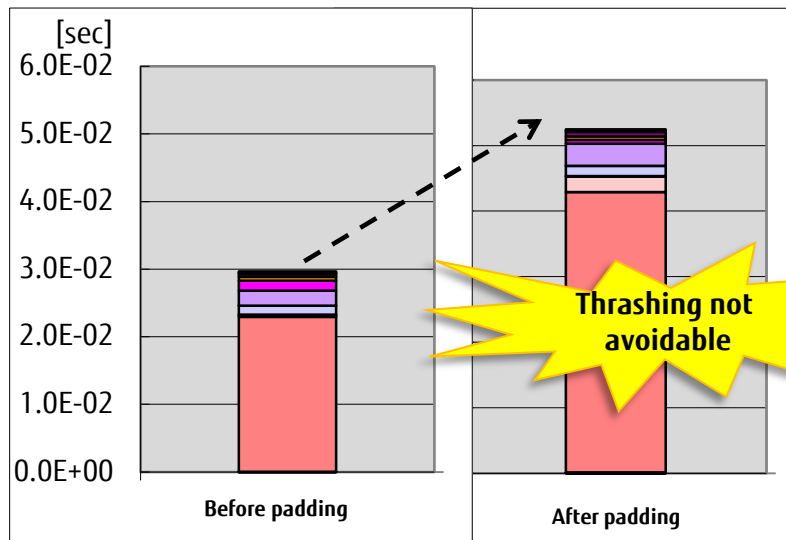
→

a(k, l, 8)

Padding

Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)
Before improvement	0.01%	26.39%	2.77E+07	92.46%
After improvement	0.01%	37.12%	3.90E+07	97.12%

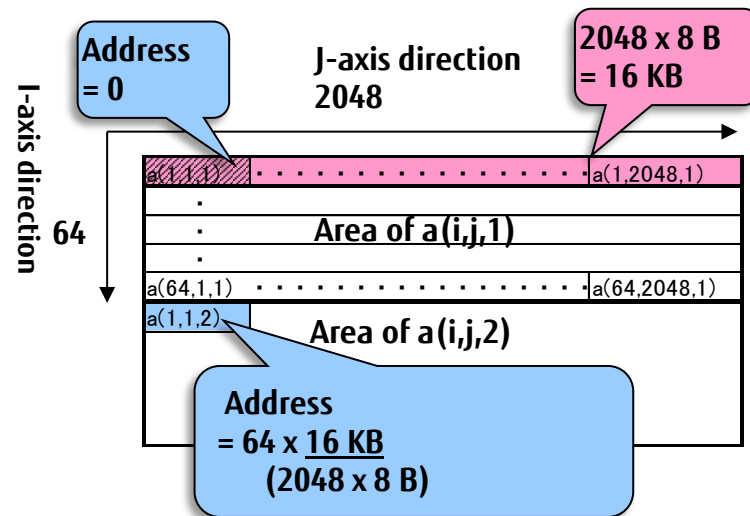


Thrashing occurs since array a remains on a 16-KB boundary.

L1D cache thrashing is avoided because padding (+1) of the second dimension causes a shift from the 16-KB boundary.

Source code before improvement

```
33 parameter(k=64,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
```

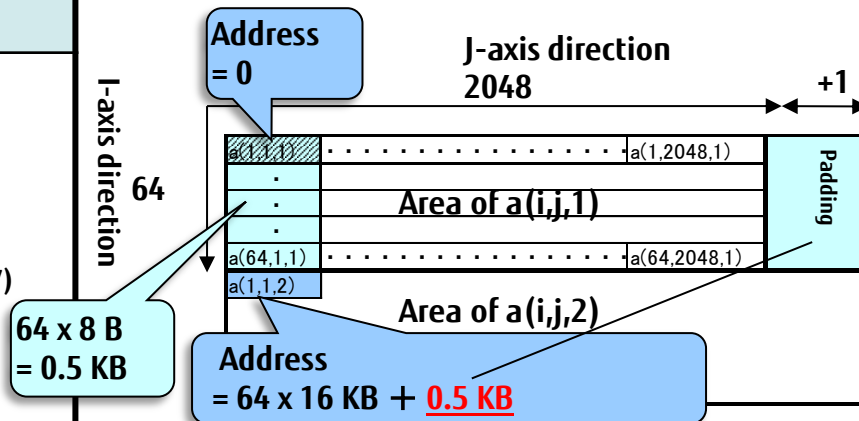


Thrashing occurs because of the 16-KB boundary.

Padding of the second dimension of array a

Source code after improvement

```
33 parameter(k=64,l=2048)
34 real*8 a(k+1, 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
```



Thrashing is avoided because the 16-KB boundary is no longer valid.

L1D cache thrashing occurs because each stream of array a is located on a 16-KB boundary. Consequently, the following is a frequent event: No instruction commit due to L2 access for a floating-point load instruction.

Source code before improvement

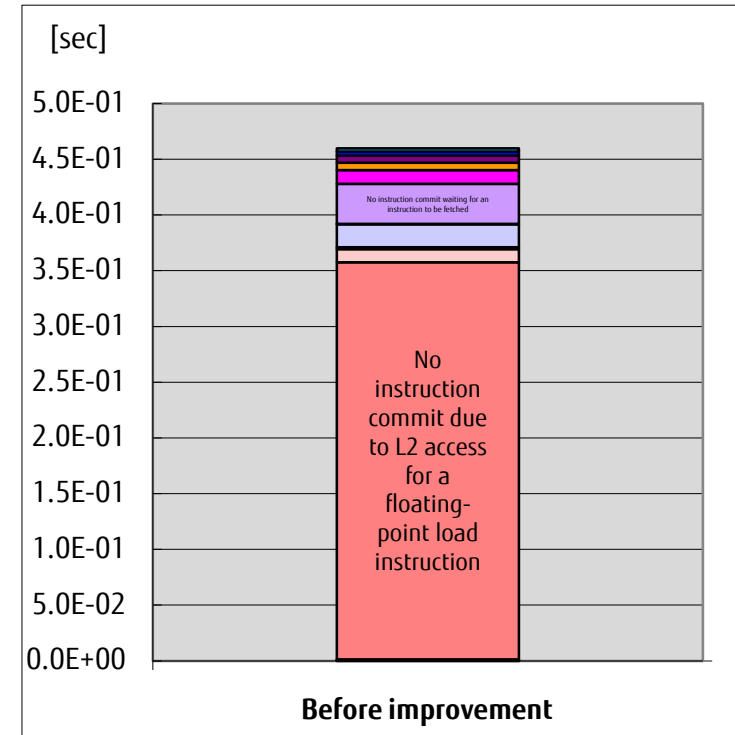
```

38  parameter(n=32,m=2048)
39  real*8 a(n, m, 8)
40  common /com/a
<<< Loop-information Start >>>
<<< [PARALLELIZATION]
<<< Standard iteration count: 422
<<< [OPTIMIZATION]
<<< COLLAPSED
<<< SIMD(VL: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End >>>
41  1 pp 6v do j = 1, m
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< COLLAPSED
<<< Loop-information End >>>
42  2 p 6 do i = 1, n
43  2 p 6v 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)
44  2 p 6v enddo
45  1 p enddo

```

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

Stream in
same array



The percentage of L1D cache misses is high and the demand percentage of L1D cache misses is high, despite the fact that the array is accessed sequentially.

⇒ L1D cache thrashing has occurred.

Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load- store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	26.31%	4.42E+08	92.60%	7.40%	0.00%	0.00%	246.39	0.01

L1D cache thrashing was avoided because of padding (+1) of the second dimension of each stream of array a. This results in improvement of the following event: No instruction commit due to L2 access for a floating-point load instruction.

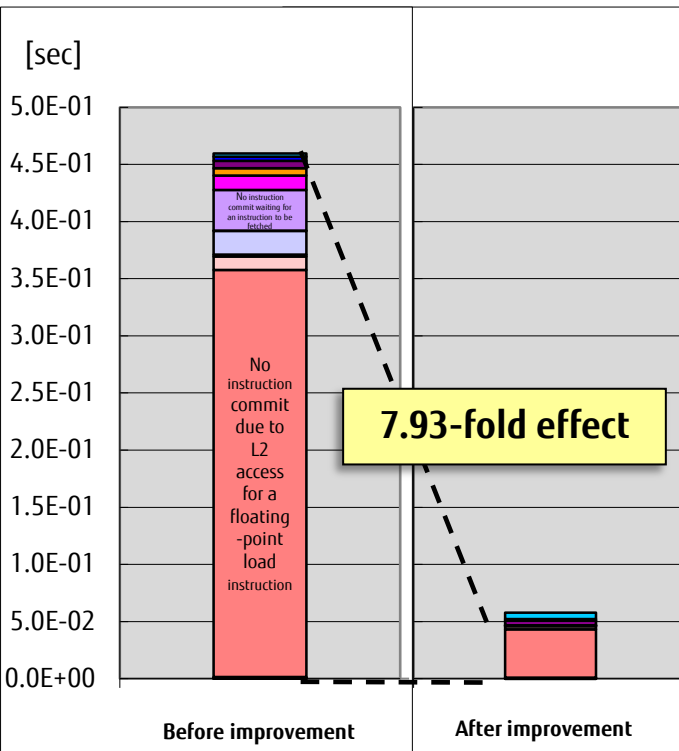
Source code after improvement (source tuning)

```

39  parameter(n=32,m=2048)
40  real*8 a(n, m+1, 8)
41  common /com/a
<<< Loop-information Start >>>
<<< [PARALLELIZATION]
<<< Standard iteration count: 422
<<< [OPTIMIZATION]
<<< COLLAPSED
<<< SIMD(VL: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End >>>
42  1 pp 6v do j = 1, m
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< COLLAPSED
<<< Loop-information End >>>
43  2 p 6 do i = 1, n
44  2 p 6v 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)
45  2 p 6v enddo
46  1 p enddo

```

Shifting from 16-KB
boundary by adding 1 to m



Cache

The percentage of L1D misses decreased from 26.31% to 6.12%, and the L1D miss dm percentage decreased too from 92.60% to 52.53%.

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	26.31%	4.42E+08	92.60%	7.40%	0.00%	0.00%	246.39	0.01
After improvement	0.00%	6.12%	1.03E+08	52.53%	47.47%	0.00%	0.00%	456.13	0.02

Padding Using a Dummy Array

- Padding Using a Dummy Array (Before Improvement)
- Effects of Padding Using a Dummy Array (Source Tuning)
- Effects of Padding Using a Dummy Array (Compiler Options Tuning)

Padding Using a Dummy Array (Before Improvement)

L1D cache thrashing occurs because each array is located on a 16-KB boundary. Consequently, the following is a frequent event: No instruction commit due to L2 access for a floating-point load instruction.

Source code before 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=' '
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: 422
      <<< [OPTIMIZATION]
      <<< COLLAPSED
      <<< SIMD(VL: 4)
      <<< SOFTWARE PIPELINING
      <<< Loop-information End >>>
38 1 pp 6v      do j = 1, m
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< COLLAPSED
      <<< Loop-information End >>>
39 2 p 6      do i = 1, n
40 2 p 6v      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 6v      enddo
42 1 p      enddo

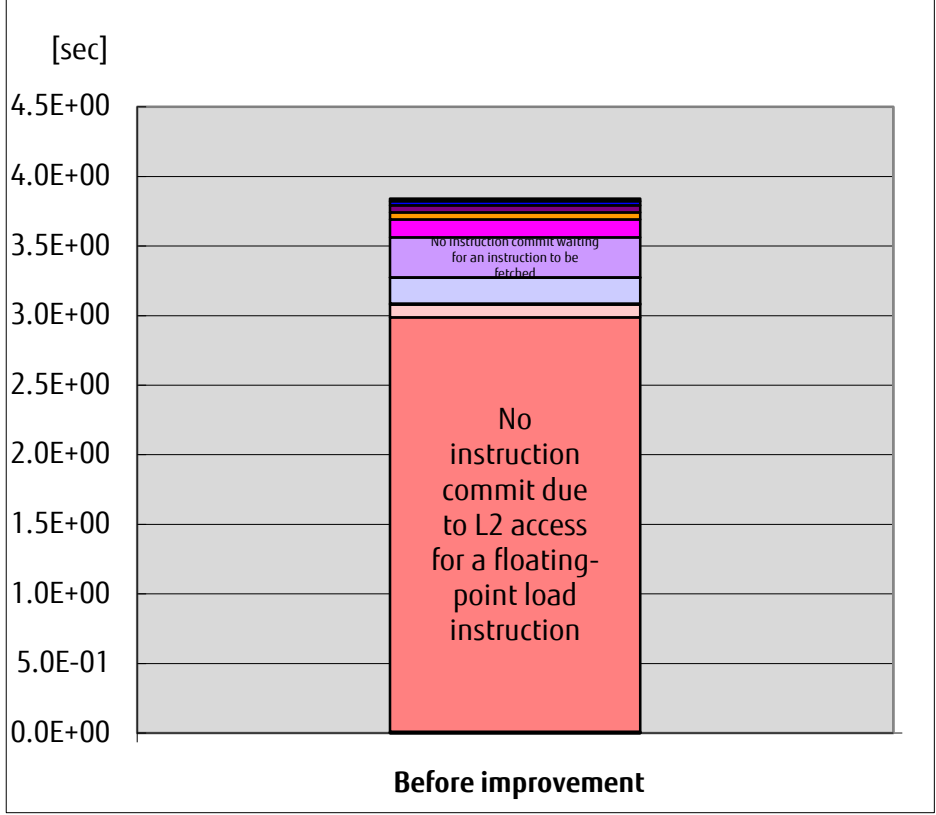
```

Array size

256 x 256 x 8 B =

32 x 16 KB

(16-KB boundary)



The percentage of L1D misses is high and the demand percentage of L1D cache misses is high, despite the fact that the array is accessed sequentially.

⇒ L1D cache thrashing has occurred.

Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load- store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	29.46%	3.96E+09	93.32%	6.68%	0.00%	0.00%	264.25	0.00

Effects of Padding Using a Dummy Array (Source Tuning)

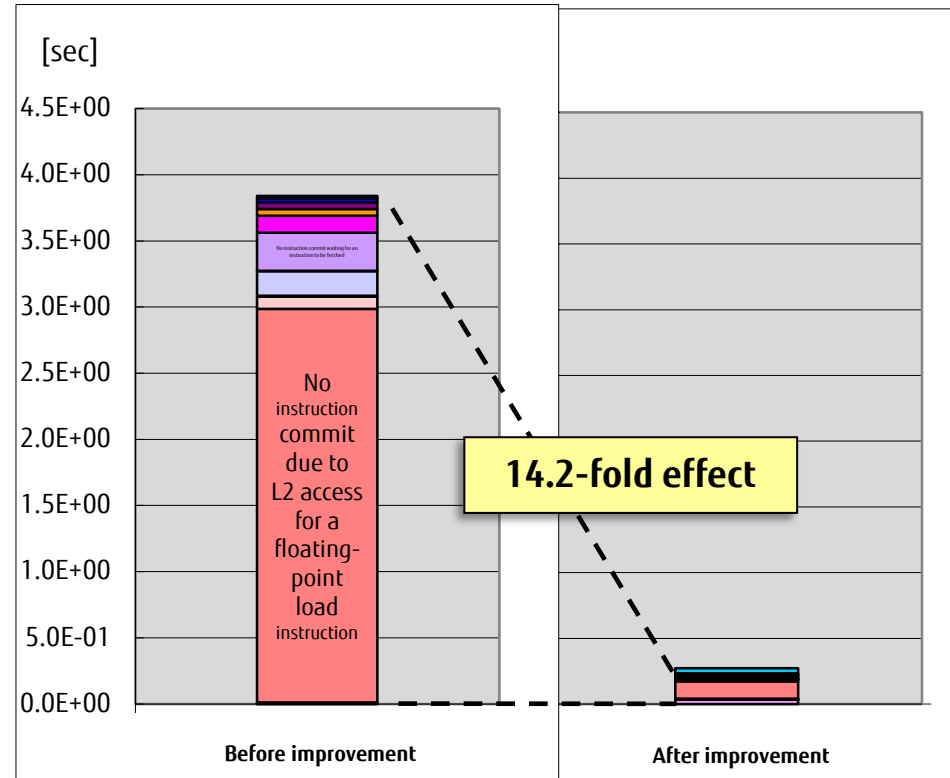
L1D cache thrashing was avoided because a dummy array was inserted between arrays to cause a shift from the 16-KB boundary. This results in improvement of the following event: No instruction commit due to L2 access for a floating-point load instruction.

Source code after improvement (source tuning)

```

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='z'00'
5      common /test/a,dummy1,b,dummy2,c,dummy3,d,dummy4,
          e,dummy5,f,dummy6,g,dummy7,h
          .....
28 1 s s   call sub()
          .....
35      subroutine sub()
36      parameter(n=256,m=256)
37      real*8 a(n, m),dummy1(64),b(n,m),dummy2(64),c(n,m),
          dummy3(64),d(n,m),dummy4(64)
38      real*8 e(n, m),dummy5(64),f(n,m),dummy6(64),g(n,m),
          dummy7(64),h(n,m)
39      common /test/a,dummy1,b,dummy2,c,dummy3,d,
          dummy4,e,dummy5,f,dummy6,g,dummy7,h
<<< Loop-information Start >>>
<<< [PARALLELIZATION]
<<< Standard iteration count: 422
<<< [OPTIMIZATION]
<<< COLLAPSED
<<< SIMD(VL: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End >>>
40 1 pp 6v do j = 1, m
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< COLLAPSED
<<< Loop-information End >>>
41 2 p 6   do i = 1, n
42 2 p 6v   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 6v   enddo
44 1 p     enddo
    
```

Adding dummy array
between arrays to cause
shift from 16-KB
boundary



The percentage of L1D misses decreased from 29.46% to 3.57%, and the L1D miss dm percentage decreased too from 93.32% to 20.40%.

Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)
Before improvement	0.00%	29.46%	3.96E+09	93.32%
After improvement	0.00%	3.57%	4.80E+08	20.40%

You can achieve effects similar to source tuning by specifying the following compiler options.

Compiler options	Description of function
-Kcommonpad[=<i>N</i>] ($4 \leq N \leq 2,147,483,644$)	Specifies that a gap be created between areas for variables in a common block to increase the data cache use efficiency. If <i>N</i> is omitted, the compiler automatically determines the optimal value.

■ Use example (source code before improvement)

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

◆ Automatic selection of target arrays ➡ Application of padding

◆ Notes

- For separate compilation with the compiler options -Kcommonpad specified for a file containing a common block, this option must also be specified for other files containing common blocks of the same name.
- For compilation with the compiler options -Kcommonpad=*N* specified for multiple files, the value of *N* must be the same.
- Also, if programs with the compiler options -Kcommonpad specified use the same common block name with its elements changed, the programs may not run correctly.

Padding Using a Dummy Array (for Arrays of Different Sizes)

- Conflict between Arrays of Different Sizes
- Padding Using a Dummy Array
(for Arrays of Different Sizes: Before Improvement)
- Effects of Padding Using a Dummy Array
(for Arrays of Different Sizes: Source Tuning)

Conflict between Arrays of Different Sizes (1/2)

Generally, stationary cache thrashing does not occur for arrays of different sizes.

```

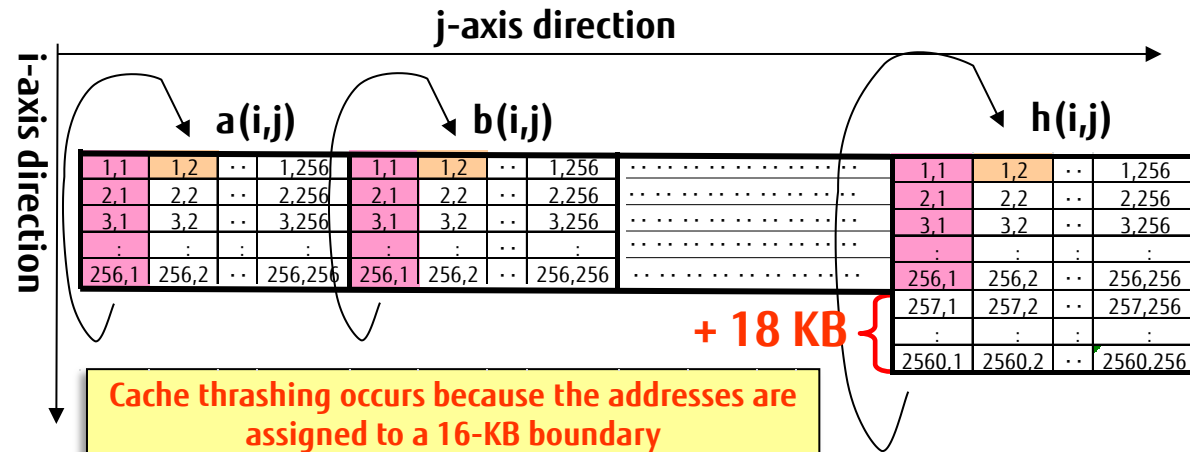
Source code example

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 because the addresses are assigned to a 16-KB boundary

Assuming that the array address of $a(1,1)$ is 0,
 the array address of $a(1,1)$ is 0 (16 KB \times 0),
 the array address of $b(1,1)$ is $256 \times 256 \times 8$ (16 KB \times 32),
 : : :
 the array address of $h(1,1)$ is $256 \times 256 \times 8 \times 7$ (16 KB \times 224)

Incrementing in the second dimension,
 the array address of $a(1,2)$ is the $a(1,1)$ address + 256×8 B,
 the array address of $b(1,2)$ is the $b(1,1)$ address + 256×8 B,
 : : :
 the array address of $h(1,2)$ is the $h(1,1)$ address + 2560×8 B

256×8 B + 18 KB

Generally, stationary cache thrashing does not occur for arrays of different sizes.

Arrays a , b , c , d , e , f , and g remain on a 16-KB boundary, but the array h address shifts from the 16-KB boundary.

Conflict between Arrays of Different Sizes (2/2)

Stationary cache thrashing occurs because an array remains on a 16-KB boundary. This happens even in cases with arrays of different sizes, depending on the array size.

```

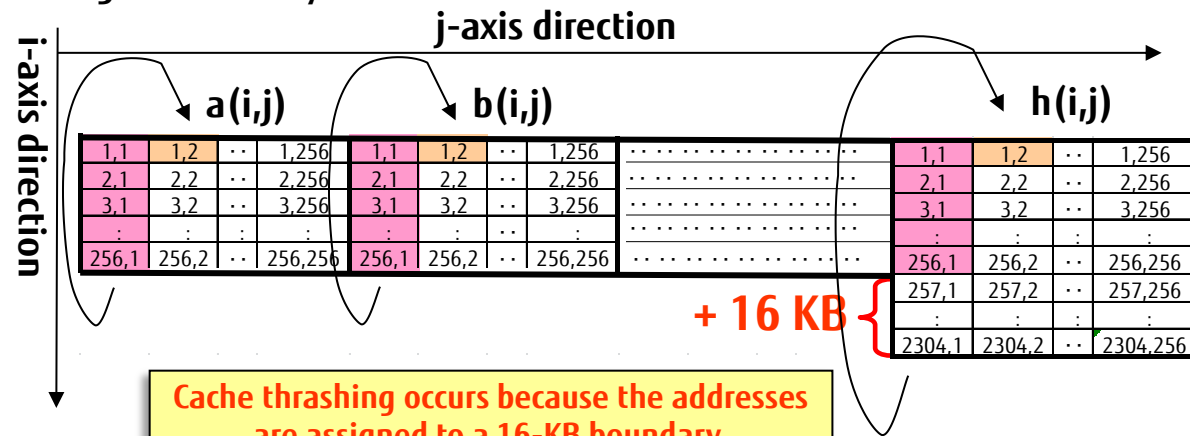
Source code example

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 because the addresses are assigned to a 16-KB boundary.

Assuming that the array address of $a(1,1)$ is 0,
 the array address of $a(1,1)$ is 0 (16 KB \times 0),
 the array address of $b(1,1)$ is 256 \times 256 \times 8 (16 KB \times 32),
 : : :
 the array address of $h(1,1)$ is 256 \times 256 \times 8 \times 7 (16 KB \times 224)

Incrementing in the second dimension,
 the array address of $a(1,2)$ is the $a(1,1)$ address + 256 \times 8 B,
 the array address of $b(1,2)$ is the $b(1,1)$ address + 256 \times 8 B,
 : : :
 the array address of $h(1,2)$ is the $h(1,1)$ address + 2304 \times 8 B

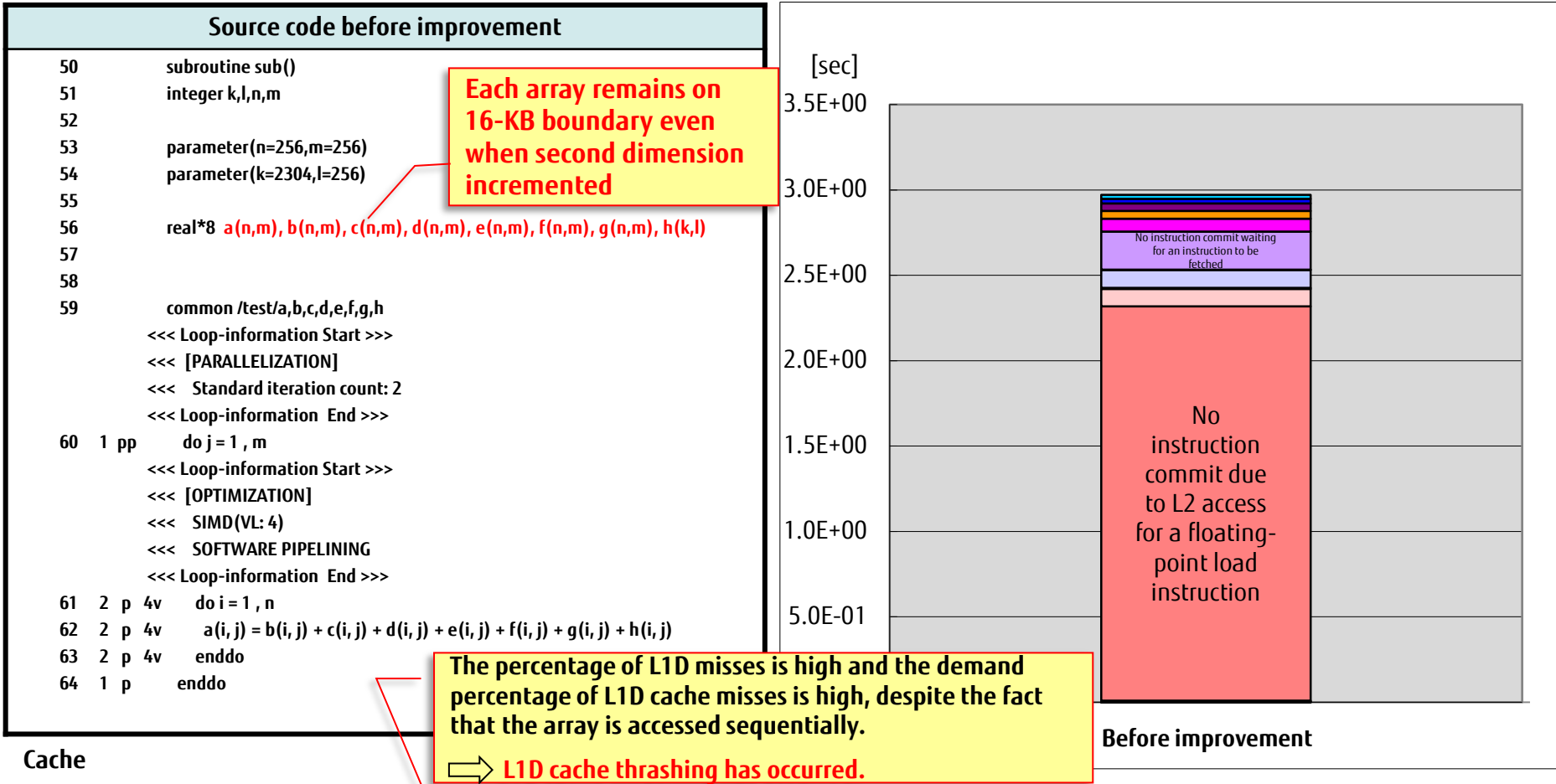
256 \times 8 B + 16 KB

Measures against thrashing are necessary for arrays of all sizes, including array h .

All of arrays a, b, c, d, e, f, g , and h remain on a 16-KB boundary.

Padding Using a Dummy Array (for Arrays of Different Sizes: Before Improvement)

L1D cache thrashing occurs because each array is located on a 16-KB boundary. Consequently, the following is a frequent event: No instruction commit due to L2 access for a floating-point load instruction.



Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	19.14%	2.57E+09	90.13%	9.87%	0.00%	0.00%	221.82	0.01

Padding Using a Dummy Array (for Arrays of Different Sizes: Source Tuning)

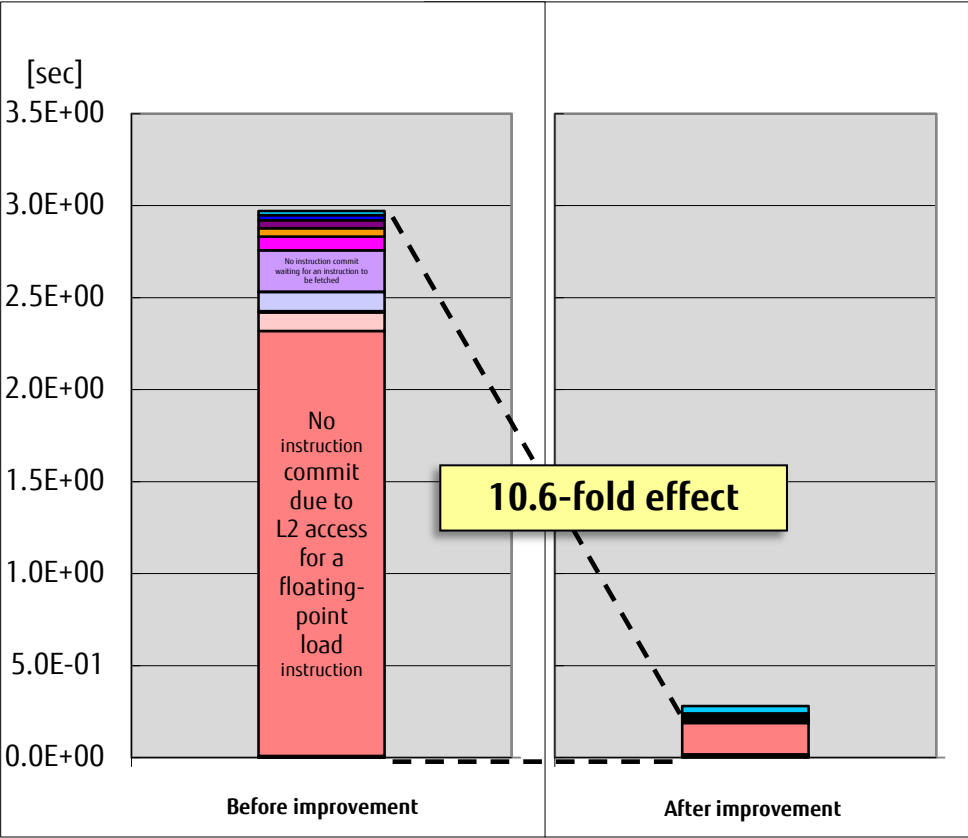
L1D cache thrashing was avoided because a dummy array was inserted between arrays to cause a shift from the 16-KB boundary. This results in improvement of the following event: No instruction commit due to L2 access for a floating-point load instruction.

Source code after improvement

```

50  subroutine sub()
51  integer k,l,n,m
52
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
62  <<< Loop-information Start >>>
63  <<< [PARALLELIZATION]
64  <<< Standard iteration count: 2
65  <<< Loop-information End >>>
66  1 pp do j = 1 , m
67  <<< Loop-information Start >>>
68  <<< [OPTIMIZATION]
69  <<< SIMD(VL: 4)
70  <<< SOFTWARE PIPELINING
71  <<< Loop-information End >>>
72  2 p 4v do i = 1 , n
73  2 p 4v a(i, j) = b(i, j) + c(i, j) + d(i, j) + e(i, j) + f(i, j) + g(i, j) + h(i, j)
74  2 p 4v enddo
75  1 p enddo
    
```

Inserting dummy array between arrays



Cache

The percentage of L1D misses decreased from 19.14% to 3.69%, and the L1D miss dm percentage decreased too from 90.13% to 27.02%.

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load- store instruction)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	19.14%	2.57E+09	90.13%	9.87%	0.00%	0.00%	221.82	0.01
After improvement	0.00%	3.69%	4.96E+08	27.02%	72.98%	0.00%	0.00%	452.07	0.04

Tuning Approach to Cache Thrashing (Application)

■ Tuning Approach (Application)

Tuning Approach (Application)

■ (array division +) array merging + dimensional displacement of an array

Original source

```
a(n,2,4)
b(n,2,4)
...
do i=1,n
  ... = a(i,1,2) + b(i,1,2) +
&      a(i,2,2) + b(i,2,2) +
&      a(i,1,3) + b(i,1,3) +
&      a(i,2,3) + b(i,2,3)
enddo
```

8 streams

Step 1 (array division)

```
a1(n,2),a2(n,2),a3(n,2),a4(n,2)
b1(n,2),b2(n,2),b3(n,2),b4(n,2)
...
do i=1,n
  ... = a2(i,1) + b2(i,1) +
&      a2(i,2) + b2(i,2) +
&      a3(i,1) + b3(i,1) +
&      a3(i,2) + b3(i,2)
enddo
```

8 streams

Step 2 (array merging)

```
a23(2,n,2)
b23(2,n,2)
...
do i=1,n
  ... = a23(1,i,1) + b23(1,i,1) +
&      a23(1,i,2) + b23(1,i,2) +
&      a23(2,i,1) + b23(2,i,1) +
&      a23(2,i,2) + b23(2,i,2)
enddo
```

4 streams

Step 3 (dimensional displacement of an array)

```
a23(2,2,n)
b23(2,2,n)
...
do i=1,n
  ... = a23(1,1,i) + b23(1,1,i) +
&      a23(1,2,i) + b23(1,2,i) +
&      a23(2,1,i) + b23(2,1,i) +
&      a23(2,2,i) + b23(2,2,i)
enddo
```

2 streams

Suppose dimensional displacement of an array is done based on the original source.

⇒ Cache efficiency will deteriorate because arrays a and b, which are used inside the loop, use only part of the areas of the declared sizes.

For this reason, preprocessing that is called array division (step 1) is done before array merging and dimensional displacement of an array (steps 2 and 3).

Following steps 1 to step 3 can **reduce the number of streams from eight to two.**

Improvement in TLB Thrashing

- What Is TLB Thrashing?
- Padding (Before Improvement)
- Effects of Padding (Source Tuning)
- Effects of Page Size Expansion (lpgparm Command)

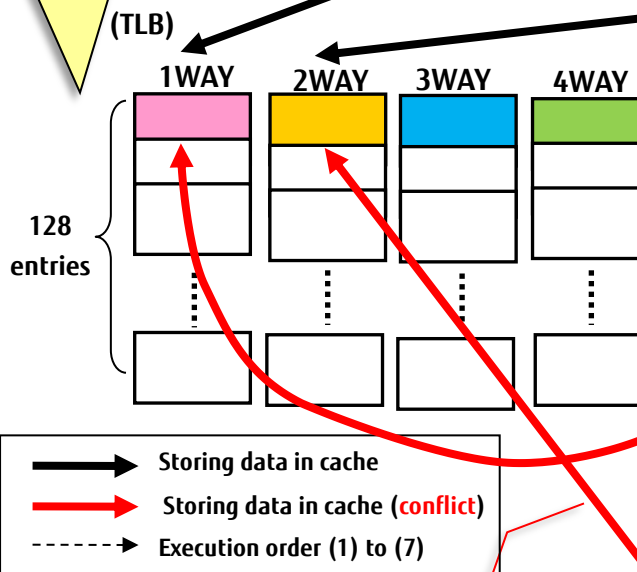
What Is TLB Thrashing?

TLB thrashing is a phenomenon in which address translation information for specific TLB indexes (TLB location information) is frequently overwritten. This phenomenon is likely to occur when the array size is a multiple of 512 MB. (* For details on the TLB, see "What Is the TLB? (Details)" in "Chapter 3 Large Page.")

Source code example (for a large page of 4 MB)

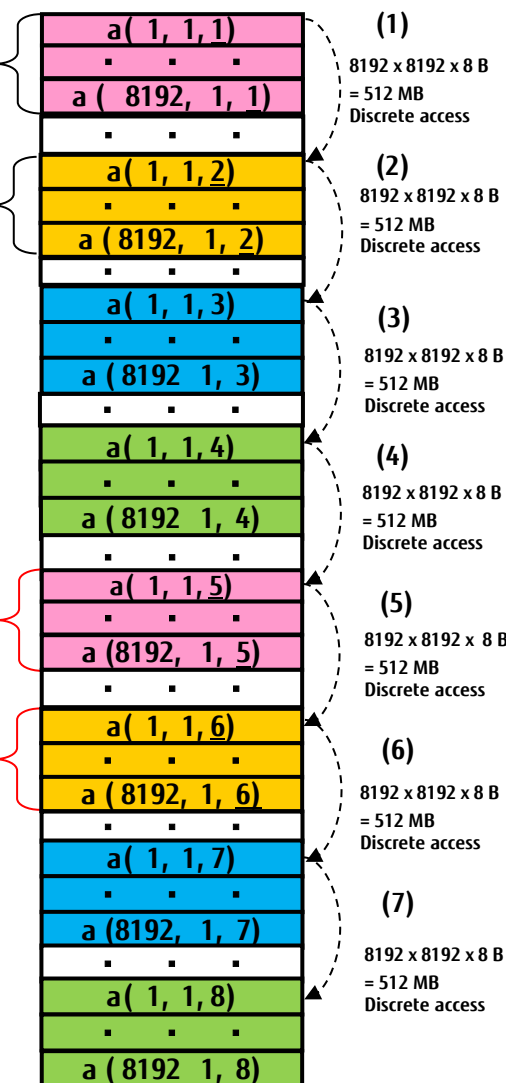
```
subroutine sub()
parameter(n=8192,m=8192)
real*8 a(n, m, 8)
common /com/a
do j = 1, m
  do i = 1, n
    a(i, j, 1) = a(i, j, 2) + a(i, j, 3) + a(i, j, 4) +
      a(i, j, 5) + a(i, j, 6) + a(i, j, 7) + a(i, j, 8)
  enddo
enddo
```

Address translation information for one page (4 MB)



(Page table)
* Stores address translation information

Execution order



Rough standard for TLB thrashing

mDTLB miss rate
(/Load-store instruction)

1.5% or higher

In this example, a(1,1,1), a(1,1,2), a(1,1,3), a(1,1,4), a(1,1,5), a(1,1,6), a(1,1,7), and a(1,1,8) are placed at an interval of 512 MB, so the eight of them are assigned to the same index. Therefore, the first and second points of data are overwritten by the fifth and sixth points of data, respectively.

Padding (Before Improvement)

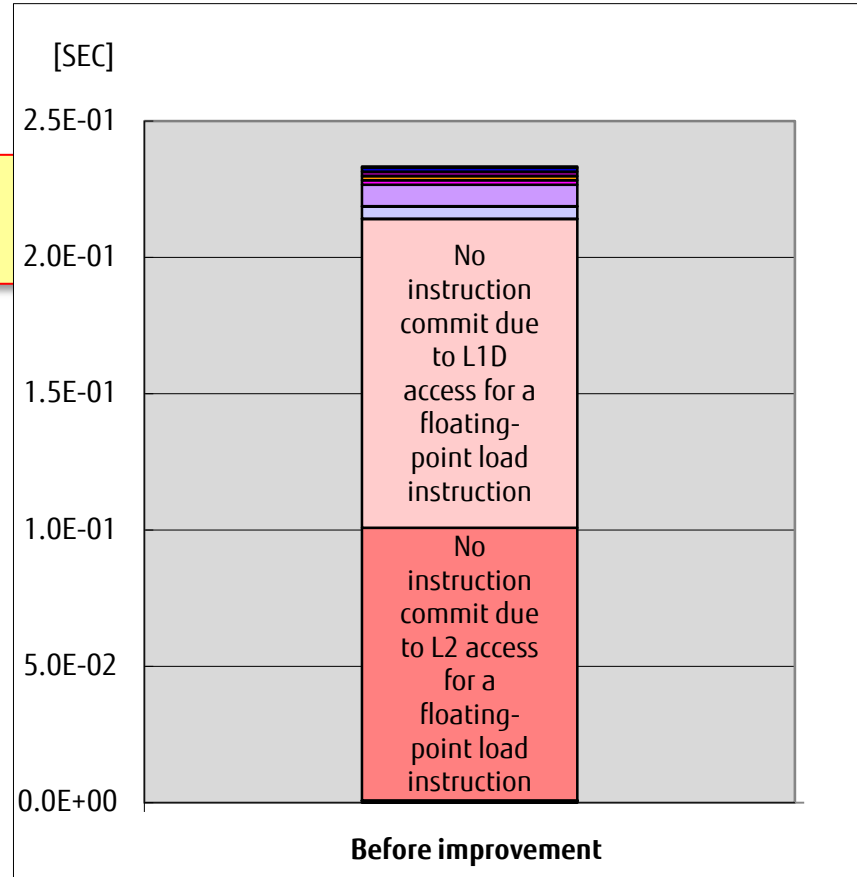
TLB thrashing occurs because the page size is 4 MB and each array is located on a 512-MB boundary. Consequently, data access wait is a frequent event.

Source code before improvement

```

27     parameter(n=8192,m=8192)
28     real*8 a(n, m, 8)
29     common /com/a
    <<< Loop-information Start >>>
    <<< [PARALLELIZATION]
    <<< Standard iteration count:
    <<< [OPTIMIZATION]
    <<< COLLAPSED
    <<< SIMD(VL: 4)
    <<< SOFTWARE PIPELINING
    <<< Loop-information End >>>
30  1 pp 6v do j = 1, m
    <<< Loop-information Start >>>
    <<< [OPTIMIZATION]
    <<< COLLAPSED
    <<< Loop-information End >>>
31  2 p 6 do i = 1, n
32  2 p 6v 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) + a(i, j, 8)
33  2 p 6v enddo
34  1 p enddo
    
```

8192 x 8192 x 8 B = 512 MB
(Page size of 4 MB x
128 entries)



Cache

	L2 throughput (GB/sec)	Memory throughput (GB/sec)	μDTLB miss rate (/Load-store instruction)	mDTLB miss rate (/Load-store instruction)
Before improvement	163.32	20.79	28.39728%	12.12630%

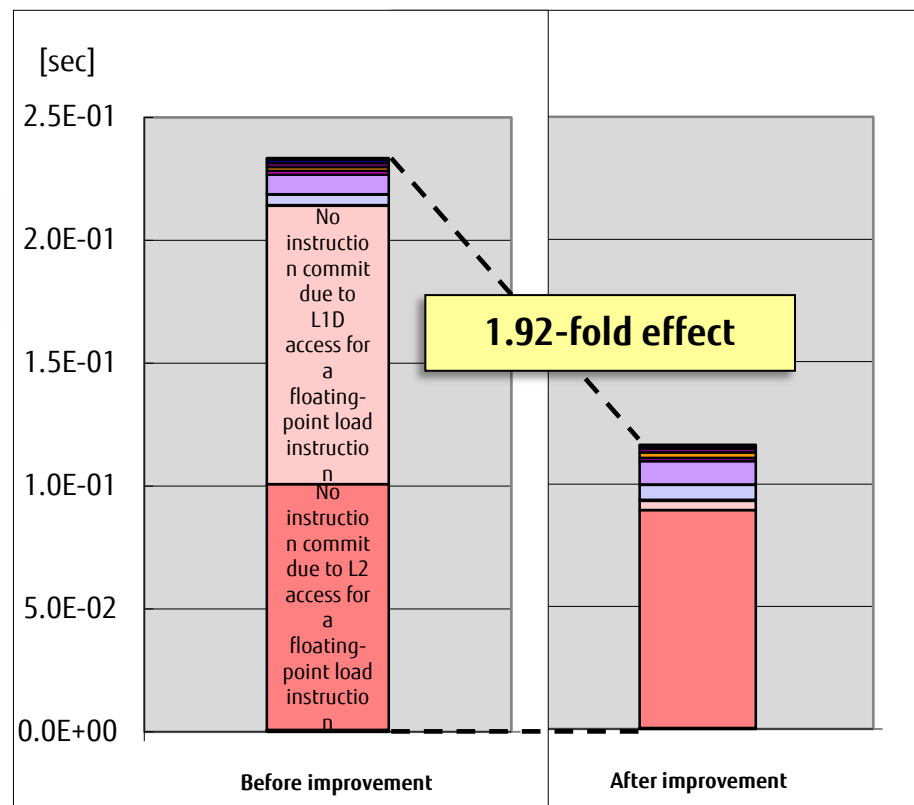
High percentage of mDTLB misses:
12.13%
⇒ TLB thrashing

Effects of Padding (Source Tuning)

TLB thrashing was avoided because the address of each stream was shifted through padding.
As a result, there was improvement in data access wait.

Source code after improvement (source tuning)	
27	parameter(n=8192+64,m=8192)
28	real*8 a(n, m, 8)
29	common /com/a
	<<< Loop-information Start
	<<< [PARALLELIZATION]
	<<< Standard iteration co
	<<< [OPTIMIZATION]
	<<< COLLAPSED
	<<< SIMD(VL: 4)
	<<< SOFTWARE PIPELINING
	<<< Loop-information End >>>
30	1 pp 6v do j = 1, m
	<<< Loop-information Start >>>
	<<< [OPTIMIZATION]
	<<< COLLAPSED
	<<< Loop-information End >>>
31	2 p 6 do i = 1, n
32	2 p 6v 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) + a(i, j, 8)
33	2 p 6v enddo
34	1 p enddo

Padding to shift address of each stream by 512 MB + one page (4 MB)



The percentage of mDTLB misses decreased to 0.00023%.

Cache

	L2 throughput (GB/sec)	Memory throughput (GB/sec)	μDTLB miss rate (/Load-store instruction)	mDTLB miss rate (/Load-store instruction)
Before improvement	163.32	20.79	28.39728%	12.12630%
After improvement	272.45	42.11	0.01764%	0.00023%

Effects of Page Size Expansion (lpgparm Command)

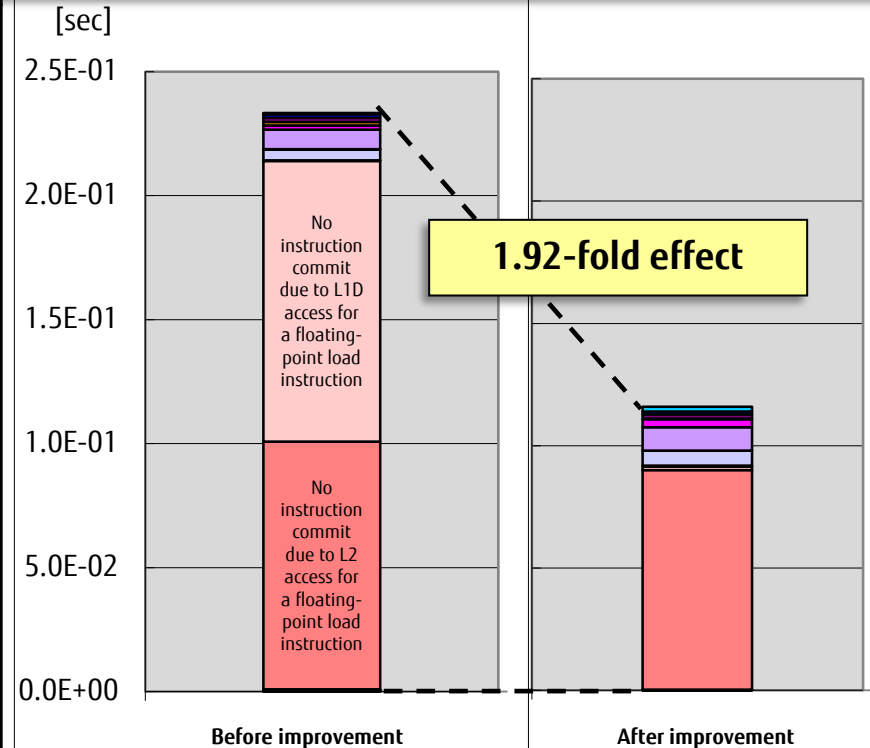
TLB thrashing was avoided because the page size was expanded to 32 MB by the `lpgparm` command.
As a result, there was improvement in data access wait.

Page size specification of 32 MB
\$ `lpgparm -s 32MB -t 32MB -d 32MB -h 32MB -p 32MB -S 32MB a.out`

Source code after improvement

```

27     parameter(n=8192,m=8192)
28     real*8 a(n, m, 8)
29     common /com/a
    <<< Loop-information Start >>>
    <<< [PARALLELIZATION]
    <<< Standard iteration count: 422
    <<< [OPTIMIZATION]
    <<< COLLAPSED
    <<< SIMD(VL: 4)
    <<< SOFTWARE PIPELINING
    <<< Loop-information End >>>
30  1 pp 6v do j = 1, m
    <<< Loop-information Start >>>
    <<< [OPTIMIZATION]
    <<< COLLAPSED
    <<< Loop-information End >>>
31  2 p 6 do i = 1, n
32  2 p 6v 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) + a(i, j, 8)
33  2 p 6v enddo
34  1 p enddo
    
```



The percentage of mDTLB misses decreased to 0.00005%.

Cache

	L2 throughput (GB/sec)	Memory throughput (GB/sec)	μDTLB miss rate (/Load-store instruction)	mDTLB miss rate (/Load-store instruction)
Before improvement	163.32	20.79	28.39728%	12.12630%
After improvement	270.29	41.83	0.00829%	0.00005%

Improvement in Data Access Wait (Increase in Data Locality)

- What Is Data Locality?
- Strip Mining
- Loop Blocking
- Sector Cache
- Loop Interchange
- Loop Fusion
- Array Merging (Indirect Access)



What Is Data Locality?

Data locality means the repeatedly accessing of data loaded in cache.

Higher data locality reduces memory access load, resulting in an improvement in data access wait.

The assumed model in these descriptions is an L2 cache (12 MB).

Conceptual diagrams of L2 cache (12-MB) states

 Overwritten data
 Data forced out

Source code example

```

real*8
a(n),b(n),c(n),d(n),e(n)
do i=1, n !! Loop 1
  a(i)=b(i)+c(i)
enddo
do j=1, n !! Loop 2
  e(i)=a(i)+d(i)
enddo
        
```

n = 1000000

Arrays a to e each have a data size of about 4 MB.

A cache miss occurs because the array a data loaded in cache by loop 1 was already forced out at the loop 2 execution time.



Improving the data locality of array a will increase cache efficiency.

Up to this point, no data is overwritten.

Data that can still be reused is forced out.

Old data is overwritten by new data and forced out of the cache.

A cache miss occurs because the data of a(1) was already forced out.

Loop 1, with i = 500000

b(1)	b(33)	...	b(499937)	b(499969)	4 MB
b(2)	b(34)	...	b(499938)	b(499970)	
...	
b(32)	b(64)	...	b(499968)	b(500000)	
c(1)	c(17)	...	c(499937)	c(499969)	4 MB
c(2)	c(18)	...	c(499938)	c(499970)	
...	
c(32)	c(64)	...	c(499968)	c(500000)	
a(1)	a(17)	...	a(499937)	a(499969)	4 MB
a(2)	a(18)	...	a(499938)	a(499970)	
...	
a(32)	a(64)	...	a(499968)	a(500000)	

Loop 1, with i = 500001

b(1)	b(500001)	b(33)	...	b(499937)	b(499969)
b(2)	b(500002)	b(34)	...	b(499938)	b(499970)
...
b(32)	b(500032)	b(64)	...	b(499968)	b(500000)
c(1)	c(500001)	c(33)	...	c(499937)	c(499969)
c(2)	c(500002)	c(34)	...	c(499938)	c(499970)
...
c(32)	c(500032)	c(64)	...	c(499968)	c(500000)
a(1)	a(500001)	a(33)	...	a(499937)	a(499969)
a(2)	a(500002)	a(34)	...	a(499938)	a(499970)
...
a(32)	a(500032)	a(64)	...	a(499968)	a(500000)

Loop 2, with j = 1

b(500001)	d(1)	b(500033)	...	b(499937)	b(499969)
b(500002)	d(2)	b(500034)	...	b(499938)	b(499970)
...
b(500032)	d(32)	b(500064)	...	b(499968)	b(500000)
c(500001)	e(1)	c(500033)	...	c(499937)	c(499969)
c(500002)	e(2)	c(500034)	...	c(499938)	c(499970)
...
c(500032)	e(16)	c(500064)	...	c(499968)	c(500000)
a(500001)	a(1)	a(500001)	...	a(499937)	a(499969)
a(500002)	a(2)	a(500002)	...	a(499938)	a(499970)
...
a(500032)	a(16)	a(500064)	...	a(499968)	a(500000)

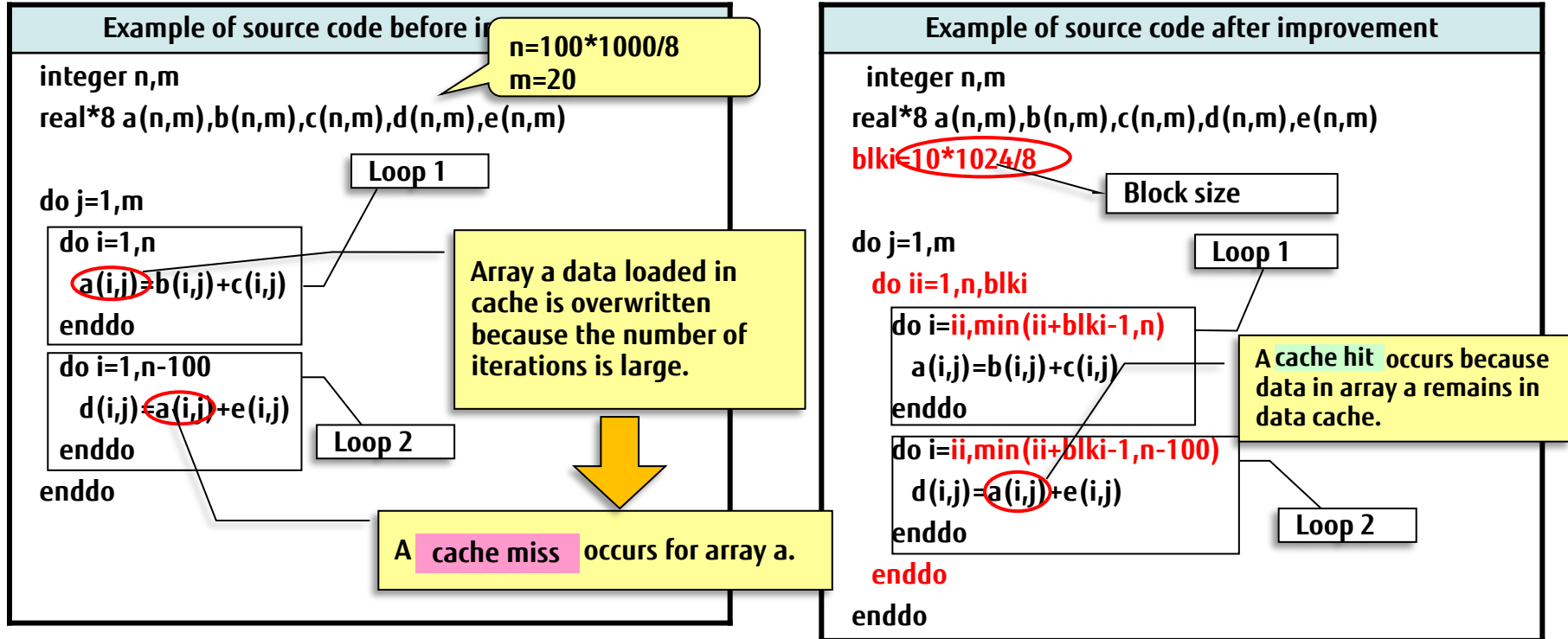
Cache miss

Strip Mining

- What Is Strip Mining?
- Strip Mining (Before Improvement)
- Effects of Strip Mining (Source Tuning)

What Is Strip Mining?

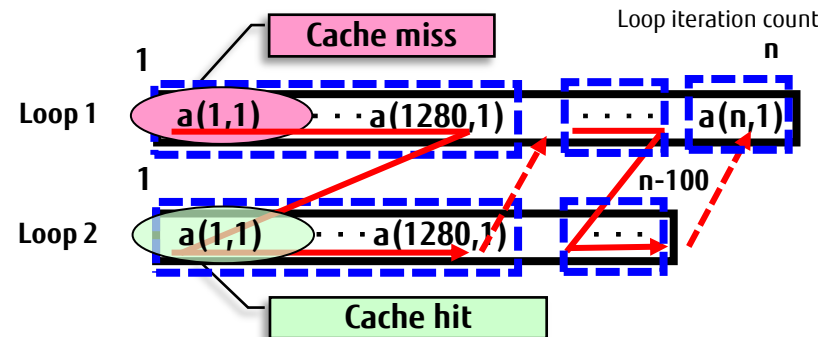
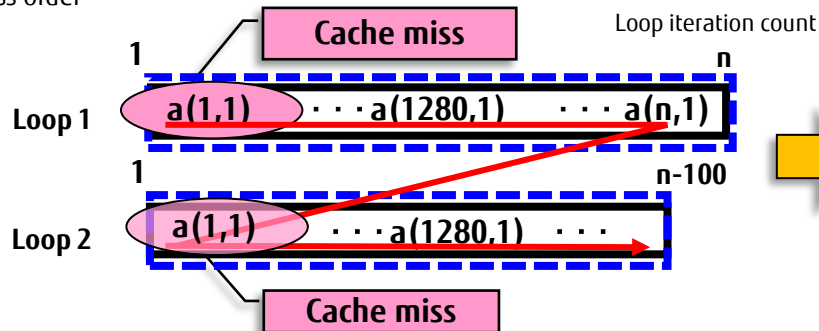
Strip mining is a technique for increasing cache efficiency through alternating execution, in units of blocks, of two loops nested at the same level.



→ Array access order

Block size

Array size



Strip Mining (Before Improvement)

Not all array data can be loaded in cache because loop 1 has many iterations, so loop 2 cannot reuse the data. Consequently the following is a frequent event: No instruction commit due to memory and cache busy.

Source code before improvement

```

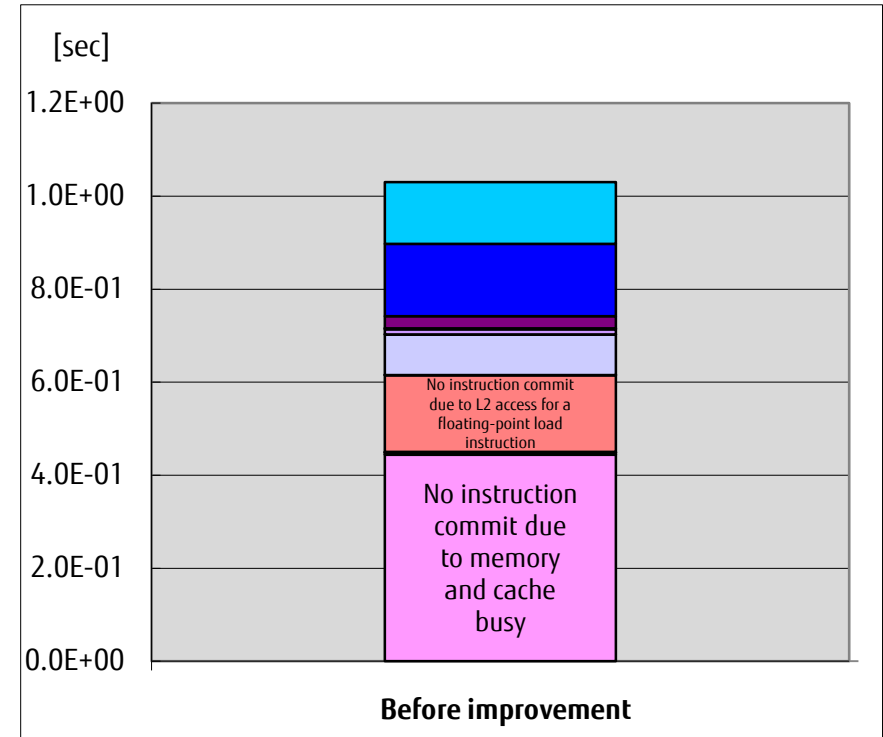
32      !$omp parallel do reduction(+:s1)
33  1 p    do j=1,m
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< SOFTWARE PIPELINING
      <<< Loop-information End >>>
34  2 p 4v  do i=1,n
35  2 p 4v    s1 = s1 + a(i,j) / (s3 / b(i,j) + c(i,j) / (s2 + s3 / d(i,j)))
36  2 p 4v  enddo
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< SOFTWARE PIPELINING
      <<< Loop-information End >>>
37  2 p 4v  do i=1,n-100
38  2 p 4v    e(i,j) = s2 / (a(i,j) + b(i,j) / (s3 + c(i,j) / d(i,j)))
39  2 p 4v  enddo
40  1 p    enddo
        
```

Loop iteration count: 375000
Total of array sizes: 12 MB
Not all array data can be loaded in cache because the number of iterations is large.

Array access resulting in cache miss

Loop 1

Loop 2



The percentages of L1D misses and L2 misses are around 3.125%, which is the theoretical value for stream access, because data cannot be reused.

Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load- store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.01%	3.13%	4.22E+08	0.18%	99.82%	0.00%	3.13%	4.23E+08	104.86	116.67

Effects of Strip Mining (Source Tuning)

Strip mining increases cache efficiency, which improves the following event: No instruction commit due to memory and cache busy.

Source code after improvement (source tuning)

```

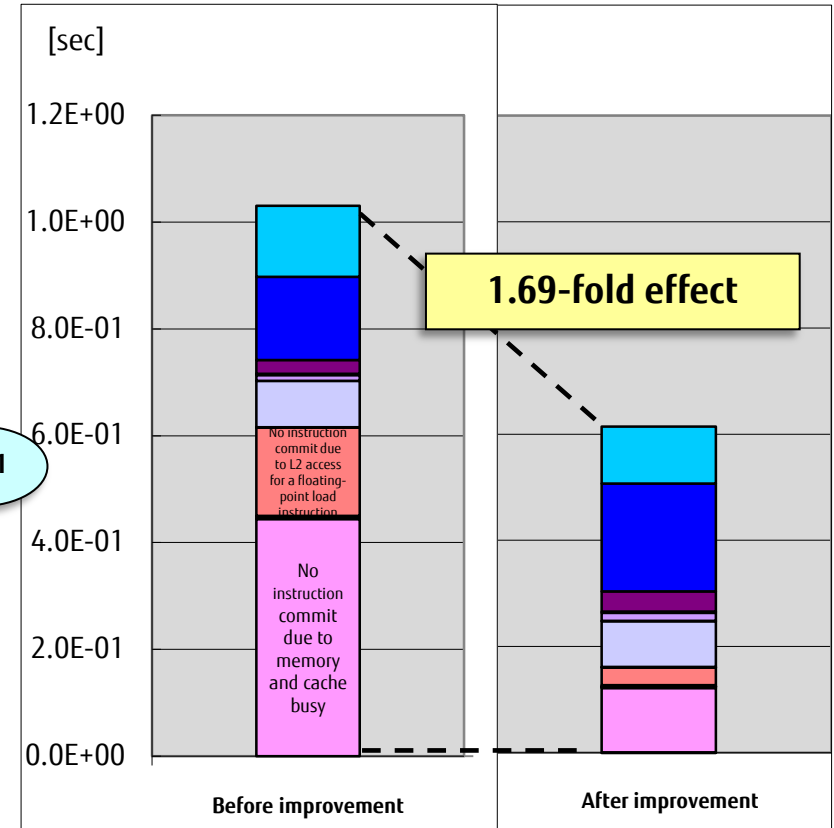
34      blki=4*1024/8
35
36      !$omp parallel do reduction(+:s1)
37  1 p    do j=1,m
38  2 p      do ii=1,n,blki
39  3 p  4v    do i=ii,min(ii+blki-1,n)
40  3 p  4v      s1 = s1 + a(i,j) / (s3 / b(i,j) + c(i,j) / (s2 + s3 / d(i,j)))
41  3 p  4v    enddo
42  3 p  4v    do i=ii,min(ii+blki-1,n-100)
43  3 p  4v      e(i,j) = s2 / (a(i,j) + b(i,j) / (s3 + c(i,j) / d(i,j)))
44  3 p  4v    enddo
45  2 p      enddo
46  1 p    enddo
  
```

Block size: 4 KB
4 KB x 4 streams = 16 KB
⇒ Size for placing data in L1 cache

Array access resulting in cache hit

Loop 1

Loop 2



The numbers of L1D misses and L2 misses decreased significantly.

Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate (/L1D miss)	L1D miss hwpf rate (/L1D miss)	L1D miss swpf rate (/L1D miss)	L2 miss rate (/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.01%	3.13%	4.22E+08	0.18%	99.82%	0.00%	3.13%	4.23E+08	104.86	116.67
After improvement	0.01%	1.73%	2.35E+08	0.33%	99.67%	0.00%	1.74%	2.35E+08	97.94	117.69

Loop Blocking

- What Is Loop Blocking?
- Loop Blocking (Before Improvement)
- Effects of Loop Blocking (Source Tuning)

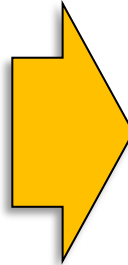
Loop Blocking (1/3)

Loop blocking is a technique for increasing cache efficiency. This technique divides source code into blocks of the specified size before execution.

Example of source code 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 of source code 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 of 1 array
(= 96 x 16 x 8 B)

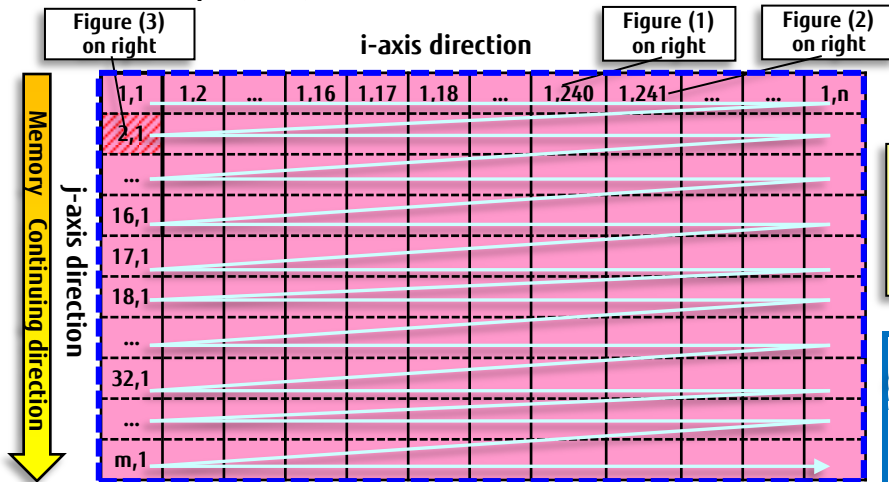
Loop Blocking (2/3)

■ Array access (before improvement)

Memory is accessed every time i is updated because of the stride access of array a .

This results in the data loaded in the cache by access to $a(1,1)$ being forced out before access at the $a(2,1)$ access time.

Array $a(m,n)$ access order and cache miss status



L1 cache (64 KB) state

 Overwritten data
 Data forced out

(1) $j = 1$ and $i = 240$

$a(1,1)$	$a(1,2)$	$a(1,3)$	$a(1,240)$
$a(2,1)$	$a(2,2)$	$a(2,3)$	$a(2,240)$
...
$a(32,1)$	$a(32,2)$	$a(32,3)$	$a(32,240)$

Up to this point, no data is overwritten.

Data that can still be reused is forced out.

(2) $j = 1$ and $i = 241$

$a(1,1)$	$a(1,241)$	$a(1,2)$	$a(1,3)$	$a(1,240)$
$a(2,1)$	$a(2,241)$	$a(2,2)$	$a(2,3)$	$a(2,240)$
...
$a(32,1)$	$a(32,241)$	$a(32,2)$	$a(32,3)$	$a(32,240)$

Old data is overwritten by new data and forced out of the cache.

(3) $j = 2$ and $i = 1$

$a(1,241)$	$a(1,1)$	$a(1,242)$	$a(1,243)$	$a(1,480)$
$a(2,241)$	$a(2,1)$	$a(2,242)$	$a(2,243)$	$a(2,480)$
...
$a(32,241)$	$a(32,1)$	$a(32,242)$	$a(32,243)$	$a(32,480)$

Cache miss

A cache miss occurs because the data of $a(2,1)$ was already forced out.

Example of source code (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 access order
- Block size
- Cache line
- Cache miss
- Cache hit

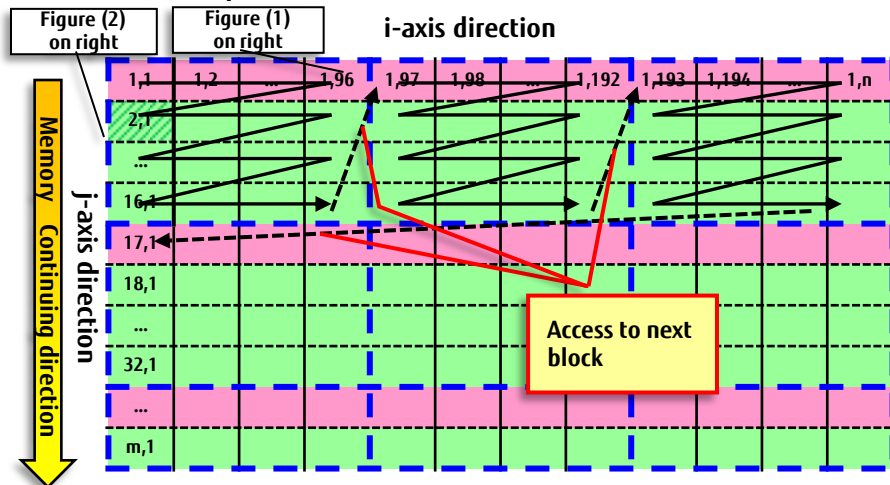
Loop Blocking (3/3)

■ Array access (after improvement) with a block size of 96 x 16

Loop blocking causes block-by-block access.

This results in a cache hit during access of $a(2,1)$ and increased cache efficiency.

Array $a(m,n)$ access order and cache miss status



Example of source code (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 order
- Block size
- Cache line
- Cache miss
- Cache hit

L1 cache (64 KB) state

(1) $j = 1$ and $i = 96$

$a(1,1)$	$a(1,2)$	$a(1,3)$...	$a(1,96)$		
$a(2,1)$	$a(2,2)$	$a(2,3)$...	$a(2,96)$		
:	Array data placed in cache			:		
:				:		
$a(32,1)$	$a(32,2)$	$a(32,3)$...	$a(32,96)$		

(2) $j = 2$ and $i = 1$

$a(1,1)$	$a(1,2)$	$a(1,3)$...	$a(1,96)$		
$a(2,1)$	$a(2,2)$	$a(2,3)$...	$a(2,96)$		
:				:		
:				:		
$a(32,1)$	$a(32,2)$	$a(32,3)$...	$a(32,96)$		

A cache hit occurs because data remains in cache.

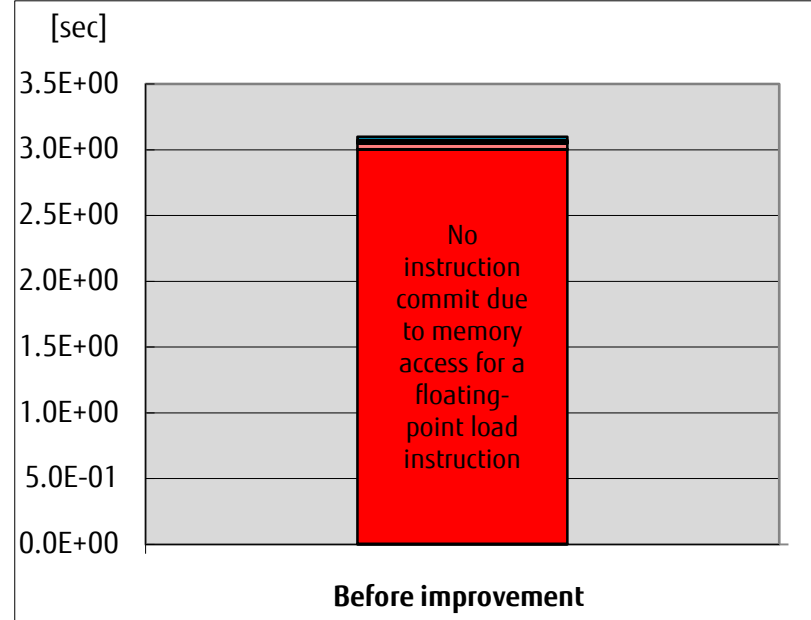
Loop Blocking (Before Improvement)

Cache use efficiency decreases because of stride access of array a, and the following event occurs: No instruction commit due to memory access for a floating-point load instruction.

Source code before improvement

```

48 1  !$omp do
49 2  p    do j=1,n2
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< SOFTWARE PIPELINING
      <<< Loop-information End >>>
50 3  p 6v    do i=1,n1
51 3  p 6v    b(i,j) = c0 + a(j,i)*(c1 + a(j,i)*(c2 + a(j,i)*(c3 + a(j,i)*
52 3      &    (c4 + a(j,i)*(c5 + a(j,i)*(c6 + a(j,i)*(c7 + a(j,i)*
53 3      &    (c8 + a(j,i)*c9))))))
54 3  p 6v    enddo
55 2  p    enddo
56 1  !$omp enddo
57 1  enddo
58  !$omp end parallel
    
```



Data in array a is placed in the cache once during the i iteration, but the data is already forced out by the time of the next j iteration. Consequently, a cache miss occurs.

Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate (/L1D miss)	L1D miss hwfp rate (/L1D miss)	L1D miss swfp rate (/L1D miss)	L2 miss rate (/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.09%	51.31%	1.28E+09	96.98%	3.02%	0.00%	51.34%	1.28E+09	106.04	109.34

The percentages of L1D misses and L2 misses are high at 51%.

Effects of Loop Blocking (Source Tuning)

Reuse of data in array a through loop blocking increases cache efficiency, which improves the following event:
No instruction commit due to memory access for a floating-point load instruction.

Source code after improvement (after source tuning)

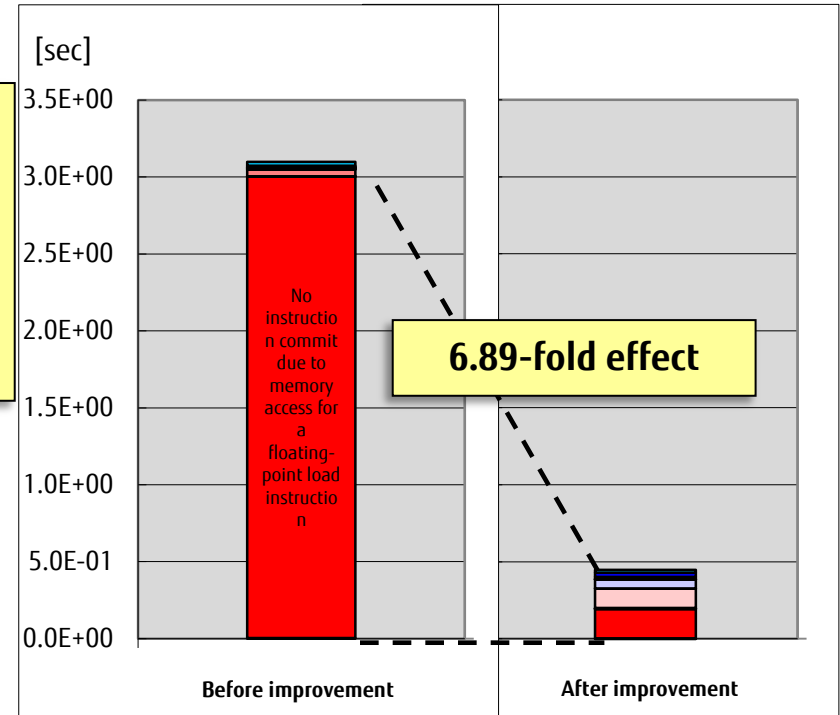
```

55 1  !$omp do
56 2 p  do jj=1,n1,16
57 3 p  do ii=1,n1,96
58 4 p  do j=jj,min(jj+16-1,n1)
    <<< Loop-information Start
    <<< [OPTIMIZATION]
    <<< SIMD(VL: 4)
    <<< SOFTWARE PIPELINING
    <<< Loop-information End
59 5 p 6v do i=ii,min(ii+96-1,n1)
60 5 p 6v b(i,j) = c0 + a(j,i)*(c1 + a(j,i)*(c2 + a(j,i)*(c3 + a(j,i)*
61 5      & (c4 + a(j,i)*(c5 + a(j,i)*(c6 + a(j,i)*(c7 + a(j,i)*
62 5      & (c8 + a(j,i)*c9))))))
63 5 p 6v enddo
64 4 p  enddo
65 3 p  enddo
66 2 p  enddo
67 1  !$omp enddo
  
```

Application of loop blocking

The L1 cache size is 64 KB, so the following is assumed in the cache: the size of 1 block is 12 KB (96 x 16 x 8), and the size required for processing 1 block is 24 KB (12 x 2 blocks).

This is intended to improve the data use efficiency of the cache.



Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load- store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.09%	51.31%	1.28E+09	96.98%	3.02%	0.00%	51.34%	1.28E+09	106.04	109.34
After improvement	0.01%	6.69%	1.70E+08	95.65%	4.35%	0.00%	6.12%	1.56E+08	97.61	111.78

The percentages of L1D misses and L2 misses decreased significantly.

Sector Cache

- What Is a Sector Cache?
- Overview of Sector Cache Capacity Control
- Conceptual Diagram of Actual Operation
- How to Use a Sector Cache
- Sector Cache Improvement Example
- Sector Cache: Case Example 1
- Sector Cache: Case Example 2

What Is a Sector Cache?

**A sector cache is a cache mechanism that can prevent reusable data from being forced out of the cache by non-reusable data. This mechanism enables applications to divide the cache into two parts (sector 0 and sector 1) and use them.
(Reused arrays use sector 1, and the others use sector 0.)**

The following sector cache details assume a model of a 5-MB/10-way L2 cache.

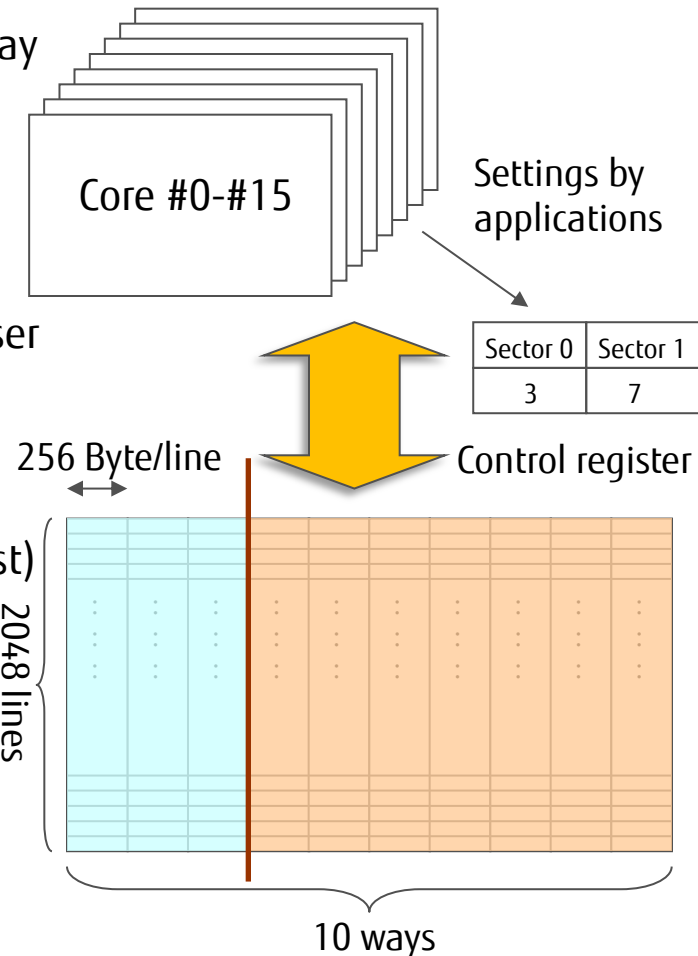
(The SPARC64™ XIfx has a 12-MB/24-way/1-CMG L2 cache.)

- The capacity of each sector is specified by the number of ways.
- Functionally, capacity values are interpreted as target values. Under control by hardware, the capacity of each sector gets closer to the specified capacity at the line replacement time.

⇒ Data is not forcibly made invalid even if the capacity is exceeded.

- The LRU algorithm (the least recently used data is discarded first) controls the forcing out of data from a sector.
- Applications determine the usage of sector 0 and sector 1. However, sector 0 stores a series of instructions.

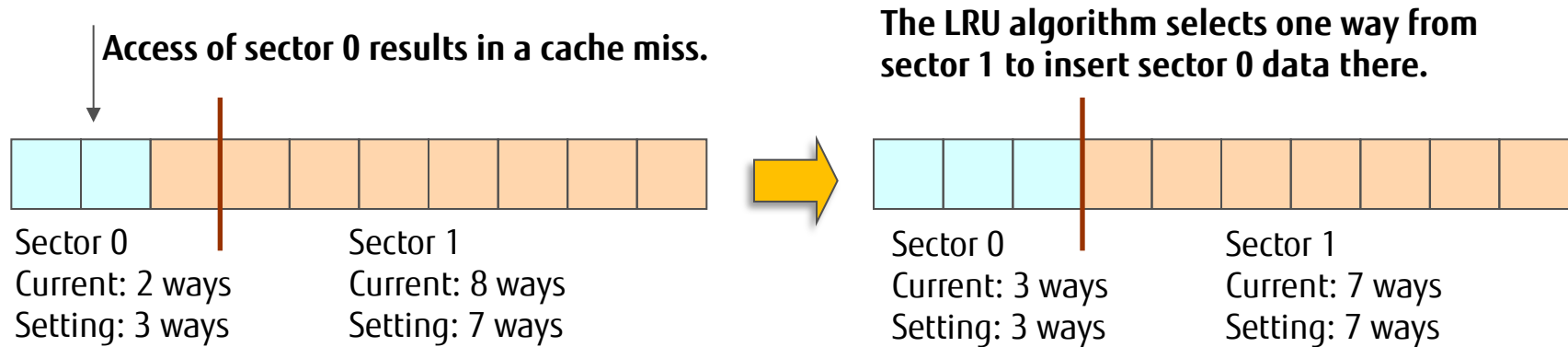
The next and subsequent pages provide an overview of capacity control in cases where sector 0 = 3 ways and sector 1 = 7 ways.



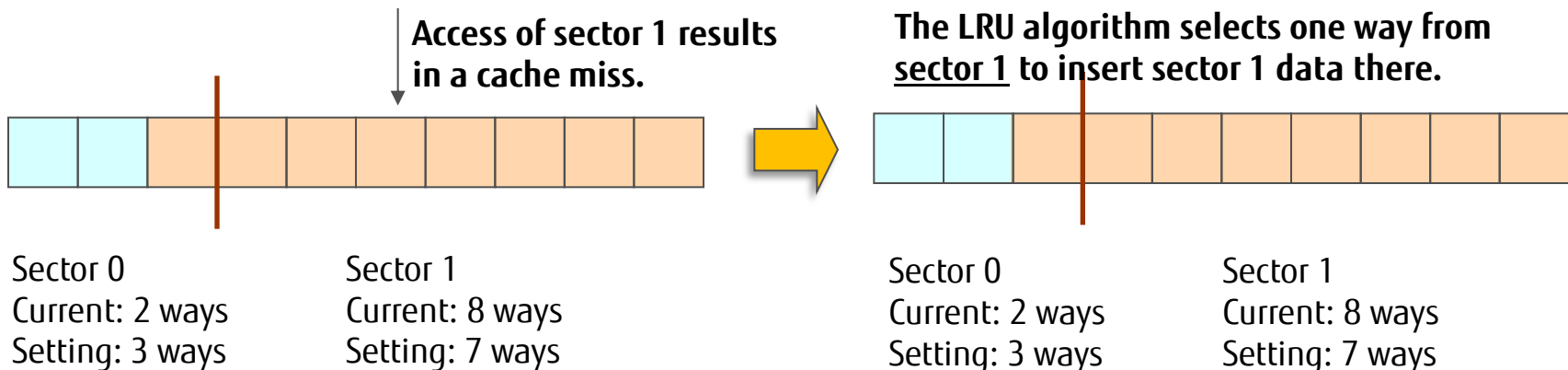
Overview of Sector Cache Capacity Control (1/2)

■ A cache miss is an opportunity to adjust the capacity. The capacity is not forcibly adjusted.

- If the capacity of a sector is less than the capacity specified in the control register, the number of ways is increased until the capacity is reached.



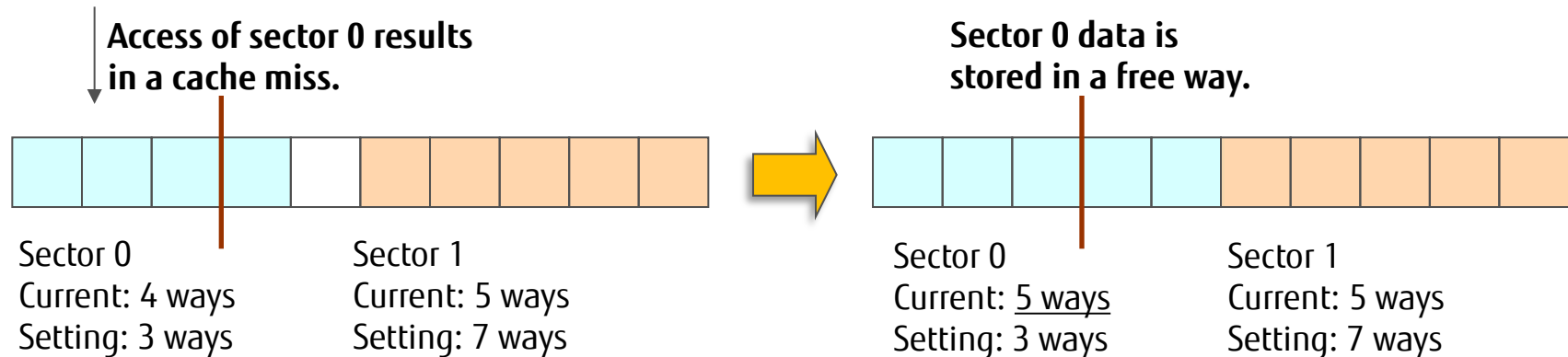
- Even if the sector of a cache miss has a greater capacity than specified, the capacity does not decrease.



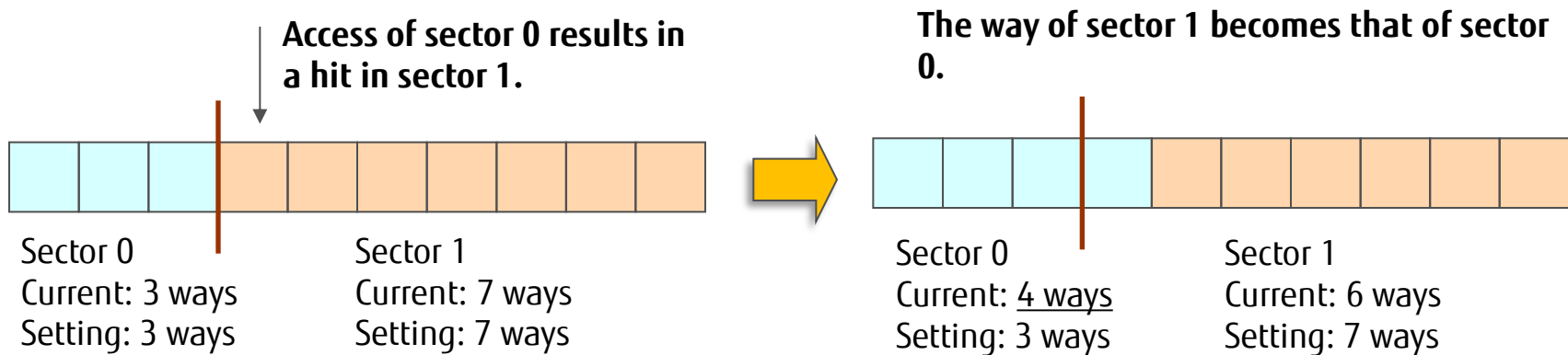
Overview of Sector Cache Capacity Control (2/2)

- The number of ways of a sector may exceed the specified capacity.

1. Case with a free way

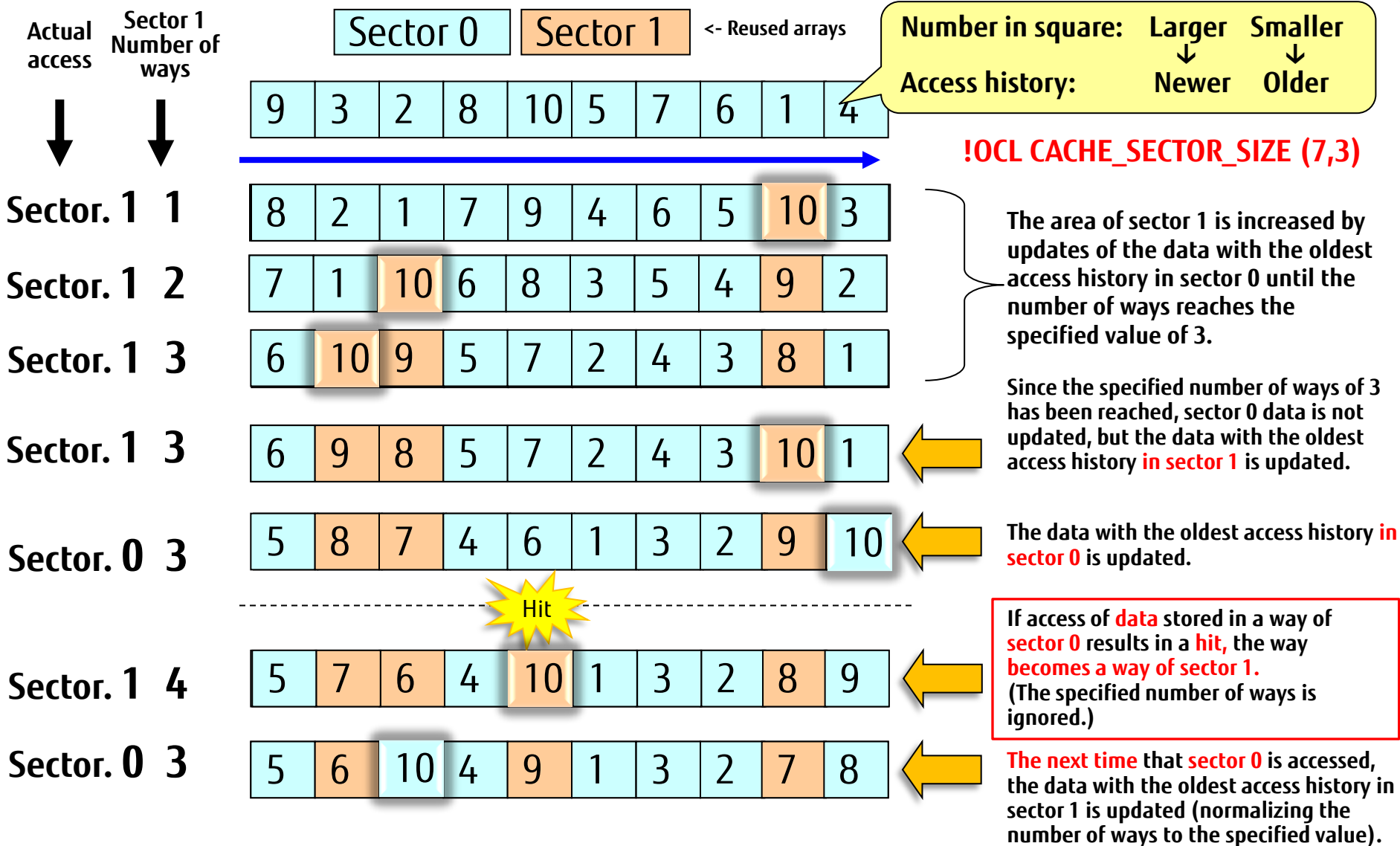


2. Case of a hit in access of another sector



Conceptual Diagram of Actual Operation

(Sector 0: 7 Ways; Sector 1: 3 Ways)

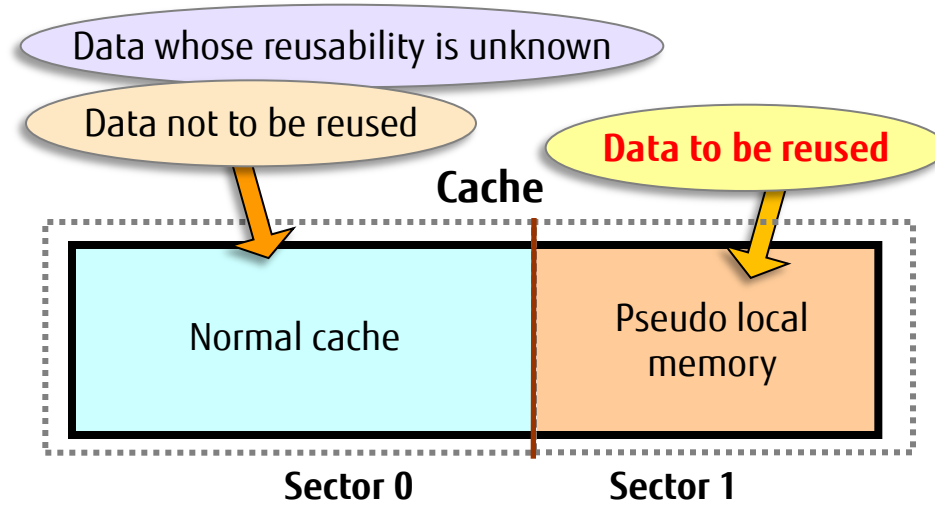


How to Use a Sector Cache (1/2)

- **Sector cache: Pseudo local memory**

Software can use sectors effectively according to the reusability of data.

- Reused arrays \Rightarrow **Sector 1 used**
- Others \Rightarrow **Sector 0 used**
- Data on sector 1 is not forced out by other data.
- The user can specify in a directive line that the array be in sector 1.



Example of using compiler directive lines for sector cache specification

```
!OCL CACHE_SECTOR_SIZE(3,7)  
!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>

The purpose is to prevent array a, which has reusability, from being forced out of the cache by access to arrays b and c in a loop.

How to Use a Sector Cache (2/2)

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

Optimization control specifiers	Meaning	Optimization control line that can be specified			
		Program unit	DO loop unit	Statement unit	Array assignment statement unit
CACHE_SECTOR_SIZE (l1_n1,l1_n2,l2_n1,l2_n1) END_CACHE_SECTOR_SIZE	Specifies the maximum numbers of ways of sector 0 and ways of sector 1 in the L1 cache and L2 cache.	Yes	No	Yes	No
CACHE_SECTOR_SIZE (l2_n1,l2_n2) END_CACHE_SECTOR_SIZE	Specifies the maximum number of ways of sector 0 and the maximum number of ways of sector 1 in the L2 cache.	Yes	No	Yes	No
CACHE_SUBSECTOR_ASSIGN(array 1[,array2...]) END_CACHE_SUBSECTOR	Specifies the array to place in sector 1 of the cache.	Yes	No	Yes	No

Sector Cache Improvement Example (1/2)

In this example, reusable data in array b is forced out of the cache, resulting in a cache miss. The assumed model in the descriptions is a 6-MB/12-way L2 cache.

Source code before improvement

```
subroutine sub(s)
parameter(n=4*1024*1024, m=9*512*1024/8)
real*8 a(n), b(m), s
integer*8 c(n)
real*8 dummy1(140), dummy2(140)
common /data/a, dummy1, c, dummy2, b

do i=1, n
  a(i) = a(i) + s * b(c(i))
enddo
end
```

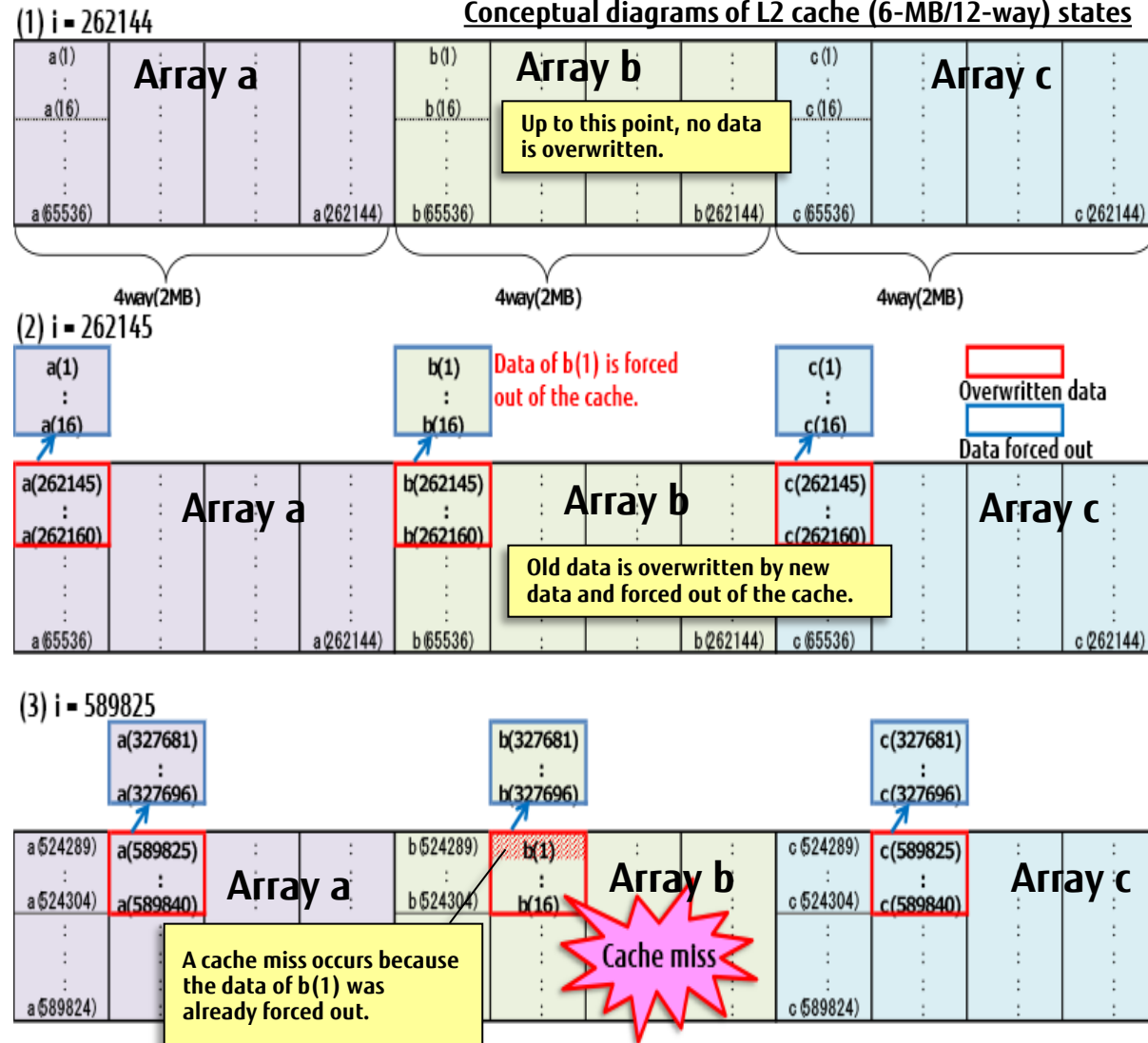
[Data amount of each array]
Array a: 32 MB
Array b: 4.5 MB
Array c: 16 MB

Access of array b]

Array b has reusability since sequential access from b(1) to b(589824) is repeated seven times.

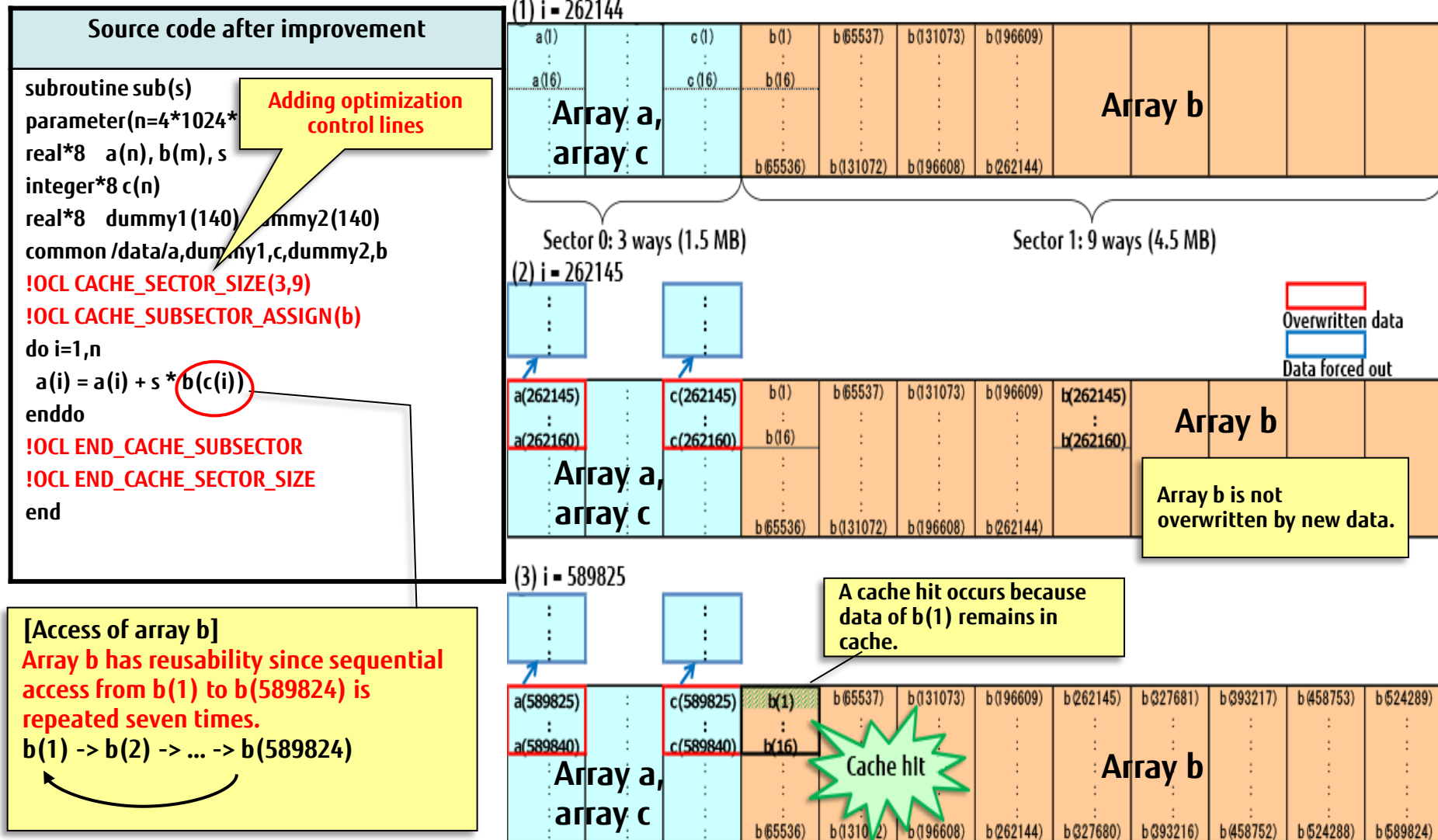
b(1) -> b(2) -> ... -> b(589824)

Conceptual diagrams of L2 cache (6-MB/12-way) states



Sector Cache Improvement Example (2/2)

The following example shows a way to prevent reusable data in array b from being forced out of the cache.



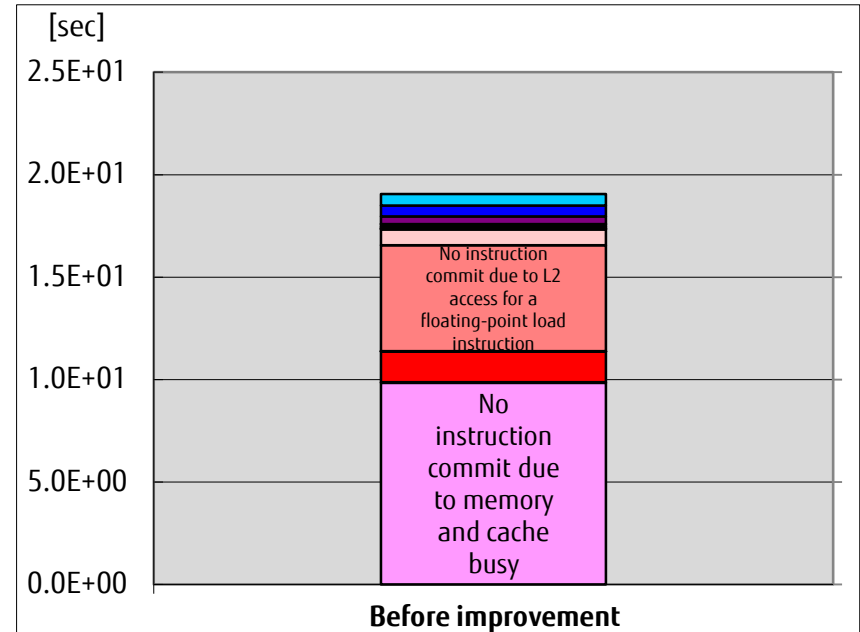
Sector Cache: Case Example 1 (Before Improvement)

Data in array b cannot be reused because it has been forced out of the cache. Consequently, the following is a frequent event: No instruction commit due to memory and cache busy.

Source code before improvement

```

63      parameter(n=8*1024*1024, m=17*512*1024/8)
64      real*8  a(n), b(m), s
65      integer*8 c(n)
66      real*8  dummy1(140), dummy2(140)
67      common /data/a, dummy1, c, dummy2, b
68
    <<< Loop-information Start >>>
    <<< [PARALLELIZATION]
    <<< Standard iteration count: 762
    <<< [OPTIMIZATION]
    <<< SIMD(VL: 4)
    <<< SOFTWARE PIPELINING
    <<< Loop-information End >>>
69  1 pp 8v      do i=1,n
70  1 p 8v      a(i) = a(i) + s * b(c(i))
71  1 p 8v      enddo
    
```



The percentage of L2 cache misses is high.

A memory throughput bottleneck has occurred.

Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load- store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.01%	2.78%	9.35E+09	18.29%	81.71%	0.00%	2.35%	7.87E+09	125.54	140.97

Sector Cache: Case Example 1 (After Improvement)

Placing array b in sector 1 increases cache efficiency, which improves the following event: No instruction commit due to memory and cache busy.

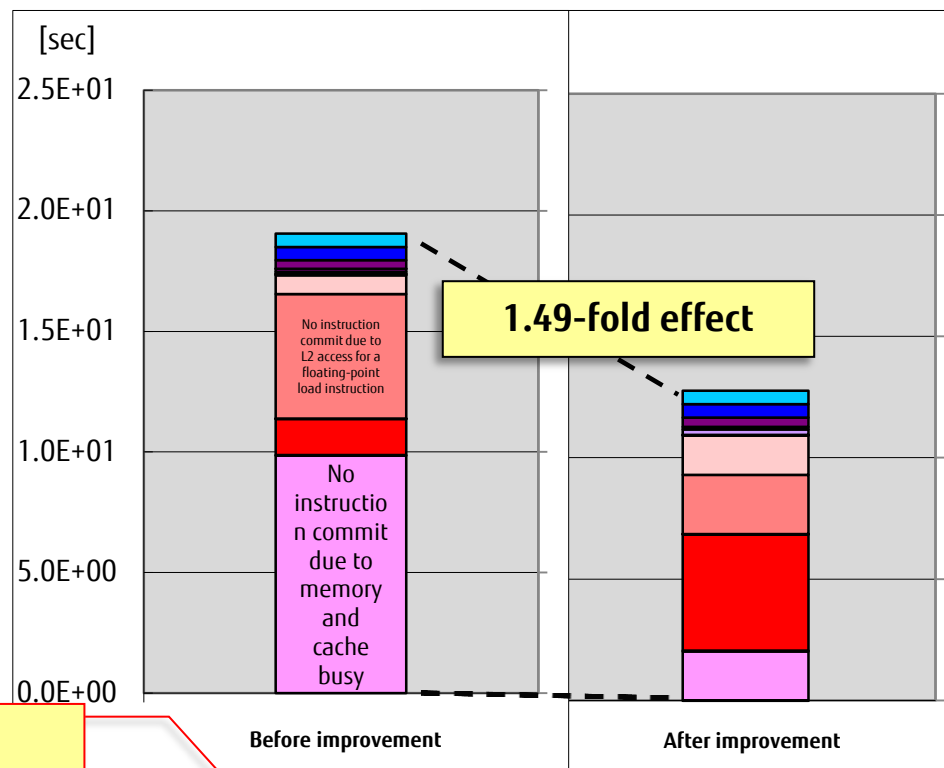
Source code after improvement (optimization control line tuning)

```

59      parameter(n=8*1024*1024, m=17*512*1024/8)
60      real*8  a(n), b(m), s
61      integer*8 c(n)
62      real*8  dummy1(140), dummy2(140)
63      common /data/a,dummy1,c,dummy2,b
64
65      !OCL CACHE_SECTOR_SIZE(6,18)
66      !OCL CACHE_SUBSECTOR_ASSIGN(b)
67
68      <<< Loop-information Start >>>
69      <<< [PARALLELIZATION]
70      <<< Standard iteration count: 762
71      <<< [OPTIMIZATION]
72      <<< SIMD(VL: 4)
73      <<< SOFTWARE PIPELINING
74      <<< Loop-information End >>>
75
76      1 pp 8v      do i=1,n
77      1 p 8v        a(i) = a(i) + s * b(c(i))
78      1 p 8v      enddo
79
80      !OCL END_CACHE_SUBSECTOR
81      !OCL END_CACHE_SECTOR_SIZE
    
```

Increases reusability of array b

The L2 miss decreased significantly.



Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.01%	2.78%	9.35E+09	18.29%	81.71%	0.00%	2.35%	7.87E+09	125.54	140.97
After improvement	0.02%	2.71%	9.10E+09	16.76%	83.24%	0.00%	1.56%	5.25E+09	182.57	157.96

Sector Cache: Case Example 2 (Before Improvement)

Data in array u cannot be reused because it has been forced out of the cache.
Consequently, the following is a frequent event: No instruction commit due to memory and cache busy.

Source code before improvement

```

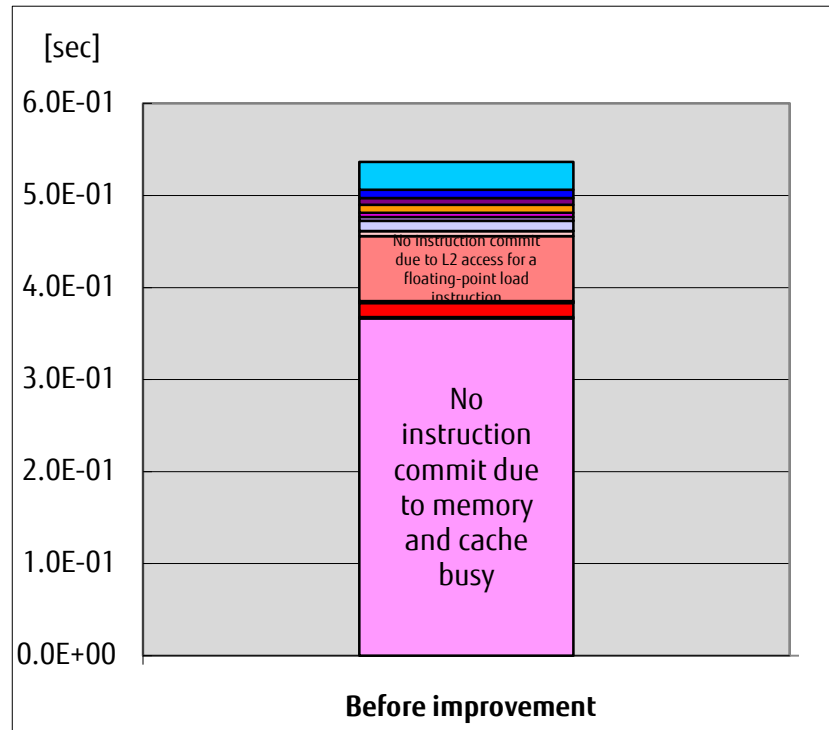
128      subroutine JACOBI(u, rhs, niter, unew)
129      REAL(double), INTENT(IN), dimension(0:n1-1, 0:n2-1, 0:n3-1) :: u
130      REAL(double), INTENT(IN), dimension(0:n1-1, 0:n2-1, 0:n3-1) :: rhs
131      REAL(double), INTENT(OUT), dimension(0:n1-1, 0:n2-1, 0:n3-1) :: unew
132      INTEGER, INTENT(IN) :: niter
133      INTEGER :: iter, i, j, k
      <<< Loop-information Start >>>
      <<< [PARALLELIZATION]
      <<< Standard iteration count: 2
      <<< Loop-information End >>>
161  2 pp      do k=1, n3-2
162  3 p        do j=1, n2-2
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< SIMD (VL: 4)
      <<< SOFTWARE PIPELINING
      <<< Loop-information End >>>
163  4 p 6v      do i=1, n1-2
164  4 p 6v        unew(i,j,k) = &
165  4              ((u(i+1,j,k) + u(i-1,j,k)) * h1sqinv &
166  4                + (u(i,j+1,k) + u(i,j-1,k)) * h2sqinv &
167  4                + (u(i,j,k+1) + u(i,j,k-1)) * h3sqinv &
168  4                - rhs(i,j,k)) * hhhinv
169  4 p 6v      end do
170  3 p          end do
171  2 p          end do
  
```

n1 = 452
n2 = 52
n3 = 322

Array u should be in the cache because of the reusability in the i,j dimensions of array u.

The percentage of L2 misses is high.

Size of each array: unew, u, rhs: 60.5 MB



A memory throughput bottleneck has occurred.

Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load- store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.02%	2.01%	2.32E+08	3.91%	96.09%	0.00%	1.92%	2.22E+08	110.62	127.79

Sector Cache: Case Example 2 (After Improvement)

Placing part of the k dimension of array u in sector 1 increases cache efficiency, which improves the following event: No instruction commit due to memory and cache busy.

Source code after improvement (optimization control line tuning)

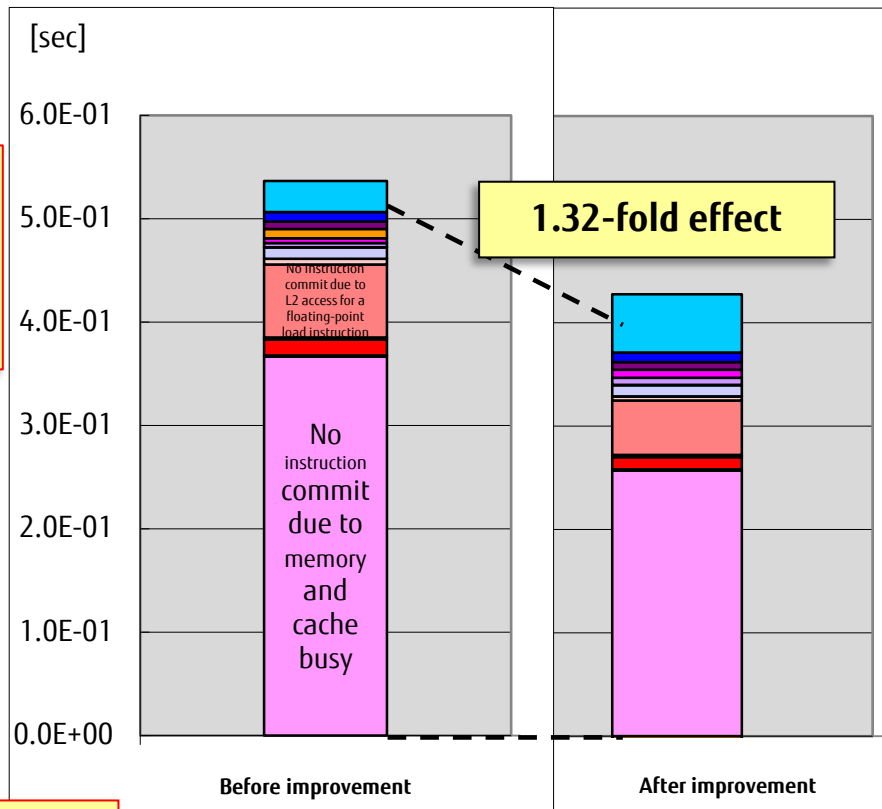
```

159      !ocl CACHE_SECTOR_SIZE(6,18)
160      !ocl CACHE_SUBSECTOR_ASSIGN(u)
      <<< Loop-information Start >>>
      <<< [PARALLELIZATION]
      <<< Standard iteration count: 2
      <<< Loop-information End >>>
165  2 pp      do k=1,n3-2
166  3 p        do j=1,n2-2
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< SOFTWARE PIPELINING
      <<< Loop-information End >>>
167  4 p 6v      do i=1,n1-2
168  4 p 6v      unew(i,j,k) = &
169  4            ((u(i+1,j,k) + u(i-1,j,k)) * h1sqinv &
170  4            +(u(i,j+1,k) + u(i,j-1,k)) * h2sqinv &
171  4            +(u(i,j,k+1) + u(i,j,k-1)) * h3sqinv &
172  4            -rhs(i,j,k)) * hhhinv
173  4 p 6v      end do
174  3 p        end do
175  2 p        end do
177  1          end do
178      !ocl END_CACHE_SUBSECTOR
179      !ocl END_CACHE_SECTOR_SIZE
    
```

To place part of array u in the cache for each thread, a cache size of 9 MB is required.

Increases reusability of array u

The percentage of L2 misses decreased from 1.92% to 1.41%.



Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.02%	2.01%	2.32E+08	3.91%	96.09%	0.00%	1.92%	2.22E+08	110.62	127.79
After improvement	0.03%	2.01%	2.32E+08	3.94%	96.06%	0.00%	1.41%	1.63E+08	146.05	131.47

Sector Cache: Case Example 2 (Cyclic Distribution)

In this case example, the `schedule(static,1)` specification divides the array into smaller parts to which cache memory is cyclically allocated. Then, parallel execution is performed. The effect of this technique can be equivalent to a sector cache.

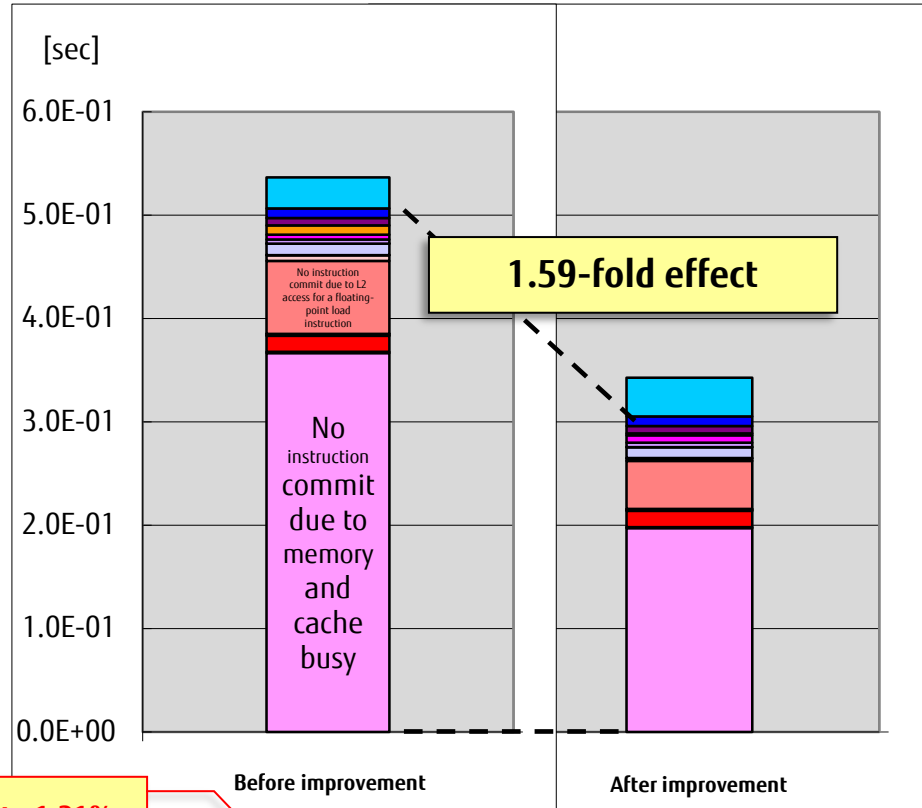
Source code after improvement

```

157      !$omp parallel shared(n1,n2,n3,u,rhs, niter,
      h1sqinv,h2sqinv,h3sqinv,
      hhhinv)
161  1      !$omp do schedule(static,1)
162  2 p      do k=1,n3-2
163  3 p      do j=1,n2-2
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< SOFTWARE PIPELINING
      <<< Loop-information End >>>
164  4 p 6v      do i=1,n1-2
165  4 p 6v          unew(i,j,k) = &
166  4              ((u(i+1,j,k) + u(i-1,j,k)) * h1sqinv &
167  4              +(u(i,j+1,k) + u(i,j-1,k)) * h2sqinv &
168  4              +(u(i,j,k+1) + u(i,j,k-1)) * h3sqinv &
169  4              -rhs(i,j,k)) * hhhinv
170  4 p 6v      end do
171  3 p      end do
172  2 p      end do
173  1      !$omp end do
176      !$omp end parallel
    
```

Increases
reusability of
array u

The percentage of L2 misses decreased from 1.92% to 1.21%.



Cache

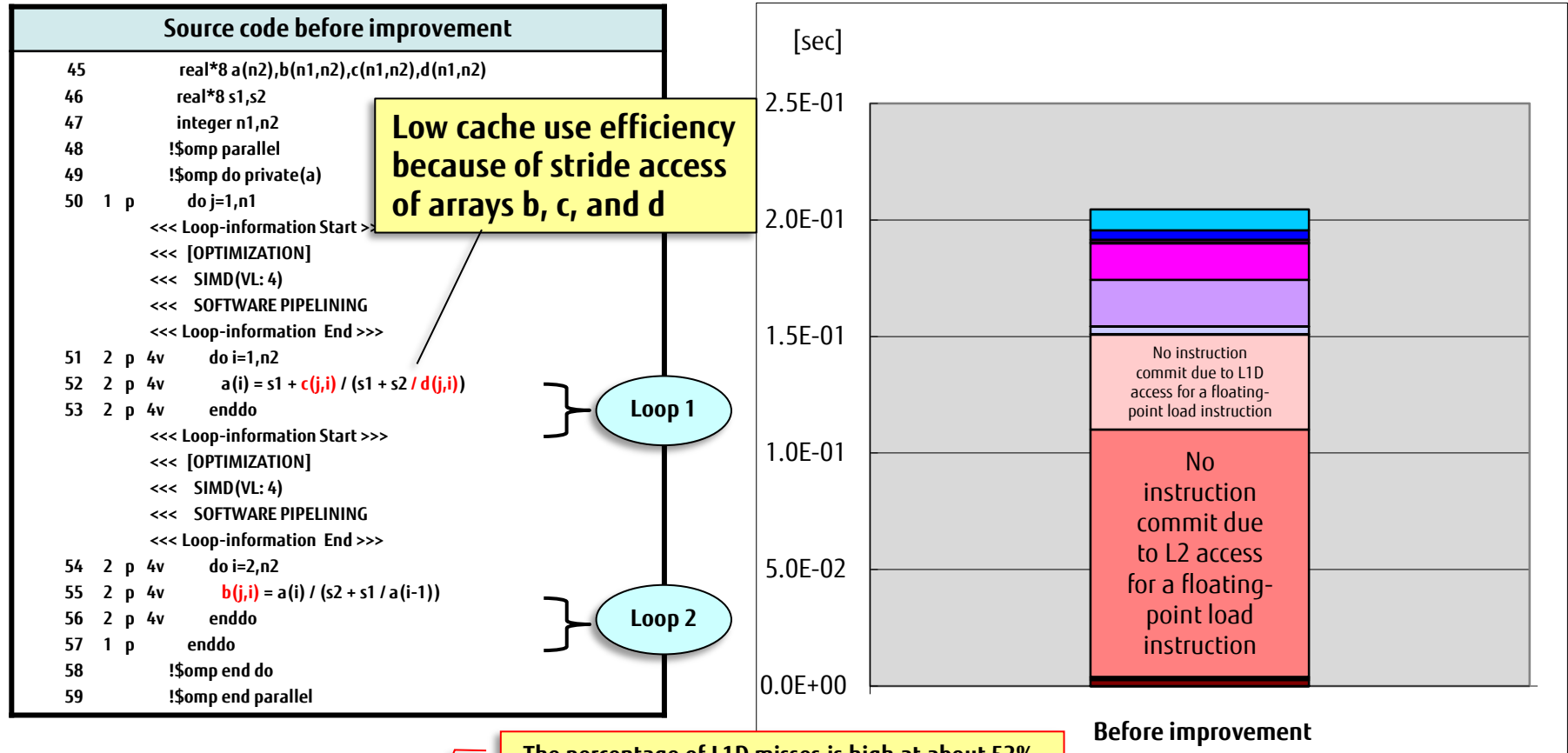
	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.02%	2.01%	2.32E+08	3.91%	96.09%	0.00%	1.92%	2.22E+08	110.62	127.79
After improvement	0.01%	2.02%	2.33E+08	4.66%	95.34%	0.00%	1.21%	1.40E+08	174.05	138.63

Loop Interchange

- Loop Interchange (Before Improvement)
- Contents of Loop Interchange Tuning
- Effects of Loop Interchange (Source Tuning)

Loop Interchange (Before Improvement)

Cache use efficiency decreases because of stride access of arrays b, c, and d. Consequently, the following is a frequent event: No instruction commit due to L2 access for a floating-point load instruction.



Cache

The percentage of L1D misses is high at about 52%.

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	51.15%	3.86E+08	98.73%	1.27%	0.00%	0.00%	9.03E+03	483.01	0.02

Contents of Loop Interchange Tuning

Source code 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
```

Low cache use efficiency because of stride access of arrays b, c, and d.

(1) Converting array a into a two-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
```

There is no longer an array a dependency, which is a factor hindering loop fission.

(3) Interchanging loops

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

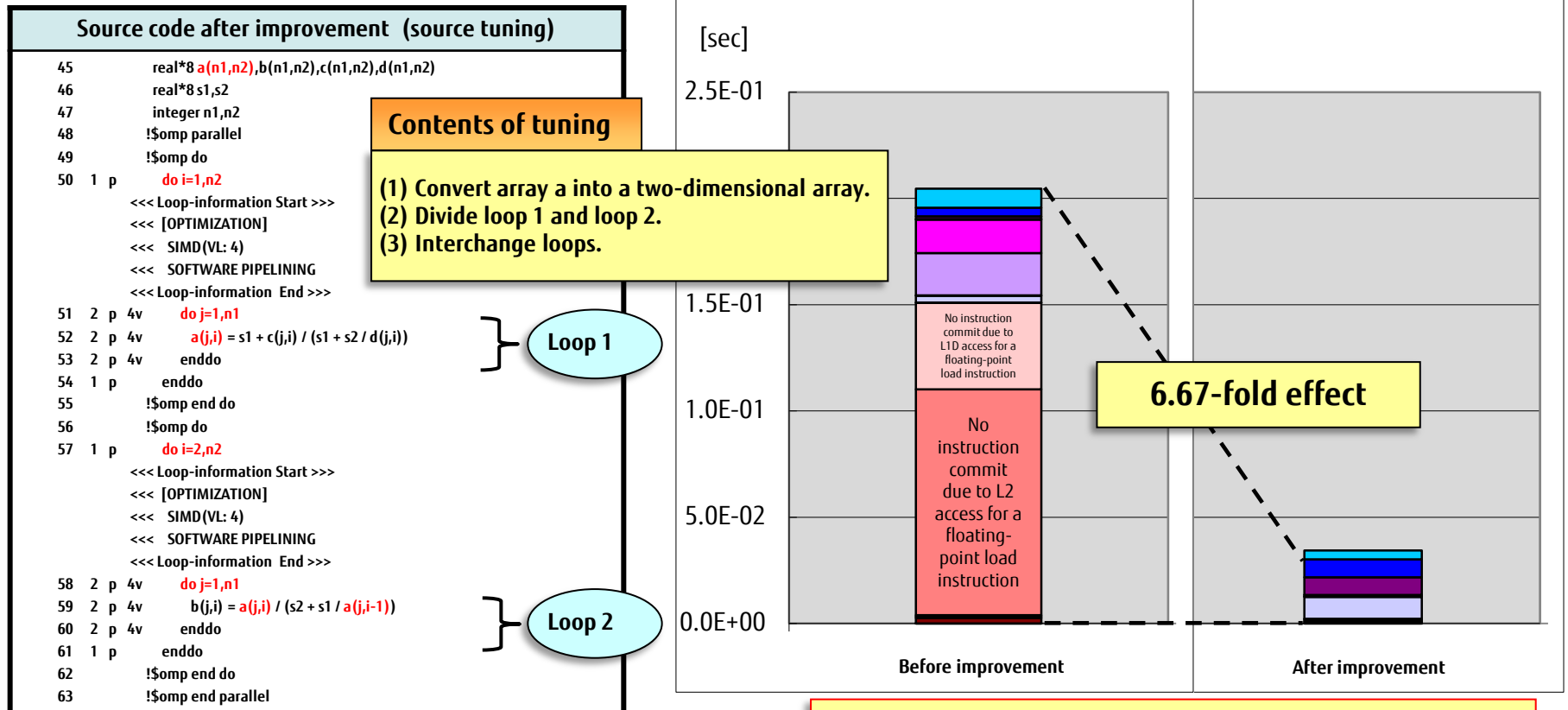
Access to arrays b, c, and d becomes sequential access, which improves cache use efficiency.

(2) Dividing loop 1 and loop 2

```
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
```

Effects of Loop Interchange (Source Tuning)

Cache efficiency increases because of sequential access of an array through loop interchange. This results in improvement of the following event: No instruction commit due to L2 access for a floating-point load instruction.



Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	51.15%	3.86E+08	98.73%	1.27%	0.00%	0.00%	9.03E+03	483.01	0.02
After improvement	0.00%	2.64%	2.04E+07	6.97%	93.03%	0.00%	0.00%	5.00E+03	152.22	0.07

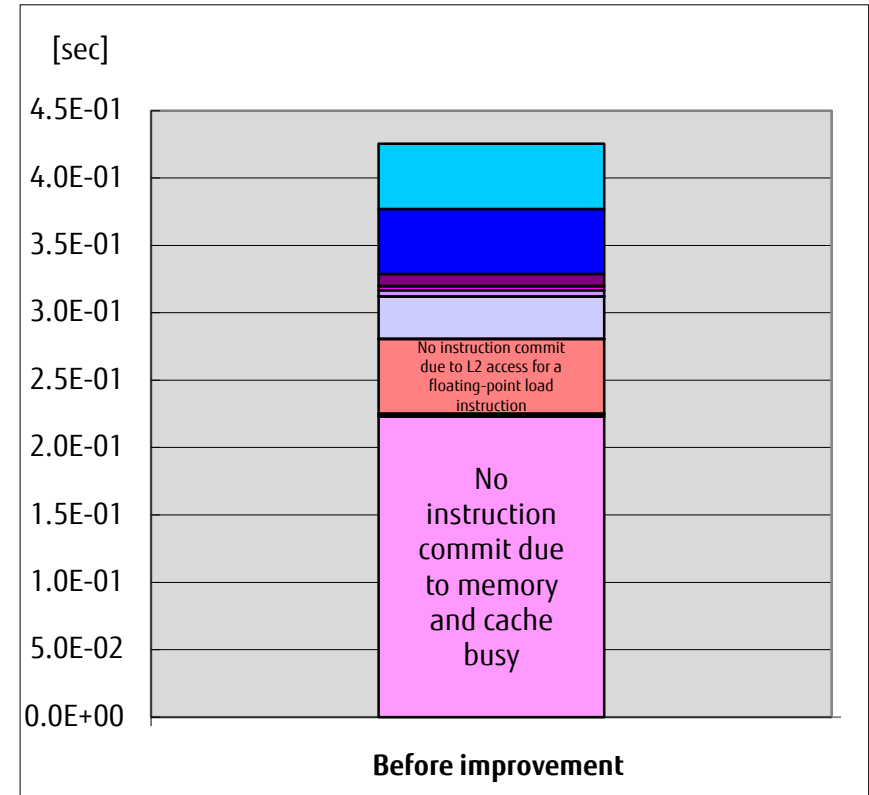
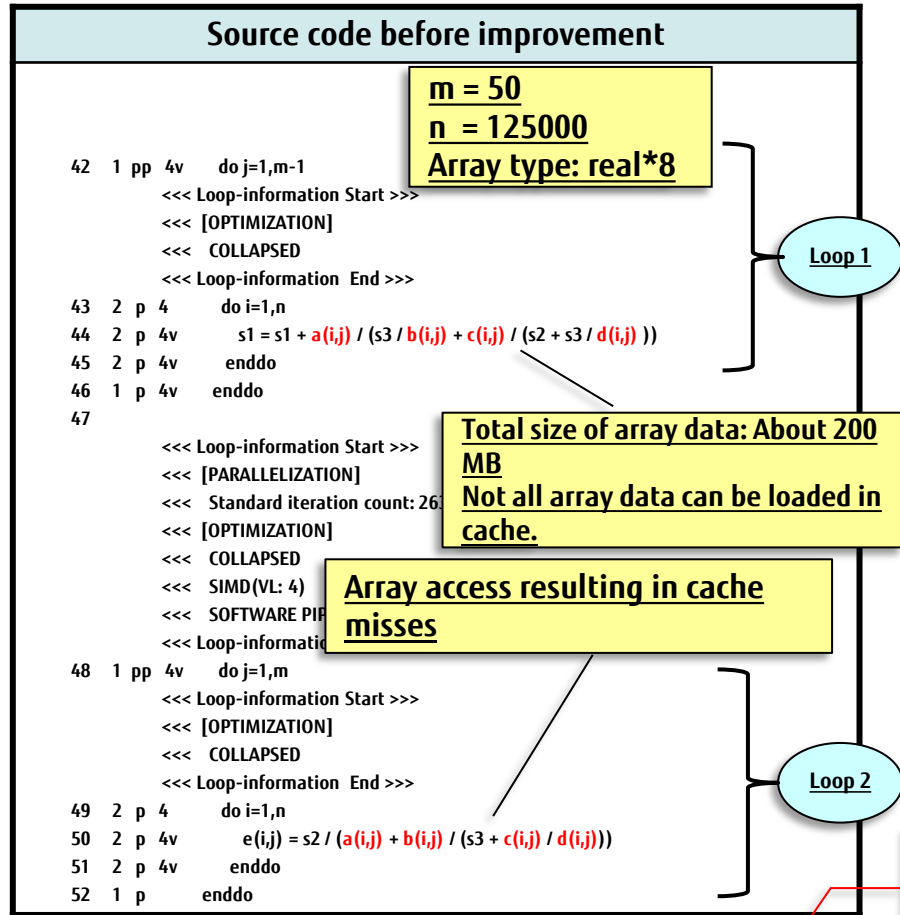
The percentage of L1D misses decreased significantly.

Loop Fusion

- Loop Fusion (Before Improvement)
- Effects of Loop Fusion (Source Tuning)
- Loop Fusion (in C Language) (Before Improvement)
- Effects of Loop Fusion (in C Language) (Source Tuning)

Loop Fusion (Before Improvement)

Not all array data can be loaded in cache because loop 1 has many iterations, so loop 2 cannot reuse the data. Consequently, the following is a frequent event: No instruction commit due to memory and cache busy.



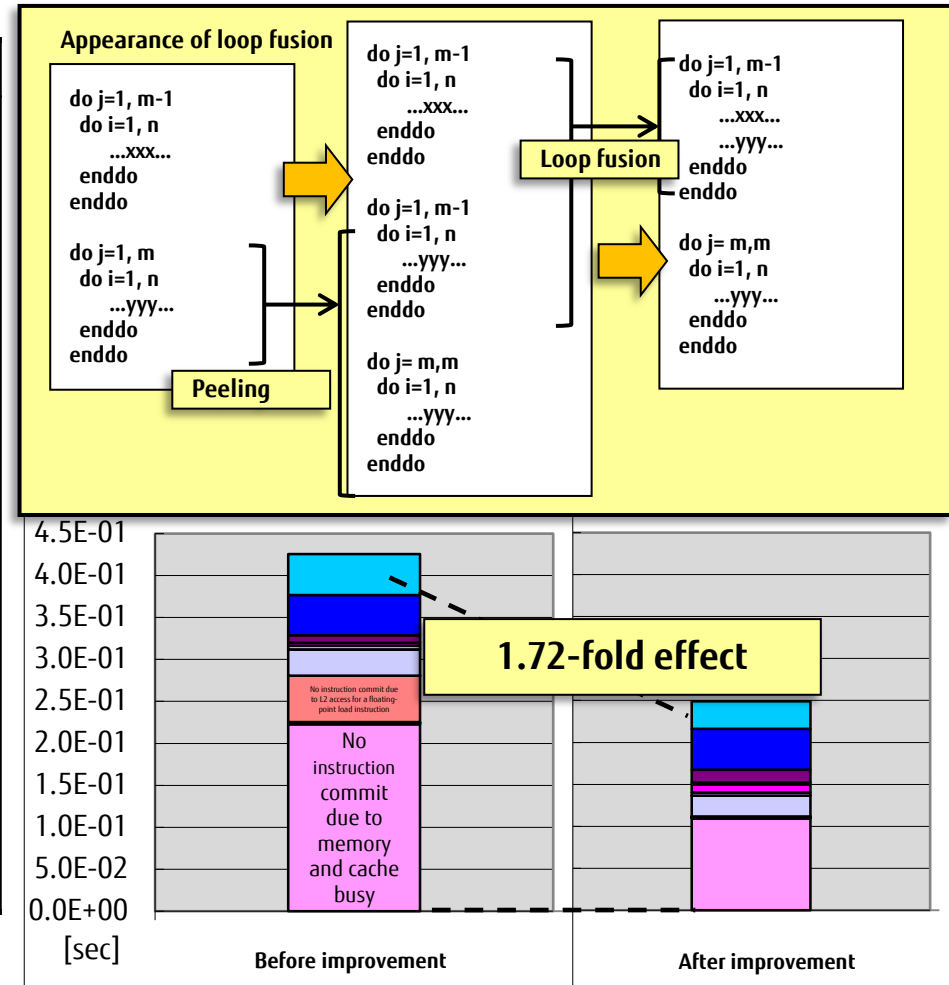
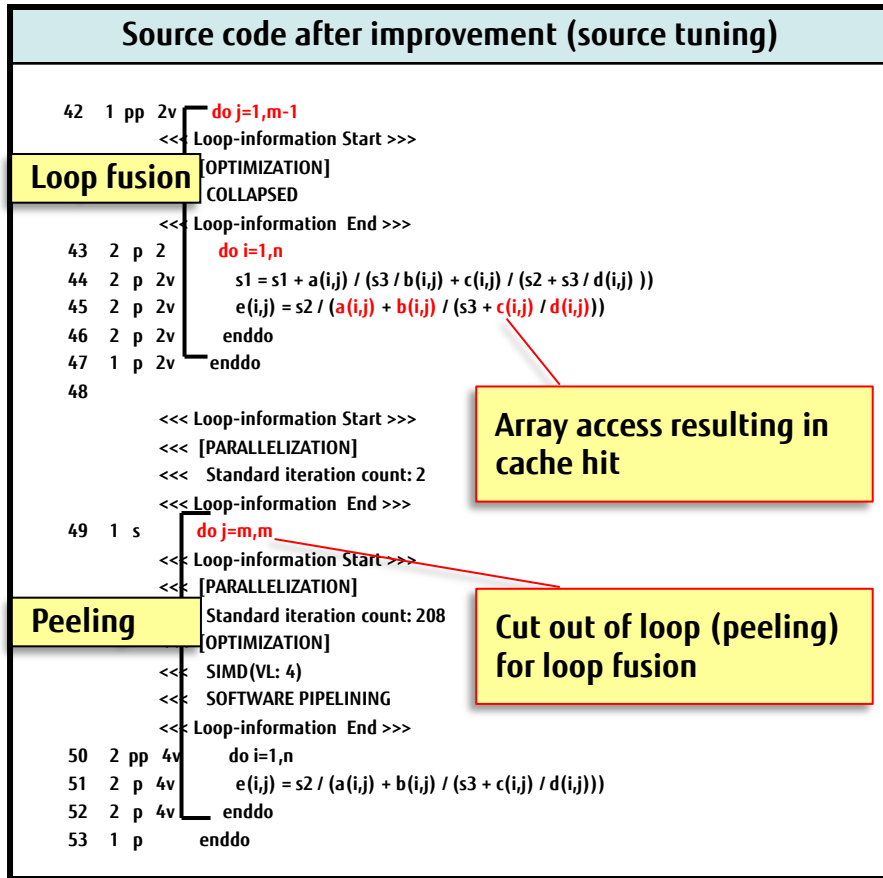
The percentages of L1 cache misses and L2 cache misses are 3.125%, which is the theoretical value for stream access. -> Misses have occurred in both loops 1 and 2. This means that loop 2 could not use data placed in the cache in loop 1.

Cache

	L1I miss rate (effective instruction)	Number of loads and stores	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate (/L1D miss)	L1D miss hwpf rate (/L1D miss)	L1D miss swpf rate (/L1D miss)	L2 miss rate (/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.01%	5.58E+09	3.13%	1.74E+08	0.23%	99.77%	0.00%	3.13%	1.74E+08	104.92	116.69

Effects of Loop Fusion (Source Tuning)

Loop fusion increases cache efficiency, which improves the following event: No instruction commit due to memory and cache busy.



Cache

The numbers of L1D misses and L2 misses decreased significantly.

	L1I miss rate (effective instruction)	Number of loads and stores	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.01%	5.58E+09	3.13%	1.74E+08	0.23%	99.77%	0.00%	3.13%	1.74E+08	104.92	116.69
After improvement	0.00%	3.13E+09	3.13%	9.78E+07	0.68%	99.32%	0.00%	3.13%	9.78E+07	100.58	120.70

Loop Fusion (in C Language) (Before Improvement)

Not all array data can be loaded in cache because loop 1 has many iterations, so loop 2 cannot reuse the data.
 Consequently, the following is a frequent event: No instruction commit due to memory and cache busy.

Source code before improvement

```

<<< Loop-information Start >>>
<<< [PARALLELIZATION]
<<< Standard iteration count: 2
<<< Loop-information End >>>
45 pp   for(j=0;j<M-1;j++){
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< SIMD(VL: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End >>>
46 p 2v   for(i=0;i<N;i++){
47 p 2v     *s1 = *s1 + a[j][i] / (s3 / b[j][i] + c[j][i] / (s2 + s3 / d[j][i]));
48 p 2v   }
49   }
50
<<< Loop-information Start >>>
<<< [PARALLELIZATION]
<<< Standard iteration count: 2
<<< Loop-information End >>>
51 pp   for(j=0;j<M;j++){
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< SIMD(VL: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End >>>
52 p 2v   for(i=0;i<N;i++){
53 p 2v     e[j][i] = s2 / (a[j][i] + b[j][i] / (s3 + c[j][i] / d[j][i]));
54 p 2v   }
55   }
                
```

M = 50

N = 125000

Array type: double

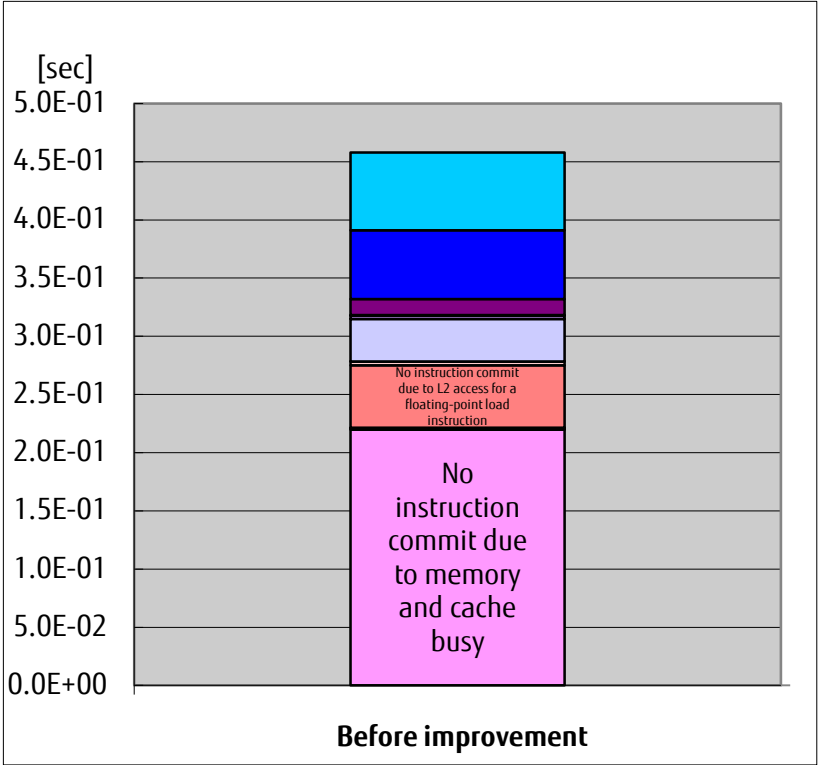
Loop 1

Total size of array data: About 200 MB

Not all array data can be loaded in cache.

Array access resulting in cache misses

Loop 2



The percentages of L1 cache misses and L2 cache misses are 3.125%, which is the theoretical value for stream access. -> Misses have occurred in both loops 1 and 2. This means that loop 2 could not use data placed in the cache in loop 1.

Cache

	L1I miss rate (effective instruction)	Number of loads and stores	L1D miss rate(/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	5.58E+09	3.13%	1.74E+08	0.22%	99.78%	0.00%	3.13%	1.75E+08	97.51	108.55

Effects of Loop Fusion (in C Language) (Source Tuning)

Loop fusion increases cache efficiency, which improves the following event: No instruction commit due to memory and cache busy.

Source code after improvement (source tuning)

```

<<< Loop-information Start >>>
<<< [PARALLELIZATION]
<<< Standard iteration count: 2
<<< Loop-information End >>>
44 pp  for(j=0;j<M-1;j++){
Loop fusion  Loop-information Start >>>
[OPTIMIZATION]
<<< SIMD(VL: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End >>>
45 p  2v  for(i=0;i<N;i++){
46 p  2v      *s1 = *s1 + a[j][i] / (s3 / b[j][i] + c[j][i] / (s2 + s3 / d[j][i] ));
47 p  2v      e[j][i] = s2 / (a[j][i] + b[j][i] / (s3 + c[j][i] / d[j][i]));
48 p  2v  }
49  }
50
51  for(j=M;j<M;j++){
52  for(i=0;i<N;i++){
Peeling      e[j][i] = s2 / (a[j][i] + b[j][i] / (s3 + c[j][i] / d[j][i]));
54  }
55  }
56
57  }
    
```

Array access resulting in cache hit

Appearance of loop fusion

```

for(j=0;j<M-1;j++){
  for(i=0;i<N;i++){
    . . . xxx . . .
  }
}

for(j=0;j<M-1;j++){
  for(i=0;i<N;i++){
    . . . yyy . . .
  }
}
    
```

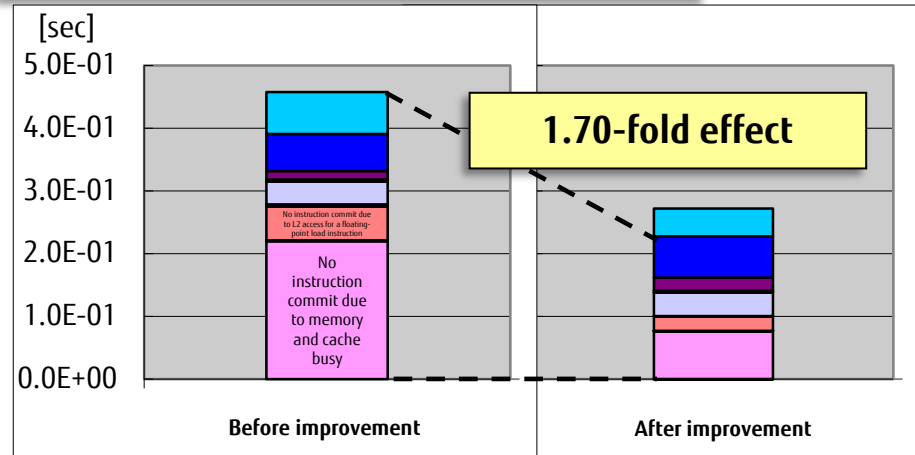
```

for(j=0;j<M-1;j++){
  for(i=0;i<N;i++){
    . . . xxx . . .
    . . . yyy . . .
  }
}

for(j=0;j<M-1;j++){
  for(i=0;i<N;i++){
    . . . yyy . . .
  }
}
    
```

Loop fusion

Peeling



The numbers of L1D misses and L2 misses decreased significantly.

Cache

	L1I miss rate (effective instruction)	Number of loads and stores	L1D miss rate (Load-store instruction)	L1D miss	L1D miss dm rate (L1D miss)	L1D miss hwpf rate (L1D miss)	L1D miss swpf rate (L1D miss)	L2 miss rate (Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	5.58E+09	3.13%	1.74E+08	0.22%	99.78%	0.00%	3.13%	1.75E+08	97.51	108.55
After improvement	0.00%	3.06E+09	3.13%	9.58E+07	0.20%	99.80%	0.00%	3.13%	9.58E+07	90.26	108.38

Array Merging (Indirect Access)

- What Is Array Merging?
- Array Merging (Before Improvement)
- Effects of Array Merging (Source Tuning)
- Array Merging (in C Language) (Before Improvement)
- Effects of Array Merging (in C Language) (Source Tuning)

What Is Array Merging?

Array merging is the merging of multiple arrays into one array. These multiple arrays are processed in the same loop and have a common access pattern. This technique realizes sequential data access and increases cache efficiency.

Source code before improvement	After improvement (appearance after compiler optimization)
<pre>parameter(n=1000000) real*8 a(n), b(n), c(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 :</pre> <div>Access of different cache lines</div>	<pre>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 :</pre> <div>Access of same cache line</div>

Array merging

(L1D cache)

a(d(i))
. . .
a(d(i+1))
. . .
b(d(i))
. . .
b(d(i+1))
. . .
c(d(i))
. . .
c(d(i+1))
. . .

(L1D cache)

abc(1, d(i))
abc(2, d(i))
abc(3, d(i))
. . .
abc(1, d(i+1))
abc(2, d(i+1))
abc(3, d(i+1))
. . .

Array Merging (Before Improvement)

Cache use efficiency decreases because of a high percentage of L1D misses (list access). Consequently, the following is a frequent event: No instruction commit due to L2 access for a floating-point load instruction.

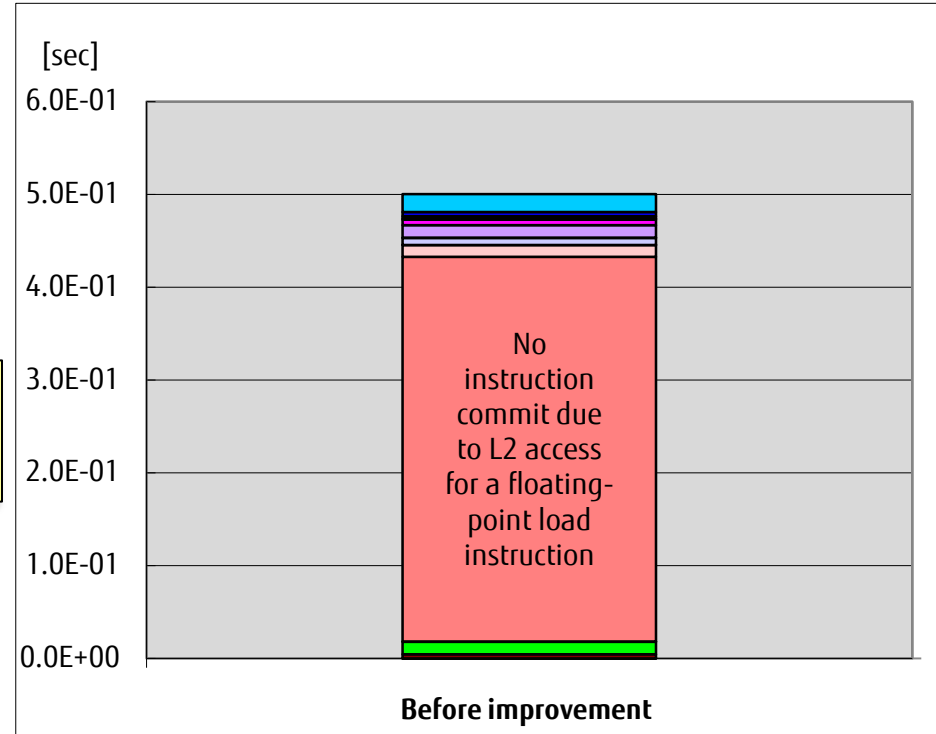
Source code before improvement

```

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 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      !$omp parallel do schedule (static,96)
    <<< Loop-information Start >>>
    <<< [OPTIMIZATION]
    <<< SIMD(VL: 4)
    <<< SOFTWARE PIPELINING
    <<< Loop-information End >>>
30 1 p 2v      do i = 1, n
31 1 p 2v          ii = d(i)
32 1 p 2v          a(ii) = s / (s + f(ii)) / (s + e(ii) / (b(ii) + s / c(ii)))
33 1 p 2v      enddo
34      !$omp end parallel do
:

```

Arrays a, f, e, b, and c are list access.



The percentage of L1D misses is high at 78.45%.

Cache

	L1I miss rate (effective instruction)	L1D miss rate (/Load-store instruction)	L1D miss	L1D miss dm rate (/L1D miss)	L1D miss hwpf rate (/L1D miss)	L1D miss swpf rate (/L1D miss)	L2 miss rate (/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	78.45%	1.27E+09	100.00%	0.00%	0.00%	0.00%	6.70E+04	649.65	0.06

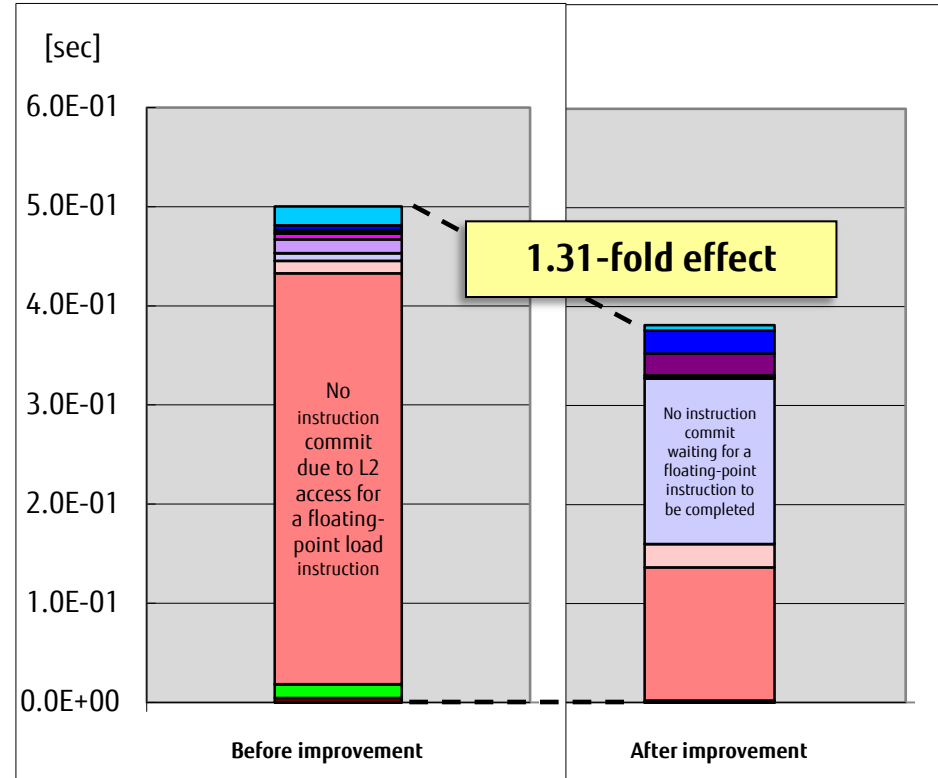
Effects of Array Merging (Source Tuning)

Array merging for list access increases cache efficiency, which improves the following event: No instruction commit due to L2 access for a floating-point load instruction.

Source code 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 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      !$omp parallel do schedule (static,96)
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< SIMD(VL: 4)
<<< Loop-information End >>>
29 1 p 2v      do i = 1, n
30 1 p 2v      ii = d(i)
31 1 p 2v      abcef(1,ii) = s / (s + abcef(5,ii)) / (s + abcef(4,ii)
32 1          * / (abcef(2,ii) + s / abcef(3,ii)))
33 1 p 2v      enddo
34      !$omp end parallel do
:
    
```



Cache

The percentage of L1D misses decreased significantly.

	L1I miss rate (effective instruction)	L1D miss rate(/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load- store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	78.45%	1.27E+09	100.00%	0.00%	0.00%	0.00%	6.70E+04	649.65	0.06
After improvement	0.00%	18.01%	2.90E+08	99.99%	0.01%	0.00%	0.00%	1.52E+04	194.49	0.03

Array Merging (in C Language) (Before Improvement)

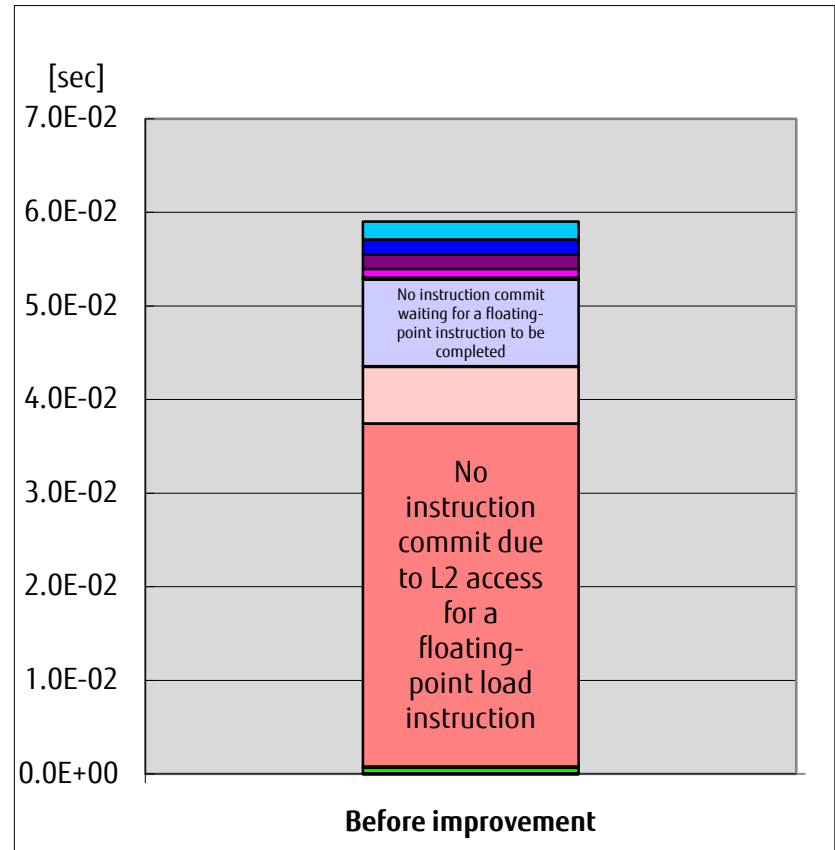
Cache use efficiency decreases because of a high percentage of L1D misses (list access). Consequently, the following is a frequent event: No instruction commit due to L2 access for a floating-point load instruction.

Source code before improvement

```

4  #define N 2*1000*1000/8
5
6  void sub(double a[N],const double b[N],const double c[N],
           const int d[N],const double e[N],const double f[N],double);
:
10  int i;
11  double a[N],b[N],c[N],e[N],f[N],s;
12  int d[N];
:
40  5    sub(a,(const double (*)b,(const double (*)c,(const int (*)d,
           (const double (*)e,(const double (*)f,s);
:
48  void sub(double a[N],const double b[N],const double c[N],
           const int d[N],const double e[N],const double f[N],double s)
49  {
50  int i,ii;
51
52  #pragma omp parallel for schedule (static,96)
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< SIMD(VL: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End >>>
53  p 2v for(i=0;i<N;i++){
54  p 2v ii=d[i];
55  p 2v a[ii]=s / (s + f[ii] / (s + e[ii] / (b[ii] + s / c[ii]))));
56  p 2v }
57  }
```

Arrays a, f, e, b, and c are list access.



The percentage of L1D misses is high at 70.79%.

Cache

	L1I miss rate (effective instruction)	L1D miss rate/(Load-store instruction)	L1D miss	L1D miss dm rate/(L1D miss)	L1D miss hwpf rate/(L1D miss)	Percentage of L1D misses due to swpf (relative to number of L1D misses)	L2 miss rate/(Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	70.79%	1.28E+08	100.00%	0.00%	0.00%	0.02%	4.12E+04	552.20	0.34

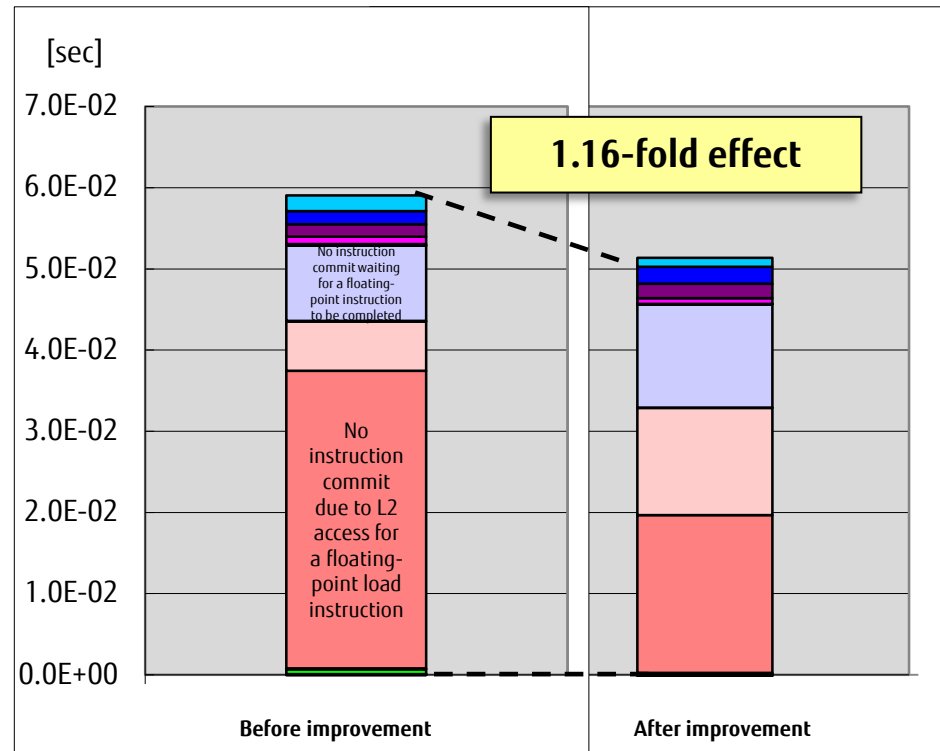
Effects of Array Merging (in C Language) (Source Tuning) FUJITSU

Array merging for list access increases cache efficiency, which improves the following event: No instruction commit due to L2 access for a floating-point load instruction.

Source code after improvement (source tuning)

```

4  #define N 2*1000*1000/8
5  #define M 5
6
7  double abcef[N][5];
8  void sub(double abcef[N][5], const int d[N], double s);
9  :
14  int d[N];
15  :
42  5  sub(abcef, (const int (*)d), s);
43  :
50  void sub(double abcef[N][5], const int d[N], double s)
51  {
52  int i, ii;
53
54  #pragma omp parallel for schedule (static, 96)
55  <<< Loop-information Start >>>
56  <<< [OPTIMIZATION]
57  <<< SIMD (VL: 4)
58  <<< Loop-information End >>>
59  p 2v for(i=0; i<N; i++){
60  p 2v ii=d[i];
61  p 2v abcef[i][0]=s / (s + abcef[i][4]) / (s + abcef[i][3]) / (abcef[i][1] +
62  s / abcef[i][2]));
63
64  p 2v }
65  }
```



Cache

The percentage of L1D misses decreased significantly.

	L1I miss rate (effective instruction)	L1D miss rate(/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	70.79%	1.28E+08	100.00%	0.00%	0.00%	0.02%	4.12E+04	552.20	0.34
After improvement	0.00%	12.41%	2.90E+07	99.99%	0.01%	0.00%	0.01%	1.36E+04	144.88	0.13

Improvement in Data Access Wait (Latency Concealment)

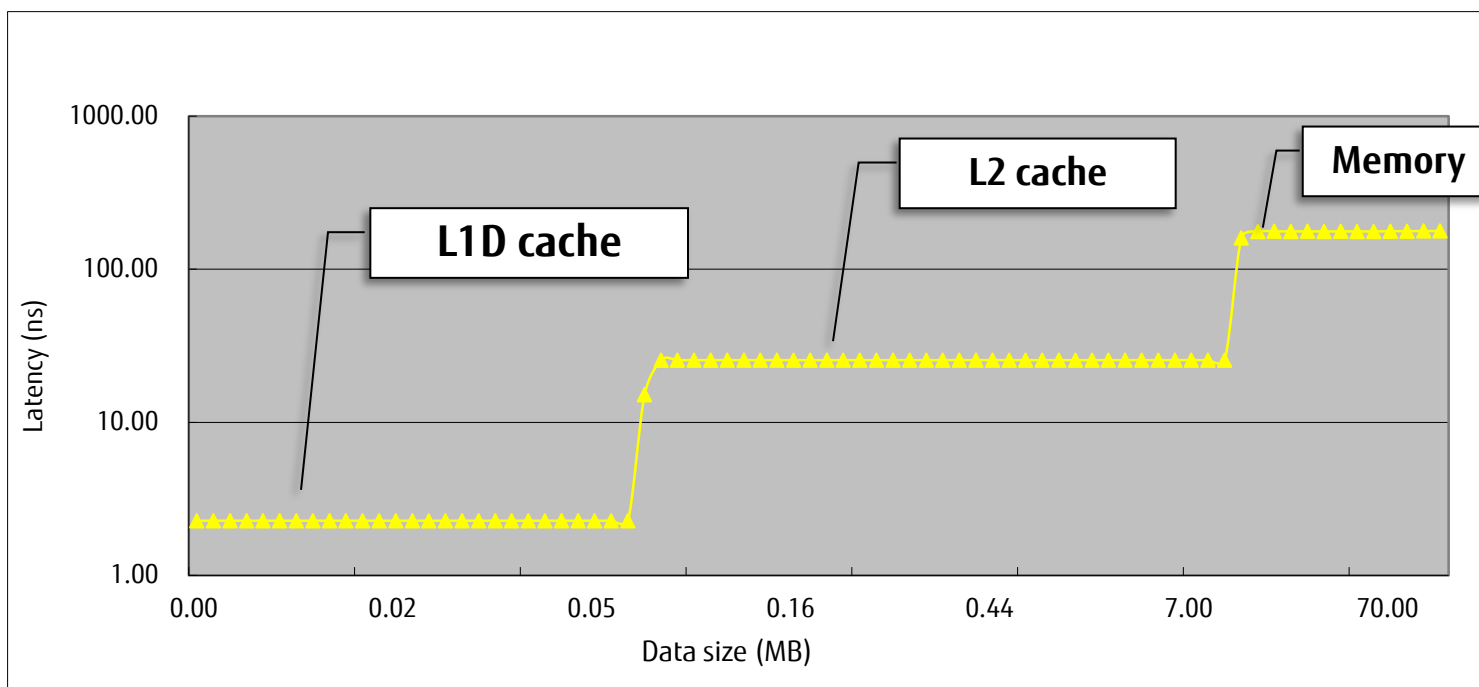
- What Is Latency Concealment?
- Indirect Access Prefetching
- Prefetching for an Outer Loop

What Is Latency Concealment?

Latency concealment means **concealing the latency of data access (the period of time from a data transfer request to its acknowledgement) by prefetching data**. There are three types of data access: L1D cache access, L2 cache access, and memory access. For L2 cache access and memory access among these types, this section discusses only the latency visible as execution time.

For the latency time of each data access type, see the LMBench results below.

■ Results of data access latency measurement with LMBench



Indirect Access Prefetching

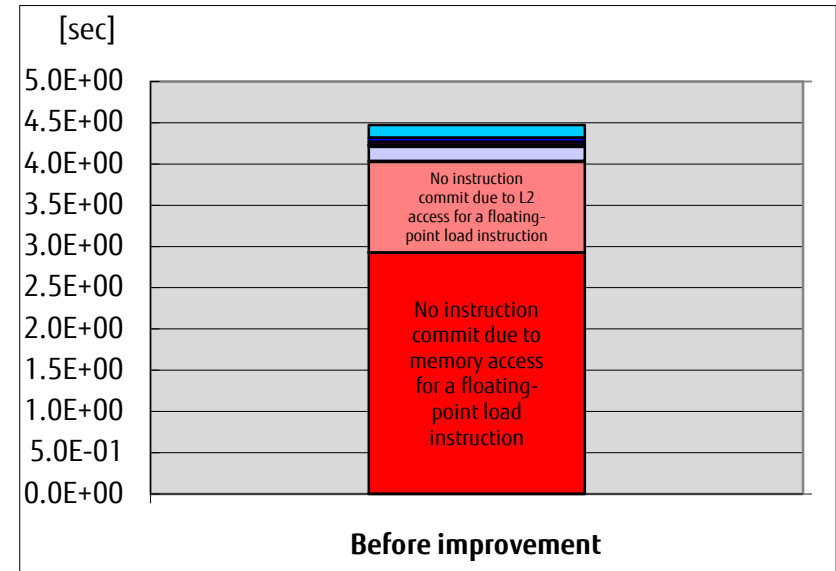
- Indirect Access Prefetching (Before Improvement)
- Effects of Indirect Access Prefetching (Optimization Control Line Tuning)
- Effects of Indirect Access Prefetching (Optimization Control Line)
- Effects of Indirect Access Prefetching (Compiler Options Tuning)

Indirect Access Prefetching (Before Improvement)

Indirect access (list access) with the recommended options does not generate a prefetch. Also, memory access latency is visible. Consequently, the following is a frequent event: No instruction commit due to memory access for a floating-point load instruction.

Source code before improvement	
51	<pre> <<< [PARALLELIZATION] <<< Standard iteration count: 485 <<< [OPTIMIZATION] <<< PREFETCH : 36 <<< e: 12, d: 12, a: 12 <<< Loop-information End >>> 52 1 pp 6 do i = 1, n 53 2 p 6 if (mod(i,2) .eq. 0) then 54 2 p 6 a(i) = b(d(i)) + scalar * c(e(i)) 55 2 p 6 endif 56 1 p 6 enddo </pre>

Indirect access for arrays b and c



Cache

	L1 miss rate (effective instruction)	L1D miss rate/(Load-store instruction)	L1D miss	L1D miss dm rate/(L1D miss)	L1D miss hwpf rate/(L1D miss)	L1D miss swpf rate/(L1D miss)	L2 miss rate/(Load-store instruction)	L2 miss	L2 miss dm rate/(L2 miss)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.01%	42.97%	1.94E+09	94.20%	0.00%	5.80%	12.48%	5.63E+08	48.76%	111.02	35.47

The L1D miss dm percentage and L2 miss dm percentage are high, and prefetching is not effective. Performance may increase because there are margins in memory throughput and L2 throughput.

Specification of the **prefetch specifier** generates a prefetch for indirect access (list access). This results in improvement of the following event: No instruction commit due to memory access for a floating-point load instruction.

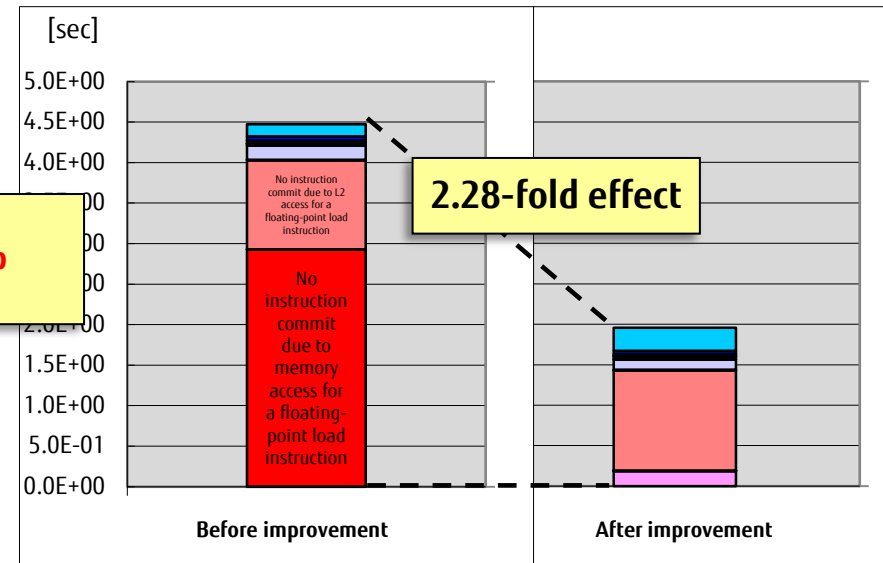
Source code after improvement (optimization control line tuning)

```

51      !ocl prefetch
      <<< Loop-information Start >>>
      <<< [PARALLELIZATION]
      <<< Standard iteration count: 485
      <<< [OPTIMIZATION]
      <<< PREFETCH      :60
      <<< e: 12, c: 12, d: 12, b: 12, a: 12
      <<< Loop-information End >>>

52  1 pp 6      do i = 1, n
53  2 p 6      if ( mod(i,2) .eq. 0 ) then
54  2 p 6          a(i) = b(d(i)) + scalar * c(e(i))
55  2 p 6      endif
56  1 p 6      enddo
    
```

Generated prefetch for indirect access (arrays b and c)



Cache

	L1I miss rate (effective instruction)	L1D miss rate(/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 miss	L2 miss dm rate(/L2 miss)	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.01%	42.97%	1.94E+09	94.20%	0.00%	5.80%	12.48%	5.63E+08	48.76%	111.02	35.47
After improvement	0.00%	38.12%	3.09E+09	52.34%	0.00%	47.66%	8.84%	7.16E+08	5.34%	404.14	101.08

The generation of prefetch instructions for indirect access (arrays b and c) reduced the L1D miss dm percentage and L2 miss dm percentage.

Here, specify the following optimization control line.

Optimization control specifiers	Meaning	Optimization control line that can be specified			
		Program unit	DO loop unit	Statement unit	Array assignment statement unit
prefetch	Enables the automatic prefetch function of the compiler.	Yes	Yes	No	No

◆ Remarks

- The prefetch optimization control specifiers is equivalent to specifying the following compiler options:
-Kprefetch_sequential,prefetch_stride,prefetch_indirect,
prefetch_conditional,prefetch_cache_level=all

◆ Notes

- Depending on the cache efficiency of loops, whether branching exists, and the complexity of subscripts, prefetching with the compiler options -Kprefetch_sequential, -Kprefetch_stride, -Kprefetch_indirect, or -Kprefetch_conditional enabled may degrade execution performance.

You can achieve effects similar to optimization control line tuning by specifying the following compiler options.

Compiler options	Description of function
-Kprefetch_indirect	Gives an instruction on whether to generate an object that uses a prefetch instruction for indirectly accessed (list access) array data used inside a loop. This option has meaning in cases where -O1 or a higher option is valid. The default is -Kprefetch_noindirect.

■ Use example (source code before improvement)

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

◆ Notes

- Depending on the cache efficiency of loops, whether IF construct are used, and the complexity of subscripts, prefetching may not have the intended effect.

Prefetching for an Outer Loop

- Prefetching for an Outer Loop (Before Improvement)
- Effects of Prefetching for an Outer Loop
(Optimization Control Line Tuning)
- Use of software prefetch

Prefetching for an Outer Loop (Before Improvement)

The innermost loop has a few iterations, and its array size is greater than its number of iterations. For this reason, the cost at the prefetching rise time is visible in normal prefetching. Consequently, the following is a frequent event: No instruction commit due to L2 access for a floating-point load instruction.

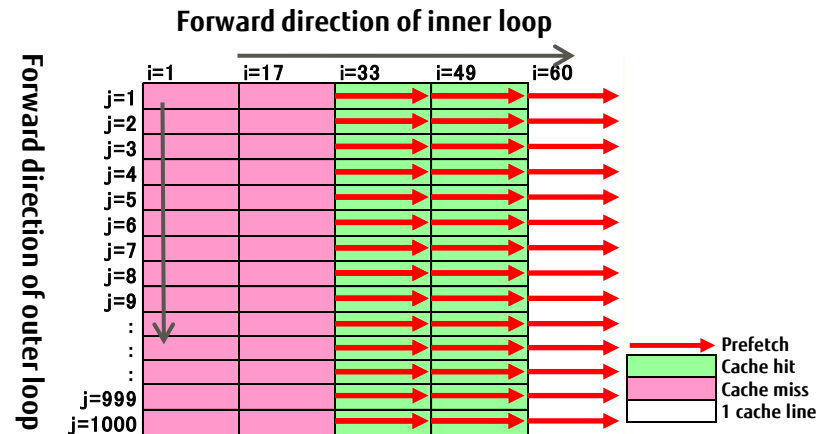
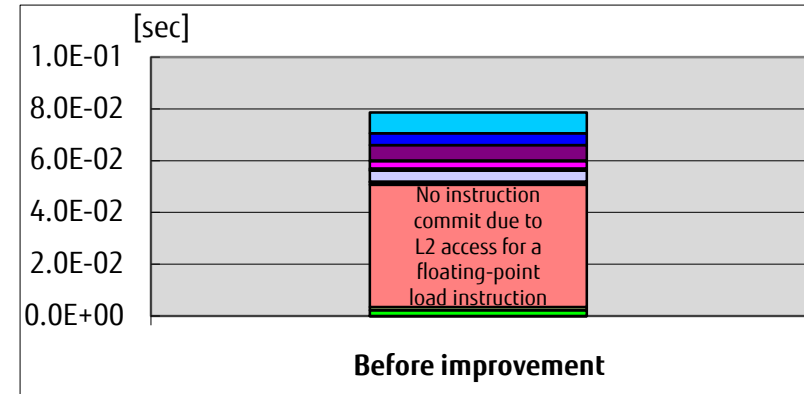
Source code before improvement

```

41  subroutine sub(scalar, isize)
42  parameter(n=1000)
43  integer n
44  real*8 a(n,n), b(n,n), c(n,n), scalar
45  common /com/a,b,c
46
47  <<< Loop-information Start >>>
48  <<< [PARALLELIZATION]
49  <<< Standard iteration count: 2
50  <<< Loop-information End >>>
51  1 pp do j=1,n
52  <<< Loop-information Start >>>
53  <<< [OPTIMIZATION]
54  <<< SIMD(VL: 4)
55  <<< SOFTWARE PIPELINING
56  <<< Loop-information End >>>
57  2 p 4v do i=1, isize, 4
58  2 p 4v a(i,j) = b(i,j) + scalar * c(i,j)
59  2 p 4v a(i+1,j) = b(i+1,j) + scalar * c(i+1,j)
60  2 p 4v a(i+2,j) = b(i+2,j) + scalar * c(i+2,j)
61  2 p 4v a(i+3,j) = b(i+3,j) + scalar * c(i+3,j)
62  2 p 4v enddo
63  1 p enddo

```

Number of elements in first dimension: 1000
 Loop iteration count: 15 iterations (isize = 60)
 Access continuity is broken when outer loop j is incremented.



Cache

	L1L miss rate (effective instruction)	L1D miss rate(/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	4.92%	8.88E+07	95.13%	4.87%	0.00%	0.00%	1.29E+04	289.19	0.08

The L1D miss dm percentage is high, and prefetching is not effective.

Effects of Prefetching for an Outer Loop(Optimization Control Line Tuning)

To conceal the cost at the prefetching rise time, the **PREFETCH_READ** and **PREFETCH_WRITE** specifiers were used to generate a prefetch for the arrays in an outer loop. This results in improvement of the following event: No instruction commit due to L2 access for a floating-point load instruction.

Source code after improvement (optimization control line tuning)

```

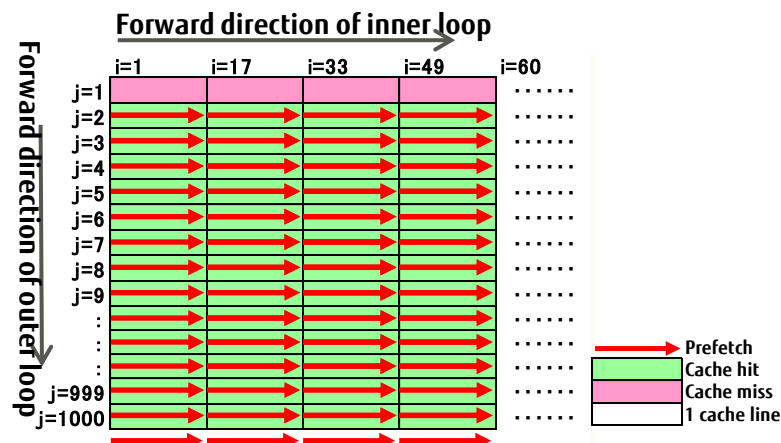
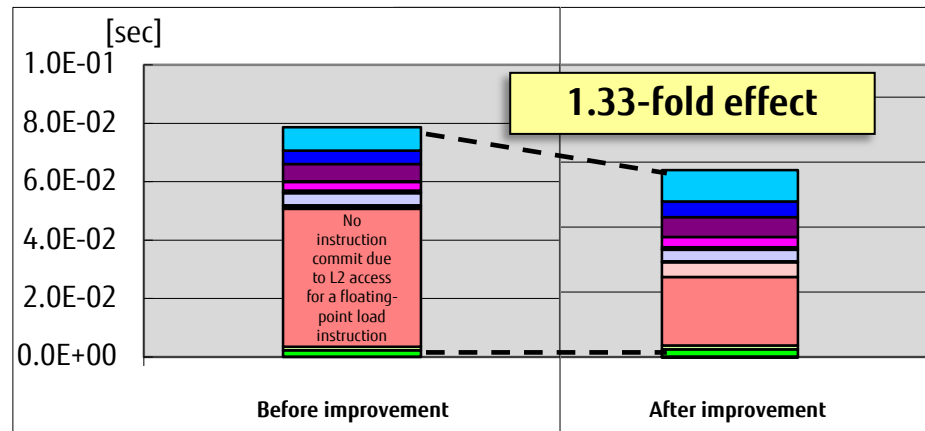
41  subroutine sub(scalar, isize)
42  parameter(n=1000)
43  integer n
44  real*8 a(n,n), b(n,n), c(n,n), scalar
45  common /com/a,b,c
46
47  <<< Loop-information Start >>>
48  <<< [PARALLELIZATION]
49  <<< Standard iteration count: 2
50  <<< [OPTIMIZATION]
51  <<< PREFETCH : 6
52  <<< c: 2, b: 2, a: 2
53  <<< Loop-information End >>>
54  1 pp do j=1,n
55  <<< Loop-information Start >>>
56  <<< [OPTIMIZATION]
57  <<< SIMD(VL: 4)
58  <<< SOFTWARE PIPELINING
59  <<< PREFETCH : 6
60  <<< c: 2, b: 2, a: 2
61  <<< Loop-information End >>>
62  2 p 2v do i=1, isize, 4
63  2 p 2 !OCL PREFETCH_WRITE(a(i,j+1), level=1)
64  2 p 2 !OCL PREFETCH_READ(b(i,j+1), level=1)
65  2 p 2 !OCL PREFETCH_READ(c(i,j+1), level=1)
66  2 p 2v a(i,j) = b(i,j) + scalar * c(i,j)
67  2 p 2v a(i+1,j) = b(i+1,j) + scalar * c(i+1,j)
68  2 p 2v a(i+2,j) = b(i+2,j) + scalar * c(i+2,j)
69  2 p 2v a(i+3,j) = b(i+3,j) + scalar * c(i+3,j)
70  2 p 2v enddo
71  1 p enddo
    
```

Prefetching for array in next iteration of outer loop

Cache

	L1I miss rate (effective instruction)	L1D miss rate(/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	4.92%	8.88E+07	95.13%	4.87%	0.00%	0.00%	1.29E+04	289.19	0.08
After improvement	0.00%	4.74%	8.57E+07	13.91%	0.19%	85.91%	0.00%	1.31E+04	382.23	0.12

The L1D miss dm percentage decreased.



Use of software prefetch

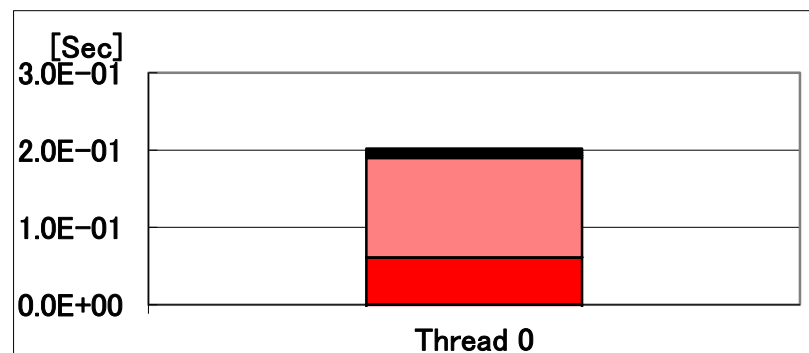
In case of sequential access, hardware prefetching may not be effective even if compiler option `-Kprefetch_sequential=auto` is effective.

When L1D miss dm rate or L2 miss dm rate is high, performance may improve with `-Kprefetch_sequential=soft` specified (software prefetch will be effective)

L1D miss dm rate and L2 miss dm rate are high

Cache

	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss dm rate(/L2 miss)
Thread 0	33.89%	66.11%	0.00%	33.64%



翻訳時オプション	機能説明
<code>-Kprefetch_sequential=auto</code>	The compiler automatically selects whether to use hardware-prefetch or to create prefetch instructions for array data that is accessed sequentially within a loop. <code>-Kprefetch_sequential=auto</code> is effective only when the <code>-O1</code> option or higher is set. The default when the <code>-O2</code> option or higher is set is <code>-Kprefetch_sequential=auto</code> .
<code>-Kprefetch_sequential=soft</code>	The compiler does not use hardware-prefetch, but rather creates prefetch instructions for array data that is accessed sequentially within a loop. <code>-Kprefetch_sequential=soft</code> is effective only when the <code>-O1</code> option or higher is set.
<code>-Kprefetch_nosequential</code>	Prefetch instructions are not generated for array data that is accessed sequentially within a loop. The default when the <code>-O0</code> or <code>-O1</code> option is set is <code>-Kprefetch_nosequential</code> .

Improvement in Data Access Wait (Reduced Amount of Access)

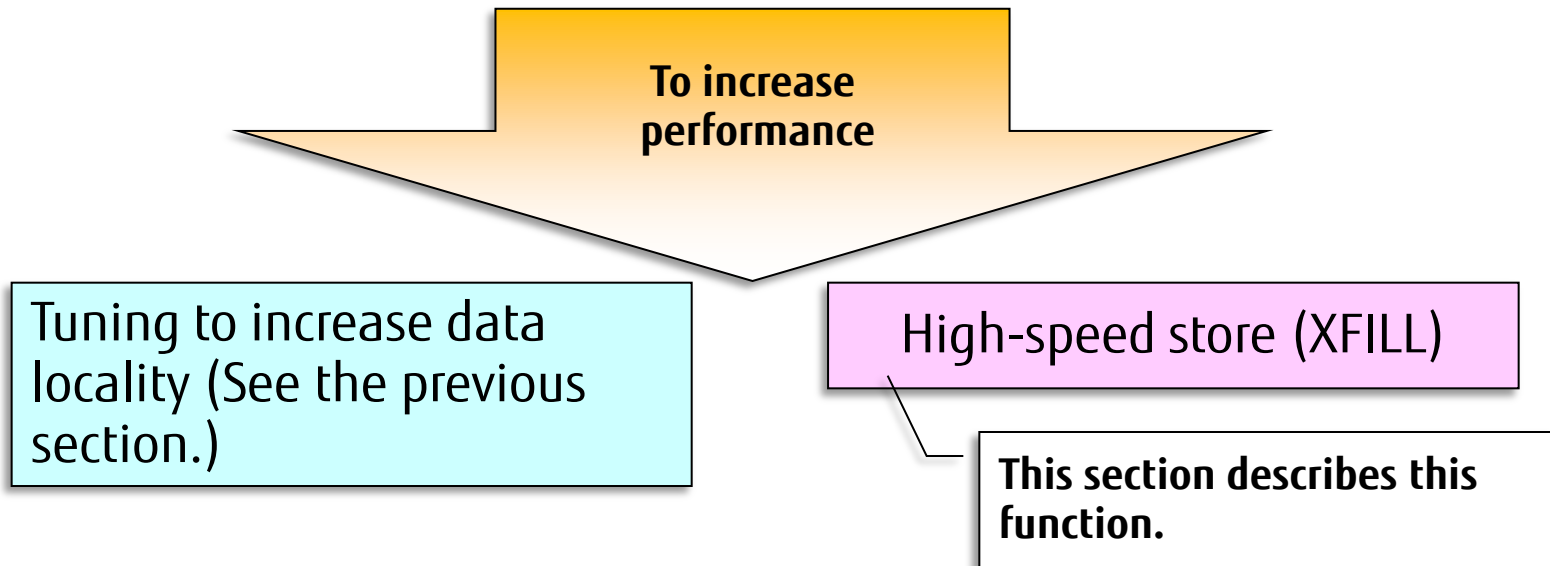
- Memory Throughput and Amount of Memory Access
- High-speed Store (XFILL)

Memory Throughput and Amount of Memory Access

■ Amount of memory access:

(number of L2 cache misses + L2 writebacks) x 256 Byte (line size)

The performance of a program with a memory throughput bottleneck does not increase unless the program is tuned to decrease the amount of memory access or the number of L2 cache misses.



High-speed Store (XFILL)

- What Is High-speed Store (XFILL)?
- XFILL (Before Improvement)
- Effects of XFILL (Optimization Control Line Tuning)
- Effects of XFILL (Compiler Options Tuning)

What Is High-speed Store (XFILL)?

What is high-speed store (XFILL)?

This function reserves a cache line for cache write operations (contents with indefinite values). The function helps reduce the number of cache lines read from memory to increase the performance of a program with a memory throughput bottleneck.

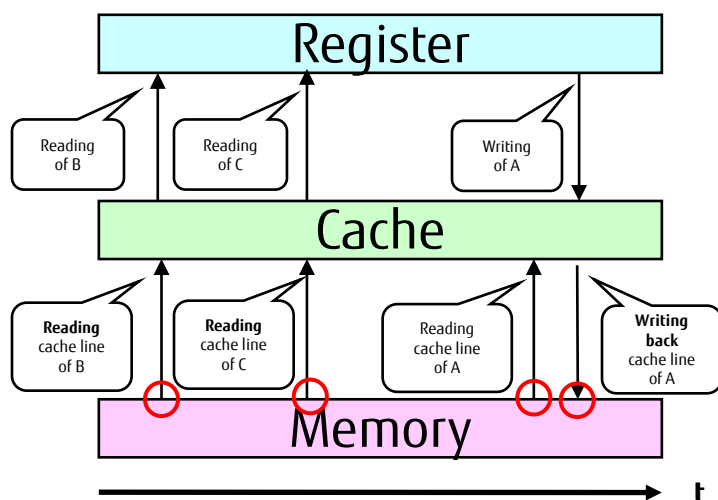
Operating conditions

- The array that is the store target has no dependency between iteration cycles.
- Arrays with definitions are not referenced.
- Memory is accessed contiguously.

Example)

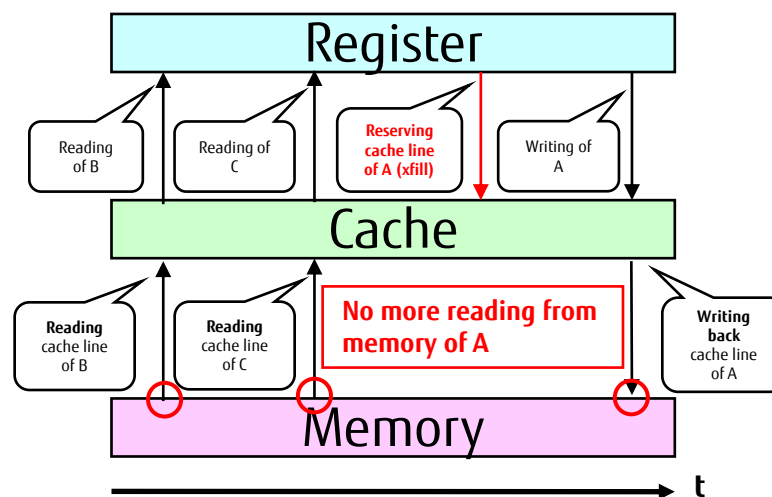
```
DO I = 1, N  
  A(I) = B(I) + C(I)  
END DO
```

XFILL not used



Total number of memory access times: **4**

XFILL used



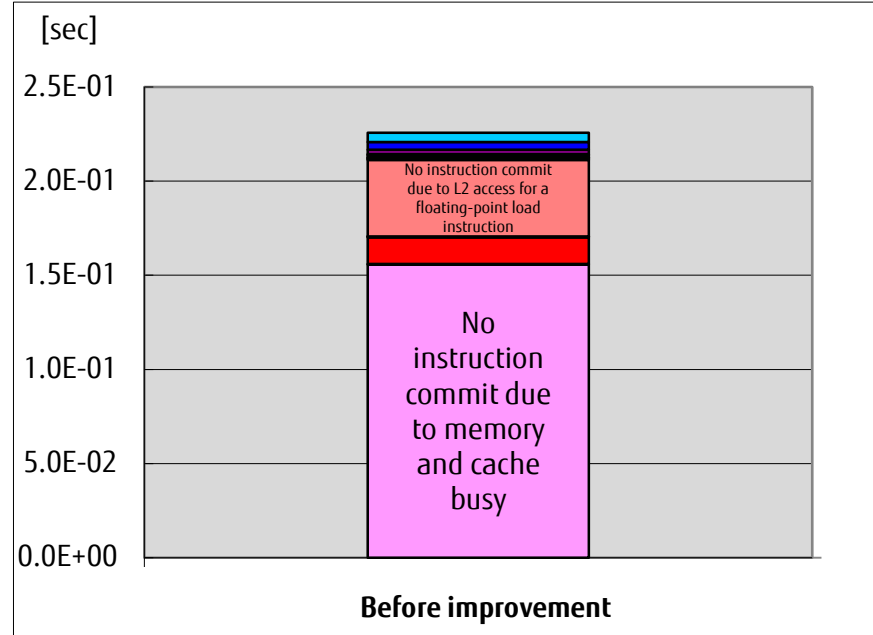
Total number of memory access times: **3**

XFILL (Before Improvement)

Memory throughput is a bottleneck because a program has a heavy load on memory access. Consequently, data access wait is a frequent event.

Source code before improvement

```
39      <<< Loop-information Start >>>
      <<< [PARALLELIZATION]
      <<< Standard iteration count: 942
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< SOFTWARE PIPELINING
      <<< Loop-information End >>>
40  1 pp 8v do i=1,n
41  1 p 8v  a(i) = b(i) + c(i)*d
42  1 p 8v  enddo
```



Cache

	L1I miss rate (effective instruction)	L1D miss rate(/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.01%	3.13%	9.38E+07	0.79%	99.21%	0.00%	3.13%	9.39E+07	106.37	141.89

Memory throughput is a bottleneck.

Effects of XFILL (Optimization Control Line Tuning)

The specification of the **XFILL specifier** eliminated the reading of cache lines from memory by a store instruction. This reduced the L2 miss. As a result, there was improvement in data access wait.

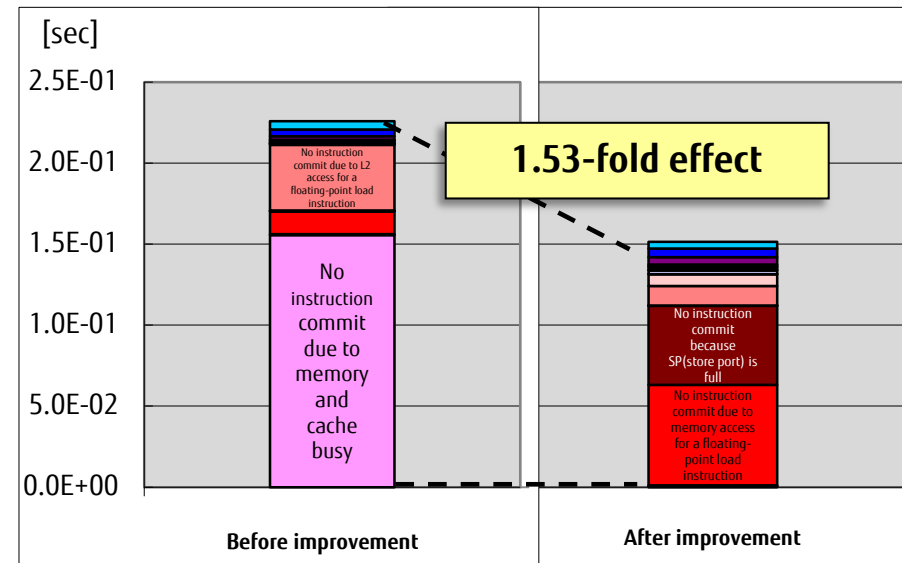
Source code after improvement (optimization control line tuning)

```

39      !ocl xfill
      <<< Loop-information Start >>>
      <<< [PARALLELIZATION]
      <<< Standard iteration count: 942
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< SOFTWARE PIPELINING
      <<< PREFETCH : 2
      <<< a: 2
      <<< XFILL : 2
      <<< a: 2
      <<< 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
    
```

Cache

	L1I miss rate (effective instruction)	L1D miss rate(/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.01%	3.13%	9.38E+07	0.79%	99.21%	0.00%	3.13%	9.39E+07	106.37	141.89
After improvement	0.01%	3.09%	9.38E+07	29.49%	65.74%	4.77%	2.06%	6.26E+07	158.73	158.85



Even after the improvement, a memory throughput bottleneck remains, but the use of XFILL has reduced the L2 miss by 1/3.

XFILL (Optimization Control Line Tuning)

Here, specify the following optimization control line.

Optimization control specifiers	Meaning	Optimization control line that can be specified			
		Program unit	DO loop unit	Statement unit	Array assignment statement unit
XFILL[(m1)]	Gives an instruction to generate an XFILL instruction. m1 is a decimal number in a range of 1 to 100 that indicates the number of lines of the cache.	No	Yes	No	Yes
NOXFILL	Gives an instruction not to generate an XFILL instruction.	No	Yes	No	Yes

◆ Notes

- The XFILL instruction is output for array data that is stored in a loop. However, it is not output for arrays referenced in the same loop, arrays accessed non-sequentially, and arrays stored in IF construct.
- No prefetch instruction is output to the L2 cache when the XFILL instruction is output.
- The following optimization methods cannot be applied because loops are transformed to always store the cache lines reserved by the XFILL instructions. For this reason, execution performance may deteriorate.
 - Loop unrolling
 - Loop striping
- Execution performance may also deteriorate in the following case:
 - Loop with a few iterations

You can achieve effects similar to optimization control line tuning by specifying the following compiler options.

Compiler options	Description of function
-K{ XFILL[=<i>N</i>] NOXFILL } $1 \leq N \leq 100$	<p>Gives an instruction regarding array data that is only written in a loop, to generate an instruction (XFILL instruction) that reserves a cache line for cache writing without loading data from memory. <i>N</i> specifies the data that is <i>N</i> cache lines away as the target of the XFILL instruction.</p> <p>You can specify a value in a range of 1 to 100 for <i>N</i>. If the specification of <i>N</i> is omitted, the compiler automatically determines a value.</p> <p>This option has meaning in cases where -O2 or a higher option is valid. The default is -KNOXFILL.</p>

■ Use example (source code before improvement)

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

Improvement in Operation Wait (Instruction Scheduling Improvement)

- Factors Hindering Instruction Scheduling
- Hindering Factor: Improvement of a Loop Containing an IF Statement
- Hindering Factor: Improvement in Data Dependency
- Hindering Factor: Improvement of a Loop with a Few Iterations

The following factors hinder instruction scheduling.

- Loop containing an IF construct
- Data dependency between iteration cycles
 - **Loop that has data dependency**
 - **Loop that has an unclear definition reference relationship**
 - **Loop containing pointer variables**
- Loop with a few iterations

Hindering Factor: Improvement of a Loop Containing an IF Construct

- SIMD Extensions with the Mask (Basics)
- SIMD Extensions with the Mask (Application)

SIMD Extensions with the Mask (Basics)

- SIMD Extensions with the Mask (Before Improvement)
- Effects of SIMD Extensions with the Mask (Optimization Control Line Tuning)
- SIMD Extensions with the Mask (Optimization Control Line)
- Effects of SIMD Extensions with the Mask (Compiler Options Tuning)

SIMD Extensions with the Mask (Before Improvement)

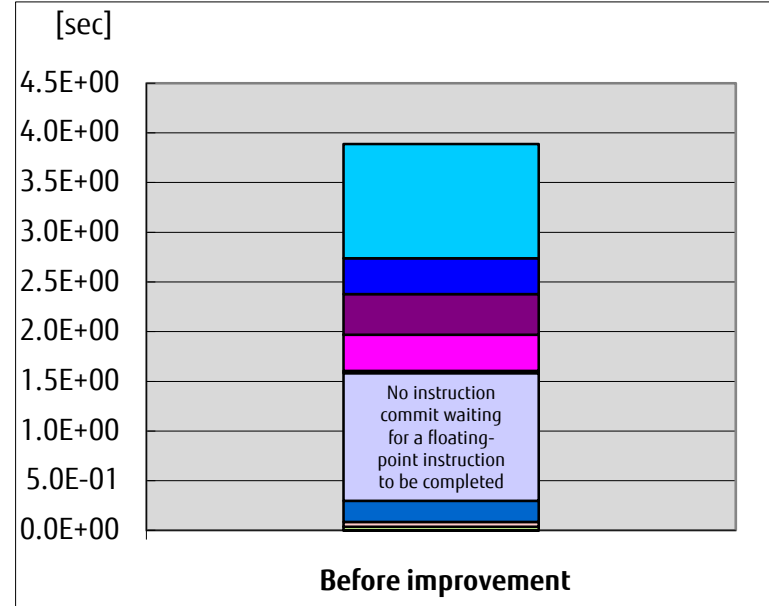
SIMD optimization and software pipelining are not facilitated because the loop contains an IF construct. Consequently, the following is a frequent event: No instruction commit waiting for a floating-point instruction to be completed.

Source code before improvement

```

95  1      !$omp do
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<<  PREFETCH   : 32
      <<<   b: 16, a: 16
      <<< Loop-information End >>>
96  2 p 8s      do i=1,n1
97  3 p 8m      if (p(i) > q) then
98  3 p 8s      a(i) = c0+b(i)*(c1+b(i)*(c2+b(i)*(c3+b(i)*c4)))
99  3 p 8v      endif
100 2 p 8v      enddo
101 1      !$omp enddo
    
```

True ratio of 90%



SIMD

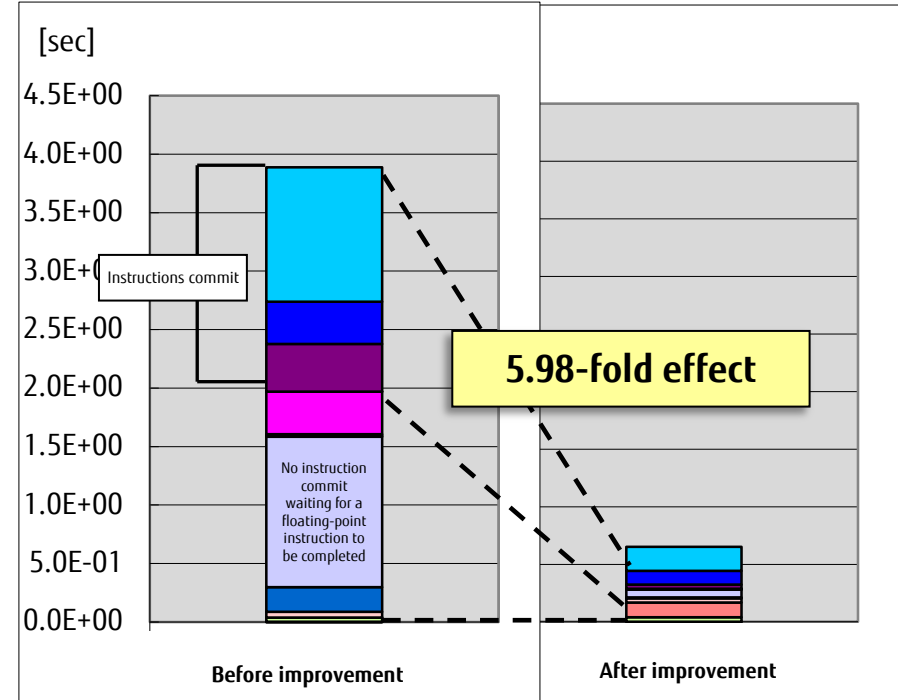
	SIMD instruction rate (effective instruction)	SIMD floating point instruction rate (/SIMD target floating point instruction)	SIMD integer instruction rate (/SIMD target integer instruction)	SIMD load-store instruction rate (/SIMD target load-store instruction)
Before improvement	0.00%	0.00%	0.00%	0.00%

There is no SIMD optimization.

Effects of SIMD Extensions with the Mask (Optimization Control Line Tuning) **FUJITSU**

Specification of the **SIMD specifier** facilitates software pipelining through SIMD extensions with the mask. The result is a reduction in effective instruction, a decrease in instruction commits, facilitation of instruction scheduling, and a significant improvement in the following event: No instruction commit waiting for a floating-point instruction to be completed.

Source code after improvement (optimization control line tuning)			
94	1	!ocl simd	Specifies SIMD optimization
95	1	!\$omp do	
		<<< Loop-information Start >>>	
		<<< [OPTIMIZATION]	
		<<< SIMD(VL: 4)	
		<<< SOFTWARE PIPELINING	
		<<< PREFETCH : 8	
		<<< a: 8	
		<<< Loop-information End >>>	
96	2	p 6v do i=1,n1	
97	3	p 6v if (p(i) > q) then	
98	3	p 6v a(i) = c0+b(i)*(c1+b(i)*(c2+b(i)*(c3+b(i)*c4)))	
99	3	p 6v endif	
100	2	p 6v enddo	
101	1	!\$omp enddo	



	Effective instruction
Before improvement	2.22E+11
After improvement	4.15E+10

Facilitating SIMD optimization reduced effective instruction.

SIMD

	SIMD instruction rate (effective instruction)	SIMD floating point instruction rate (/SIMD target floating point instruction)	SIMD integer instruction rate (/SIMD target integer instruction)	SIMD load-store instruction rate (/SIMD target load-store instruction)
Before improvement	0.00%	0.00%	0.00%	0.00%
After improvement	77.36%	100.00%	0.00%	97.17%

Here, specify the following optimization control line.

optimization control specifiers	Meaning	Optimization control line that can be specified			
		Program unit	DO loop unit	Statement unit	Array assignment statement unit
SIMD	Enables SIMD optimization.	Yes	Yes	No	Yes

◆ Notes

- SIMD optimization may not be realized depending on the operation type and loop structure.

You can achieve effects similar to optimization control line tuning by specifying the following compiler options.

Compiler options	Description of function
-Ksimd=2	Gives an instruction to generate an object that uses a SIMD expansion instruction, in addition to the -Ksimd=1 function, for loops containing an IF construct, etc.

■ Use example (source code before improvement)

```
$ frtpx -Kfast,parallel sample.f90 -Ksimd=2
```

◆ Notes

- Execution performance may deteriorate depending on the true ratio of the IF construct.
- The execution of an instruction that should not be executed from the perspective of program logic may cause an error because expressions inside IF construct are speculatively executed.

SIMD Extensions with the Mask (Application)

- SIMD Extensions with the Mask (Before Improvement)
- Effects of SIMD Extensions with the Mask: Process 1 (Optimization Control Line Tuning)
- Effects of SIMD Extensions with the Mask: Process 2 (Optimization Control Line Tuning + Source Tuning)
- Effects of SIMD Optimization through Loop Unswitching (Before Improvement)
- Effects of SIMD Optimization through Loop Unswitching (After Improvement)
- Appearance of Code Optimized by Loop Unswitching
- Array Division (Before Improvement)
- Effects of Array Division (Source Tuning)

SIMD Extensions with the Mask (Before Improvement)

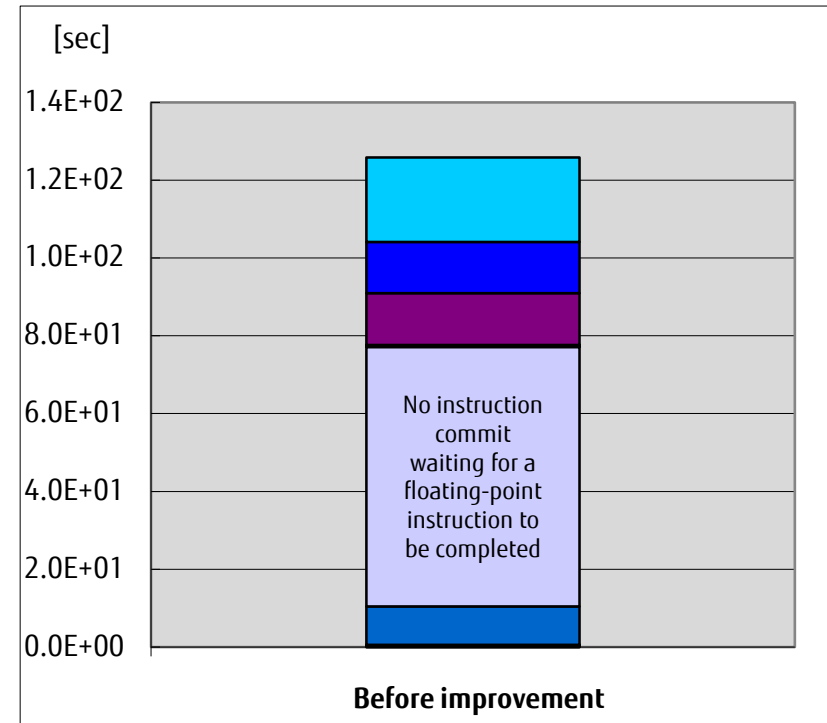
SIMD optimization and software pipelining are not facilitated because the loop contains an IF construct. Consequently, the following is a frequent event: No instruction commit waiting for a floating-point instruction to be completed.

Source code before improvement

```
63 1  !$omp do
    <<< Loop-information Start >>>
    <<< [OPTIMIZATION]
    <<< PREFETCH : 12
    <<< b: 2, a: 2, b1: 2, a1: 2, b2: 2, a2: 2
    <<< Loop-information End >>>
64 2 p s    do i=1,n1
65 3 p m      if (p(i) > q) then
66 3 p s      a(i) = c0 + b(i)*(c1 + b(i)*(c2 + b(i)*(c3 + b(i)*
67 3      &      (c4 + b(i)*(c5 + b(i)*(c6 + b(i)*(c7 + b(i)*
68 3      &      (c8 + b(i)*c9))))))
69 3 p v      endif
70 3 p s      if (p(i) < q) then
71 3 p s      a1(i) = c0+b1(i)/(c1+b1(i)/(c2+b1(i)/(c3+b1(i)/(c4+b1(i)/
72 3      &      (c5+b1(i)/(c6+b1(i)/(c7+b1(i)/(c8+b1(i)/c9))))))
73 3 p s      a2(i) = c0+b2(i)/(c1+b2(i)/(c2+b2(i)/(c3+b2(i)/(c4+b2(i)/
74 3      &      (c5+b2(i)/(c6+b2(i)/(c7+b2(i)/(c8+b2(i)/c9))))))
75 3 p v      endif
76 2 p v    enddo
```

True ratio of
98%

True ratio of
2%



SIMD

	SIMD instruction rate (effective instruction)	SIMD floating point instruction rate (/SIMD target floating point instruction)	SIMD integer instruction rate (/SIMD target integer instruction)	SIMD load-store instruction rate (/SIMD target load-store instruction)
Before improvement	0.00%	0.00%	0.00%	0.00%

There is no SIMD optimization.

Effects of SIMD Extensions with the Mask: Process 1 (Optimization Control Line Tuning) FUJITSU

Specification of the **SIMD specifier** facilitates software pipelining through SIMD extensions with the mask. This results in an improvement in the following event: No instruction commit waiting for a floating-point instruction to be completed. However, an adverse effect of SIMD extensions with the mask was an increased effective instruction, which was the cause of an increase in instruction commits.

Source code of improvement 1 (optimization control line tuning)

```

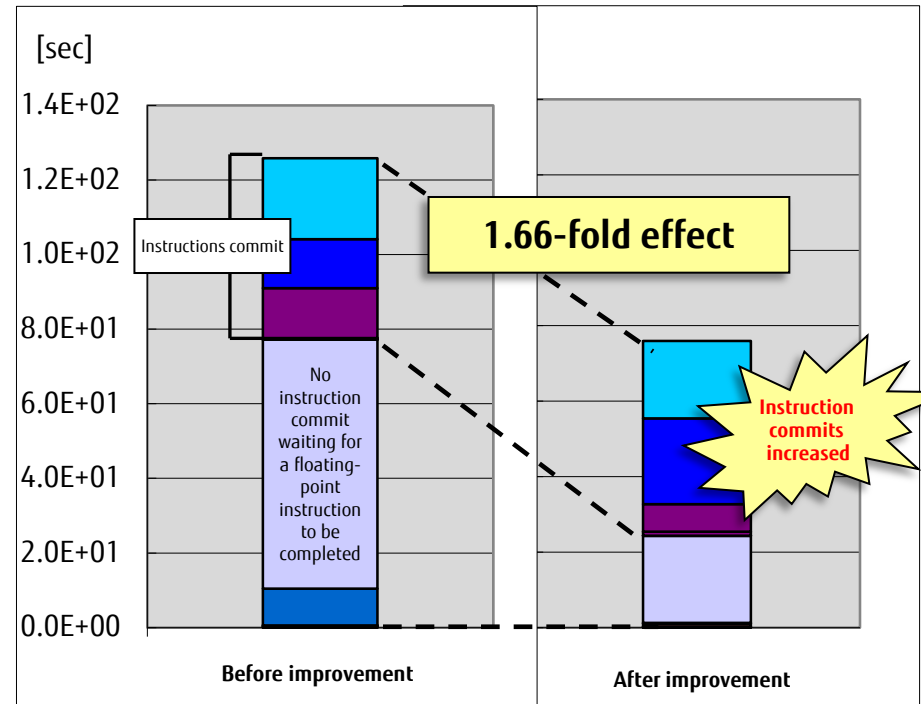
63 1      !$omp do
64 1      !ocl simd
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< SOFTWARE PIPELINING
      <<< PREFETCH : 6
      <<< a: 2, a1: 2, a2: 2
      <<< Loop-information End >>>

65 2 p v      do i=1,n1
66 3 p v      if (p(i) > q) then
67 3 p v          a(i) = c0 + b(i)*(c1 + b(i)*(c2 + b(i)*(c3 + b(i)*
68 3          & (c4 + b(i)*(c5 + b(i)*(c6 + b(i)*(c7 + b(i)*
69 3          & (c8 + b(i)*c9))))))
70 3 p v      endif
71 3 p v      if (p(i) < q) then
72 3 p v          a1(i) = c0+b1(i)/(c1+b1(i)/(c2+b1(i)/(c3+b1(i)/(c4+b1(i)/
73 3          & (c5+b1(i)/(c6+b1(i)/(c7+b1(i)/(c8+b1(i)/c9))))))
74 3 p v          a2(i) = c0+b2(i)/(c1+b2(i)/(c2+b2(i)/(c3+b2(i)/(c4+b2(i)/
75 3          & (c5+b2(i)/(c6+b2(i)/(c7+b2(i)/(c8+b2(i)/c9))))))
76 3 p v      endif
77 2 p v      enddo
    
```

Specifies SIMD optimization

True ratio of 98%

True ratio of 2%



	Effective instruction
Before improvement	4.19E+12
After improvement	5.14E+12

SIMD

	SIMD instruction rate (effective instruction)	SIMD floating point instruction target floating point ins
Before improvement	0.00%	0.00%
After improvement	95.68%	100.00%

SIMD optimization was facilitated, but this included SIMD optimization for IF construct that have a low true ratio, so effective instruction decreased only slightly because redundant instructions were issued.

The adverse effect of SIMD extensions with the mask could be reduced in the next step, which is loop division and SIMD optimization of only IF construct that have a high true ratio.
This results in a decreased effective instruction and improved execution performance.

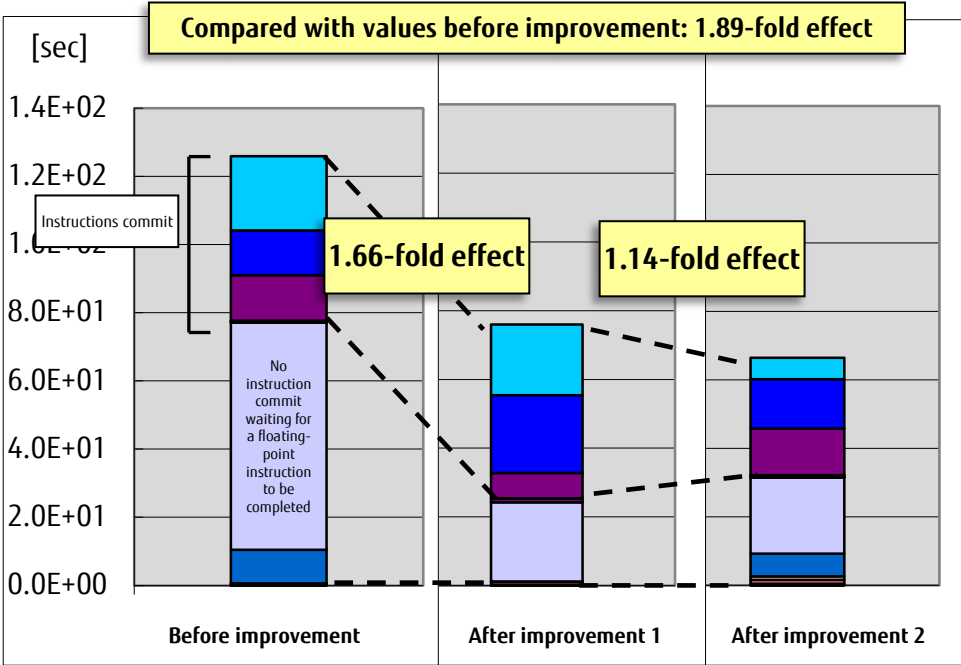
Source code of improvement 2 (optimization control + source tuning)

```
63 1 !$omp do
64 1 !ocl simd
    <<< Loop-information Start >>>
    <<< [OPTIMIZATION]
    <<< SIMD(VL: 4)
    <<< SOFTWARE PIPELINING
    <<< PREFETCH :12
    <<< a: 12
    <<< Loop-information End >>>
65 2 p 6v do i=1,n1
66 3 p 6v if (p(i) > q) then
67 3 p 6v a(i) = c0 + b(i)*(c1 + b(i)*(c2 + b(i)*(c3 + b(i)*
68 3 & (c4 + b(i)*(c5 + b(i)*(c6 + b(i)*(c7 + b(i)*
69 3 & (c8 + b(i)*c9))))))
70 3 p 6v endif
71 2 p 6v enddo
72 1 !$omp enddo
73 1 !$omp do
    <<< Loop-information Start >>>
    <<< [OPTIMIZATION]
    <<< PREFETCH :8
    <<< b1: 2, a1: 2, b2: 2, a2: 2
    <<< Loop-information End >>>
74 2 p 5 do i=1,n1
75 3 p m if (p(i) < q) then
76 3 p s a1(i) = c0+b1(i)/(c1+b1(i)/(c2+b1(i)/(c3+b1(i)/(c4+b1(i)/
77 3 & (c5+b1(i)/(c6+b1(i)/(c7+b1(i)/(c8+b1(i)/c9))))))
78 3 p s a2(i) = c0+b2(i)/(c1+b2(i)/(c2+b2(i)/(c3+b2(i)/(c4+b2(i)/
79 3 & (c5+b2(i)/(c6+b2(i)/(c7+b2(i)/(c8+b2(i)/c9))))))
80 3 p v endif
81 2 p v enddo
82 1 !$omp enddo
```

True ratio of 98%
⇒ SIMD optimization

Loop fission

True ratio of 2%
⇒ No SIMD optimization



	SIMD instruction rate	SIMD floating point instruction rate	SIMD integer instruction rate	SIMD load-store instruction rate	Effective instruction
Before improvement	0.00%	0.00%	0.00%	0.00%	4.19E+12
After improvement 1	95.68%	100.00%	0.00%	99.65%	5.14E+12
After improvement 2	18.16%	38.95%	0.00%	40.79%	2.15E+12

With SIMD optimization of only IF construct with a high true ratio, the effective instruction decreased.

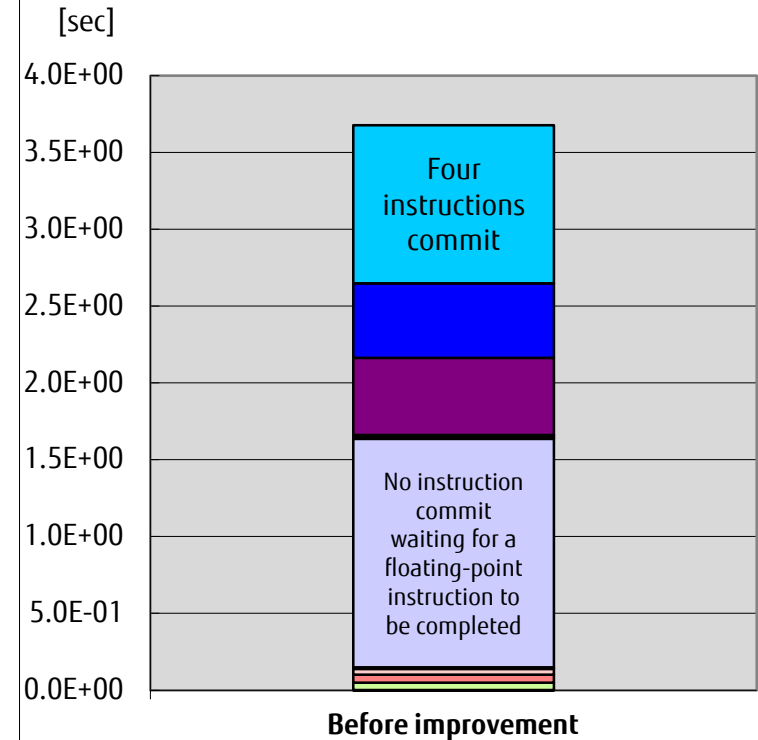
Effects of SIMD Optimization through Loop Unswitching (Before Improvement)

There is neither SIMD optimization nor effective software pipelining because the innermost loop contains an IF construct. Consequently, the following is a frequent event: No instruction commit waiting for a floating-point instruction to be completed.

Before optimization

```

97 1      !$omp do
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< UNSWITCHING
      <<< Loop-information End >>>
98 2 p 4s  do i=1,n1
99 2
100 3 p 4v  if (n1 >= q) then
101 3 p 4v    a(i) = c0+b(i)*(c1+b(i)*(c2+b(i)*
              (c3+b(i)*c4)))
102 3 p 4v  endif
103 2
104 3 p 4v  if(n1 > r) then
105 3 p 4v    a(i) = c0*b(i)/(c1*b(i)/(c2*b(i)/
              (c3*b(i)/c4)))
106 3 p 4s  endif
107 2
108 3 p 4s  if(n1 < s) then
109 3 p 4s    a(i) = c0+b(i)/(c1+b(i)/(c2+b(i)/
              (c3+b(i)/c4)))
110 3 p 4v  endif
111 2 p 4v  enddo
112 1      !$omp enddo
    
```



	SIMD instruction rate (effective instruction)	SIMD floating point instruction rate (/SIMD target floating point instruction)	SIMD integer instruction rate (/SIMD target integer instruction)	SIMD load-store instruction rate (/SIMD target load-store instruction)
Before improvement	0.00%	0.00%	0.00%	0.00%

There is no SIMD optimization

Effects of SIMD Optimization through Loop Unswitching (After Improvement)

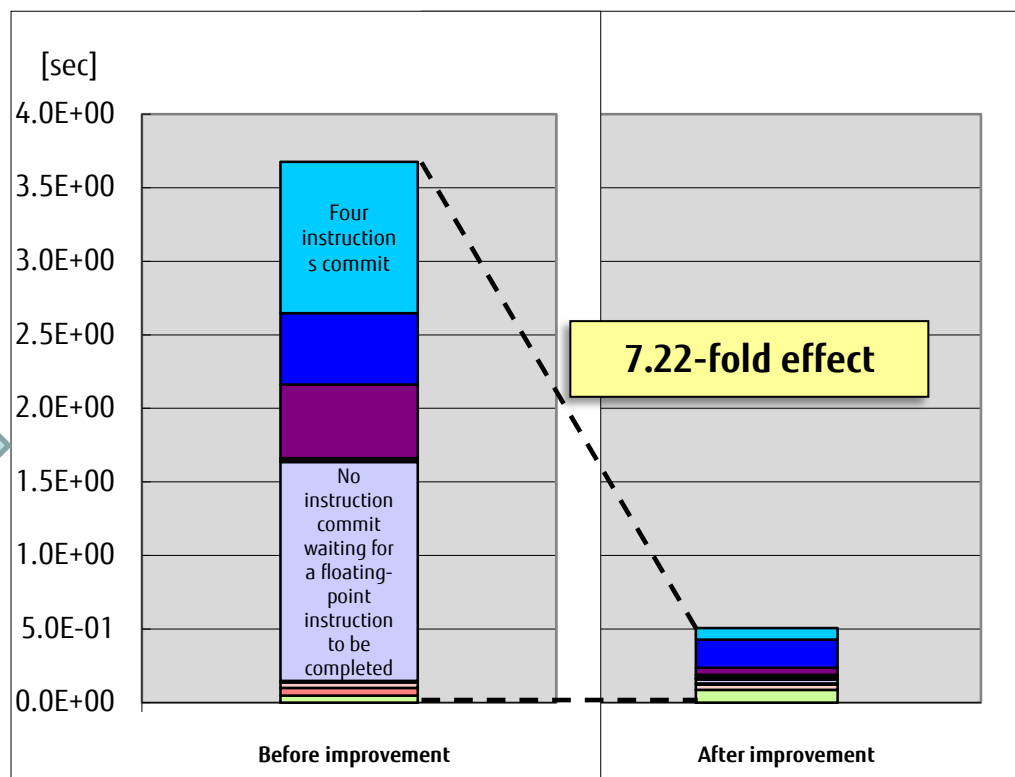
Specification of loop unswitching for the IF construct improves instruction scheduling and facilitates SIMD optimization and software pipelining. The result is a significant improvement in the following event: No instruction commit waiting for a floating-point instruction to be completed.

After optimization

```

97 1      !$omp do
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< SOFTWARE PIPELINING
      <<< UNSWITCHING
      <<< Loop-information End >>>
98 2 p 4v  do i=1,n1
99 2      !ocl unswitching
100 3 p 4v  if (n1 >= q) then
101 3 p 4v    a(i) = c0+b(i)*(c1+b(i)*(c2+b(i)*
              (c3+b(i)*c4)))
102 3 p 4v  endif
103 2      !ocl unswitching
104 3 p 4v  if (n1 > r) then
105 3 p 4v    a(i) = c0*b(i)/(c1*b(i)/(c2*b(i)/
              (c3*b(i)/c4)))
106 3 p 4v  endif
107 2      !ocl unswitching
108 3 p 4v  if (n1 < s) then
109 3 p 4v    a(i) = c0+b(i)/(c1+b(i)/(c2+b(i)/
              (c3+b(i)/c4)))
110 3 p 4v  endif
111 2 p 4v  enddo
112 1      !$omp enddo
    
```

See next page for
appearance of
optimized code



SIMD optimization reduced effective instruction.

	Effective instruction
Before improvement	1.98E+11
After improvement	2.80E+10

	SIMD instruction rate (effective instruction)	SIMD floating point instruction rate (/SIMD target floating point instruction)	SIMD integer instruction rate (/SIMD target integer instruction)	SIMD load-store instruction rate (/SIMD target load-store instruction)
Before improvement	0.00%	0.00%	0.00%	0.00%
After improvement	86.24%	100.00%	0.00%	95.50%

Appearance of Code Optimized by Loop Unswitching

Source code

```

97 1      !$omp do
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< SOFTWARE PIPELINING
      <<< UNSWITCHING
      <<< Loop-information End >>>

98 2 p 4v  do i=1,n1
99 2      !ocl unswitching
100 3 p 4v  if (n1 >= q) then
101 3 p 4v    a(i) = c0+b(i)*(c1+b(i)*(c2+b(i)*
              (c3+b(i)*c4)))
102 3 p 4v  endif
103 2      !ocl unswitching
104 3 p 4v  if(n1 > r) then
105 3 p 4v    a(i) = c0*b(i)/(c1*b(i)/(c2*b(i)/
              (c3*b(i)/c4)))
106 3 p 4v  endif
107 2      !ocl unswitching
108 3 p 4v  if(n1 < s) then
109 3 p 4v    a(i) = c0+b(i)/(c1+b(i)/(c2+b(i)/
              (c3+b(i)/c4)))
110 3 p 4v  endif
111 2 p 4v  enddo
112 1      !$omp enddo
    
```

* Loop unswitching

This optimization pertains to a loop containing an IF construct that has branches with invariable conditions. It places the IF construct outside the loop to create loops used when some or all of the conditions in the IF construct are met and a loop used when none of the conditions are met.

Appearance of optimized code

```

!Pattern (1)
if((condition (1) true).and.(condition (2)
true).and.(condition (3) true))then
  do i=1,n1
    Process (1)
    Process (2)
    Process (3)
  enddo
endif

!Pattern (2)
if((condition (1) true).and.(condition (2)
true).and.(condition (3) false))then
  do i=1,n1
    Process (1)
    Process (2)
  enddo
endif

!Pattern (3)
if((condition (1) true).and.(condition (2)
false).and.(condition (3) true))then
  do i=1,n1
    Process (1)
    Process (3)
  enddo
endif

!Pattern (4)
if((condition (1) true).and.(condition (2)
false).and.(condition (3) false))then
  do i=1,n1
    Process (1)
  enddo
endif

!Pattern (5)
if((condition (1) false).and.(condition (2)
true).and.(condition (3) true))then
  do i=1,n1
    Process (2)
    Process (3)
  enddo
endif

!Pattern (6)
if((condition (1) false).and.(condition (2)
true).and.(condition (3) false))then
  do i=1,n1
    Process (2)
  enddo
endif

!Pattern (7)
if((condition (1) false).and.(condition (2)
false).and.(condition (3) true))then
  do i=1,n1
    Process (3)
  enddo
endif

!Pattern (8)
if((condition (1) false).and.(condition (2)
false).and.(condition (3) false))then
  do i=1,n1
  enddo
endif
    
```

Expanded to 8 if statements (do statements)

Array Division (Before Improvement)

The L1 busy rate is high because indirect load and store are used. Consequently, the following is a frequent event: No instruction commit due to L1D access for a floating-point load instruction.

Source code before improvement

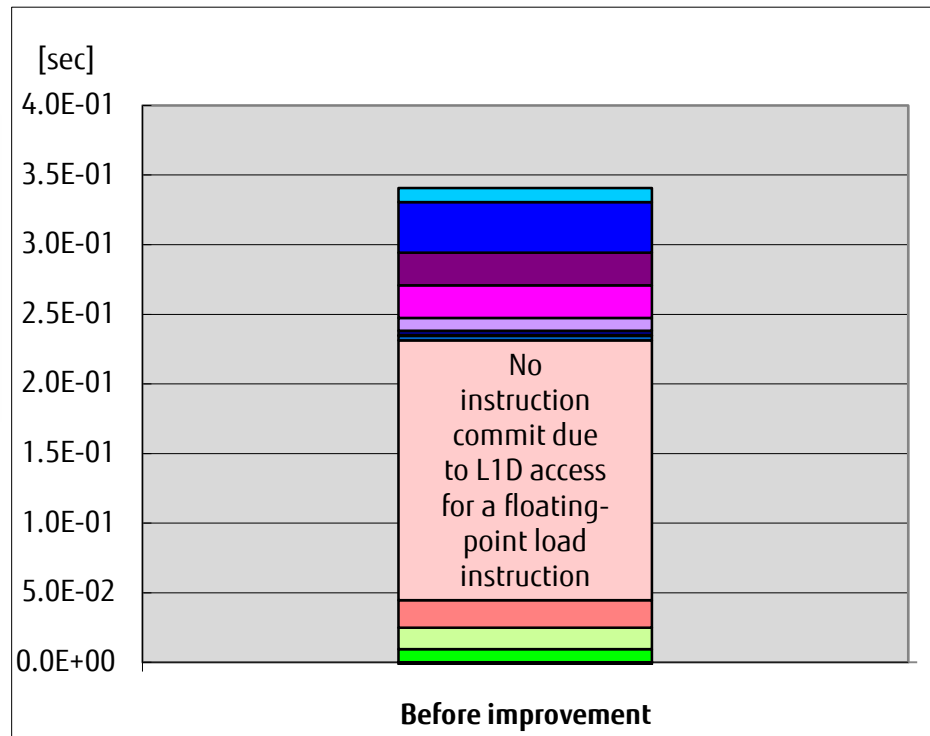
```

28      real*8 a(14,n),b(14,n),c(14,n)
29
    <<< Loop-information Start >>>
    <<< [PARALLELIZATION]
    <<< Standard iteration count: 110
    <<< [OPTIMIZATION]
    <<< SIMD(VL: 4)
    <<< SOFTWARE PIPELINING
    <<< Loop-information End >>>
30  1 pp v   do i=1,n
31  1 p v     a(1,i) = b(1,i) - c(1,i)
32  1 p v     a(2,i) = b(2,i) - c(2,i)
33  1 p v     a(3,i) = b(3,i) - c(3,i)
34  1 p v     a(4,i) = b(4,i) - c(4,i)
35  1 p v     a(5,i) = b(5,i) - c(5,i)
36  1 p v     a(6,i) = b(6,i) - c(6,i)
37  1 p v     a(7,i) = b(7,i) - c(7,i)
38  1 p v     a(8,i) = b(8,i) - c(8,i)
39  1 p v     a(9,i) = b(9,i) - c(9,i)
40  1 p v     a(10,i) = b(10,i) - c(10,i)
41  1 p v     a(11,i) = b(11,i) - c(11,i)
42  1 p v     a(12,i) = b(12,i) - c(12,i)
43  1 p v     a(13,i) = b(13,i) - c(13,i)
44  1 p v     a(14,i) = b(14,i) - c(14,i)
45  1 p v     enddo
46
47      end    :
        
```

Arrays a, b, and c are accessed contiguously. However, the respective arrays themselves (such as a(1,i)) are accessed with a stride of 14 elements per iteration.

⇒ Indirect load and store are used for access with a stride of 8 or more elements.

The L1 busy rate is high because indirect load and store instructions are used.



Memory and cache

	L1 busy rate	L2 busy rate	Memory busy rate
Before improvement	75%	2%	0%

Instruction

	SIMD floating-point load instruction rate		SIMD floating point store instruction rate		SIMD indirect load instruction rate	SIMD indirect store instruction rate	SIMD stride load instruction rate	SIMD stride store instruction rate	SIMD broadcast load instruction rate
	4 SIMD	2 SIMD	4 SIMD	2 SIMD	4 SIMD	4 SIMD	4 SIMD	4 SIMD	4 SIMD
Before improvement	0.03%	0.00%	0.00%	0.00%	22.60%	11.30%	0.00%	0.00%	1.15%

Effects of Array Division (Source Tuning)

Stride load and store are used since the loop is divided in such a way that the arrays are accessed with a stride of seven or fewer elements. This results in improvement of the following event: No instruction commit due to L1 access for a floating-point load instruction.

Source code after improvement (source tuning)

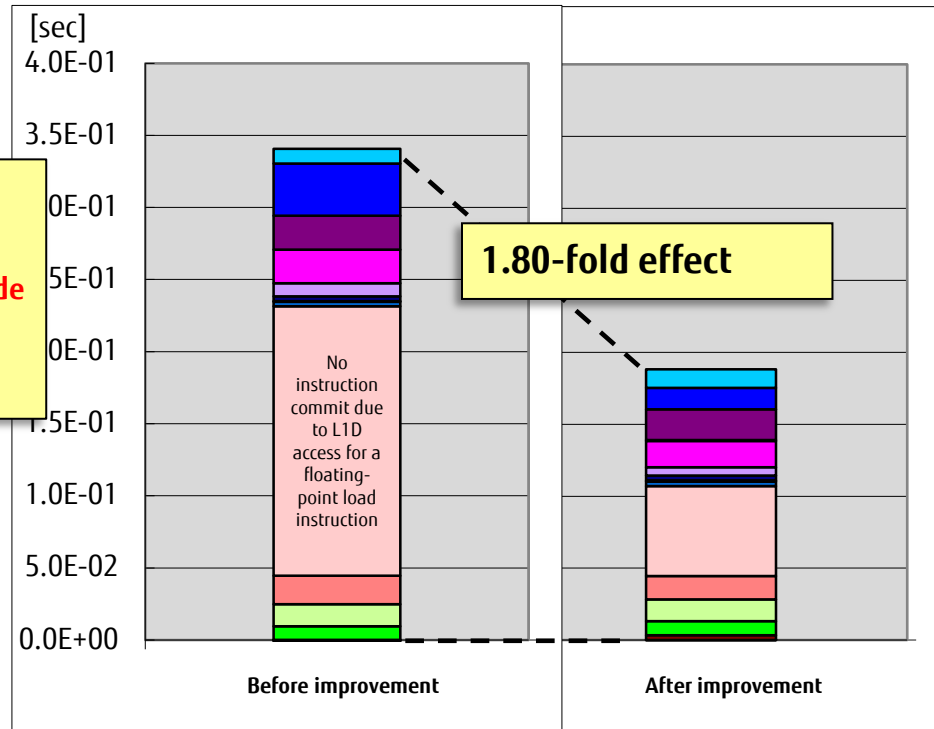
```

30      real*8 a1(7,n),b1(7,n),c1(7,n)
31      real*8 a2(7,n),b2(7,n),c2(7,n)
32
33      <<< Loop-information Start >>>
34      <<< [PARALLELIZATION]
35      <<< Standard iteration count: 1
36      <<< [OPTIMIZATION]
37      <<< SIMD(VL: 4)
38      <<< SOFTWARE PIPELINING
39      <<< Loop-information End >>>
40
41      do i=1,n
42      1 pp 2v      a1(1,i) = b1(1,i) - c1(1,i)
43      1 p 2v      a1(2,i) = b1(2,i) - c1(2,i)
44      1 p 2v      a1(3,i) = b1(3,i) - c1(3,i)
45      1 p 2v      a1(4,i) = b1(4,i) - c1(4,i)
46      1 p 2v      a1(5,i) = b1(5,i) - c1(5,i)
47      1 p 2v      a1(6,i) = b1(6,i) - c1(6,i)
48      1 p 2v      a1(7,i) = b1(7,i) - c1(7,i)
49      1 p 2v      a2(1,i) = b2(1,i) - c2(1,i)
50      1 p 2v      a2(2,i) = b2(2,i) - c2(2,i)
51      1 p 2v      a2(3,i) = b2(3,i) - c2(3,i)
52      1 p 2v      a2(4,i) = b2(4,i) - c2(4,i)
53      1 p 2v      a2(5,i) = b2(5,i) - c2(5,i)
54      1 p 2v      a2(6,i) = b2(6,i) - c2(6,i)
55      1 p 2v      a2(7,i) = b2(7,i) - c2(7,i)
56      enddo
                    
```

Arrays a, b, and c are accessed contiguously. However, the respective arrays themselves (such as a1(1,i)) are accessed with a stride of 7 elements per iteration.

⇒ Stride store is used.

Stride load and store instructions are now used.



Memory and cache

	L1 busy rate	L2 busy rate	Memory busy rate
Before improvement	75%	2%	0%
After improvement	58%	3%	0%

Instruction

	SIMD floating-point load instruction rate		SIMD floating point store instruction rate		SIMD indirect load instruction rate	SIMD indirect store instruction rate	SIMD stride load instruction rate	SIMD stride store instruction rate	SIMD broadcast load instruction rate
	4 SIMD	2 SIMD	4 SIMD	2 SIMD	4 SIMD	4 SIMD	4 SIMD	4 SIMD	4 SIMD
Before improvement	0.03%	0.00%	0.00%	0.00%	22.60%	11.30%	0.00%	0.00%	1.15%
After improvement	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	35.63%	17.81%	0.00%

Hindering Factor: Improvement in Data Dependency

- Loop That Has Data Dependency
- Loop That Has an Unclear Definition Reference Relationship
- Loop Containing Pointer Variables

Loop That Has Data Dependency

- Loop That Has Data Dependency (Before Improvement)
- Loop That Has Data Dependency (Source Tuning)

Loop That Has Data Dependency (Before Improvement)

There is neither SIMD optimization nor effective software pipelining because array a has a data dependency that references data for $i = 2$ or greater as defined when $i = 1$. Consequently, the following is a frequent event: No instruction commit waiting for a floating-point instruction to be completed.

Source code before improvement

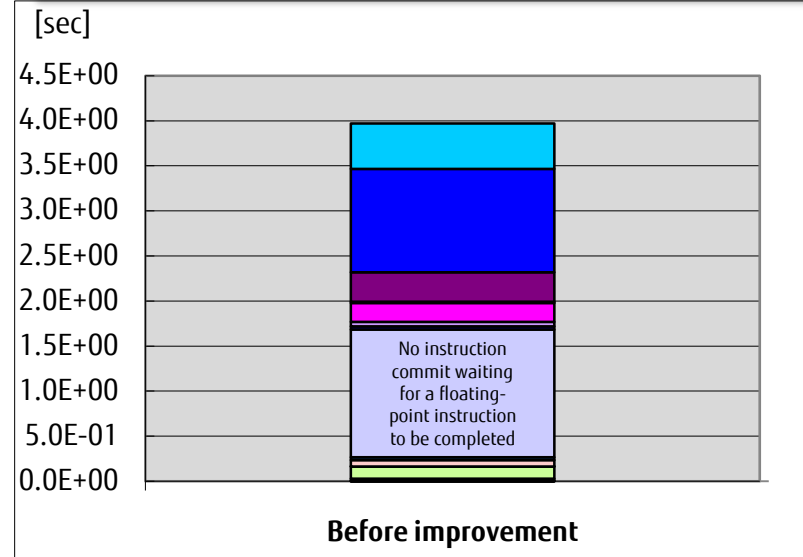
```

<<< Loop-information Start >>>
<<< [PARALLELIZATION]
<<< Standard iteration count: 2
<<< Loop-information End >>>

4  1 pp      do j = 1, m
5  2 p 6s      do i = 1, n
6  2 p 6m      a(i,j) = c0 + a(1,j)*(c1 + b(i,j)*(c2 + b(i,j)*(c3 + b(i,j)*
7  2          & (c4 + b(i,j)*(c5 + b(i,j)*(c6 + b(i,j)*(c7 + b(i,j)*
8  2          & (c8 + b(i,j)*c9))))))
9  2
10 2 p 6v      end do
11 1 p        end do
12           end
    
```

There is dependency between array a on the load side and array a on the store side when $i = 1$ or 2.

There is no effective software pipelining because of data dependency between iteration cycles.



SIMD

	SIMD instruction rate (effective instruction)	SIMD floating point instruction rate (/SIMD target floating point instruction)	SIMD integer instruction rate (/SIMD target integer instruction)	SIMD load-store instruction rate (/SIMD target load-store instruction)
Before improvement	0.00%	0.00%	0.00%	0.00%

There is no SIMD optimization.

Loop That Has Data Dependency (Source Tuning)

SIMD optimization and software pipelining were facilitated through peeling.

The result is a reduction in effective instruction, a decrease in instruction commits, facilitation of instruction scheduling, and a significant improvement in the following event: No instruction commit waiting for a floating-point instruction to be completed.

Source code after improvement (source tuning)

```

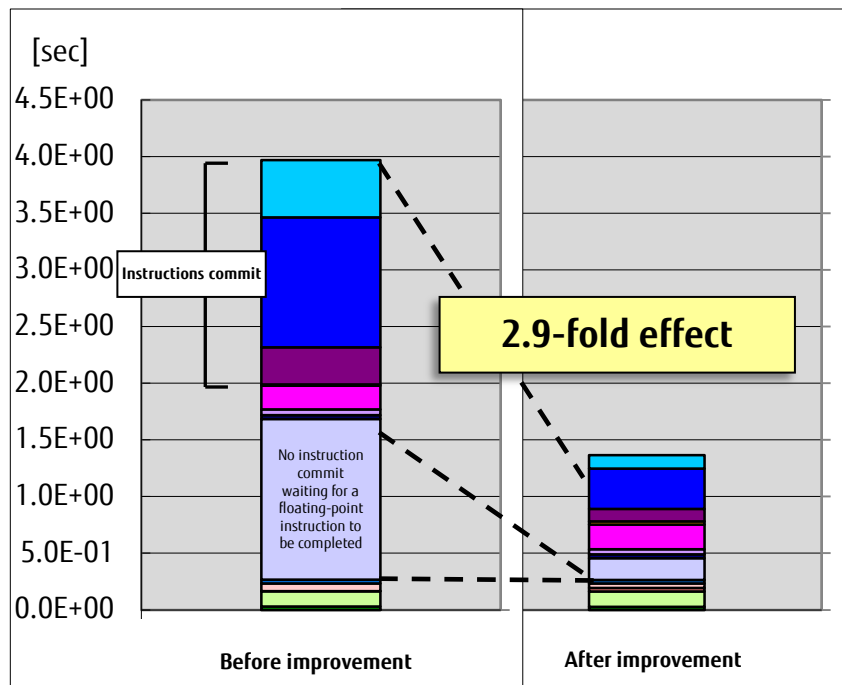
4 1 pp      do j = 1, m
5 1 p        i = 1
6 1 p        a(i,j) = c0 + a(1,j)*(c1 + b(i,j)*(c2 + b(i,j)*(c3 + b(i,j)*
7 1          & (c4 + b(i,j)*(c5 + b(i,j)*(c6 + b(i,j)*(c7 + b(i,j)*
8 1          & (c8 + b(i,j)*c9))))))

<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< SIMD (VL: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End >>>

9 2 p 6v     do i = 2, n
10 2 p 6v     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))))))
13 2 p 6v     end do
14 1 p        end do
15           end
    
```

Peeling of locations that have dependency

The elimination of dependency facilitated SIMD optimization and software pipelining in the period of i = 2 to n.



SIMD optimization reduced effective instruction.

	Effective instruction
Before improvement	1.68E+11
After improvement	4.27E+10

	SIMD instruction rate (effective instruction)	SIMD floating point instruction rate (/SIMD target floating point instruction)	SIMD integer instruction rate (/SIMD target integer instruction)	SIMD load-store instruction rate (/SIMD target load-store instruction)
Before improvement	0.00%	0.00%	0.00%	0.00%
After improvement	81.99%	96.12%	0.00%	83.91%

Loop That Has an Unclear Definition Reference Relationship

- Loop That Has an Unclear Definition Reference Relationship (Before Improvement)
- Loop That Has an Unclear Definition Reference Relationship (Optimization Control Line Tuning)
- Loop That Has an Unclear Definition Reference Relationship (Optimization Control Line)

There is neither SIMD optimization nor effective software pipelining because of unclear data dependency regarding array a. Consequently, the following is a frequent event: No instruction commit waiting for a floating-point instruction to be completed.

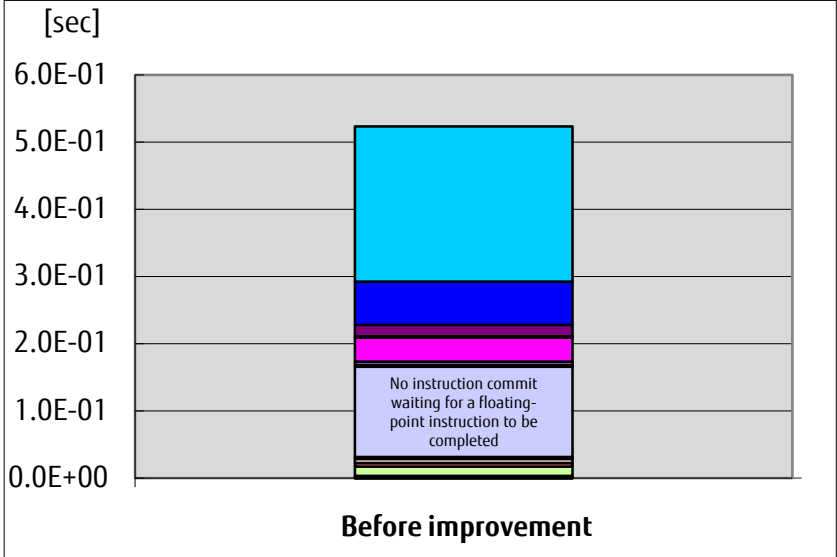
Source code before improvement

<<< Loop-information Start >>>
<<< [PARALLELIZATION]
<<< Standard iteration count: 2
<<< Loop-information End >>>

53 1 pp do j=1,n1
54 2 p 6s do i=1,n2
55 2 p 6m a(l(i),j)=a(x(i),j)/b(i,j)
56 2 p 6v end do
57 1 p end do

There is unclear dependency between array a on the load side and array a on the store side.

There is no effective software pipelining because of unclear data dependency between iteration cycles for array a.



SIMD				
	SIMD instruction rate (effective instruction)	SIMD floating point instruction rate (/SIMD target floating point instruction)	SIMD integer instruction rate (/SIMD target integer instruction)	SIMD load-store instruction rate (/SIMD target load-store instruction)
Before improvement	0.00%	0.00%	0.00%	0.00%

There is no SIMD optimization.

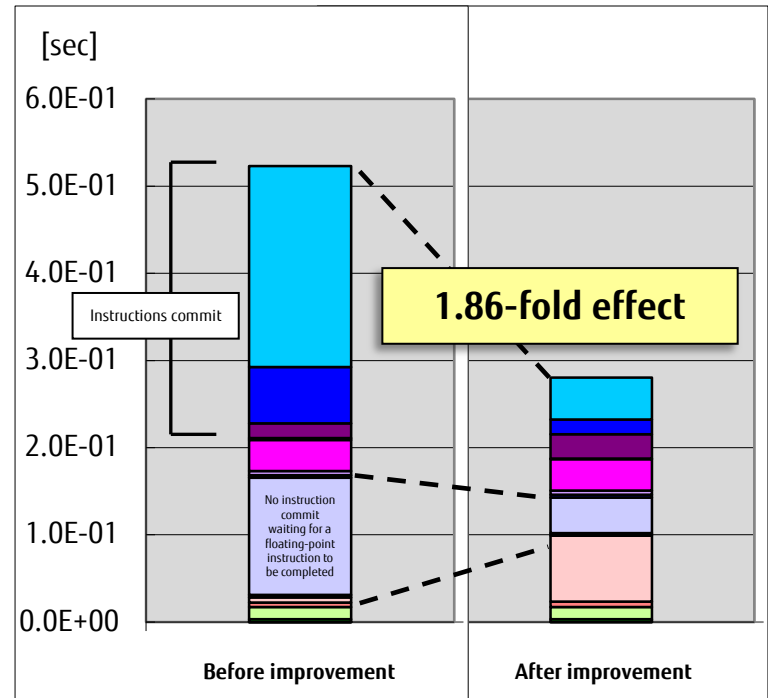
With no data dependency made explicit by **the NORECURRENCE specifier**, SIMD optimization and software pipelining were facilitated. The result is a reduction in the total number of effective instructions, a decrease in instruction commits, facilitation of instruction scheduling, and a significant improvement in the following event: No instruction commit waiting for a floating-point instruction to be completed.

Source code after improvement (optimization control line tuning)

```

53      !ocl norecurrence(a)
      <<< Loop-information Start >>>
      <<< [PARALLELIZATION]
      <<< Standard iteration count: 2
      <<< Loop-information End >>>
54  1 pp      do j=1,n1
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< Loop-information End >>>
55  2 p 6v      do i=1,n2
56  2 p 6v      a(l(i),j)=a(x(i),j)/b(i,j)
57  2 p 6v      end do
58  1 p      end do
    
```

Notifies compiler about no data dependency between $a(l(i), j)$ and $a(x(i), j)$



	Effective instruction
Before improvement	3.87E+10
After improvement	8.71E+09

SIMD optimization reduced effective instruction.

	SIMD instruction rate (effective instruction)	SIMD floating point instruction rate (/SIMD target floating point instruction)	SIMD integer instruction rate (/SIMD target integer instruction)	SIMD load-store instruction rate (/SIMD target load-store instruction)
Before improvement	0.00%	0.00%	0.00%	0.00%
After improvement	88.34%	100.00%	76.92%	97.75%

Here, specify the following optimization control line.

optimization control specifiers	Meaning	Optimization control line that can be specified			
		Program unit	DO loop unit	Statement unit	Array assignment statement unit
NORECURRENCE [(array1[,array2]...)]	Gives an instruction to the main processing system about elements of the array that is the operation target in a DO loop. The instruction is that definitions of the array elements are not to be referenced over different iterations. (Gives an instruction to arrays for which loop slicing is possible.) <i>array1, array2,...</i> are array names.	Yes	Yes	No	Yes

Loop Containing Pointer Variables

- Loop Containing Pointer Variables (Before Improvement)
- Loop Containing Pointer Variables
(Optimization Control Line Tuning)
- Making Data Dependency Explicit Regarding Array Subscripts
(Optimization Control Line)
- Speed-up by CONTIGUOUS attribute

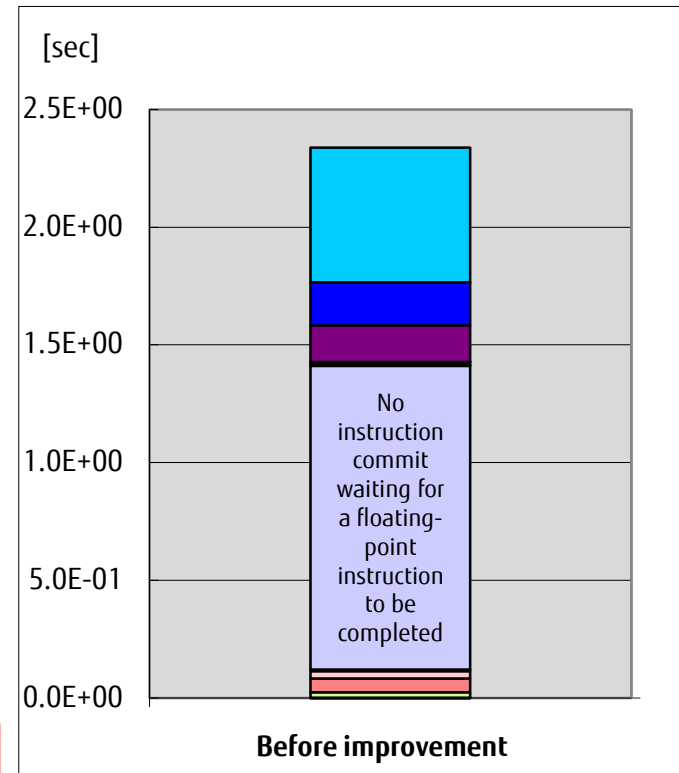
Loop Containing Pointer Variables (Before Improvement)

SIMD optimization and software pipelining are not facilitated because the pointer variables of arrays a and b point to unknown memory areas. Consequently, the following is a frequent event: No instruction commit waiting for a floating-point instruction to be completed.

Source code before improvement	
1	program sub_f
2	
3	real,dimension(100000),target::x
4	integer :: kmax
5	real,dimension(:),pointer::a,b
6	1 pp 8v do i=1,100000
7	1 p 8v x(i)=i
8	1 p 8v end do
9	a=>x(1:10000)
10	b=>x(10001:20000)
11	kmax = 1000000
12	call start_collection("1")
13	!\$omp parallel
14	1 do k=1,kmax
15	1 !\$omp do
16	2 p 8s do i=1,10000
17	2 p 8s a(i)=2.0/b(i)+1.0
18	2 p 8s end do
19	1 end do
20	!\$omp end parallel

Unclear dependency between array b on load side and array a on store side

There is no SIMD optimization.



SIMD

	SIMD instruction rate (effective instruction)	SIMD floating point instruction rate (/SIMD target floating point instruction)	SIMD integer instruction rate (/SIMD target integer instruction)	SIMD load-store instruction rate (/SIMD target load-store instruction)
Before improvement	0.00%	0.00%	0.00%	0.00%

Loop Containing Pointer Variables (Optimization Control Line Tuning)

With data dependency made explicit by **the NOALIAS specifier**, SIMD optimization and software pipelining were facilitated. This results in significant improvement of the following event: No instruction commit waiting for a floating-point instruction to be completed.

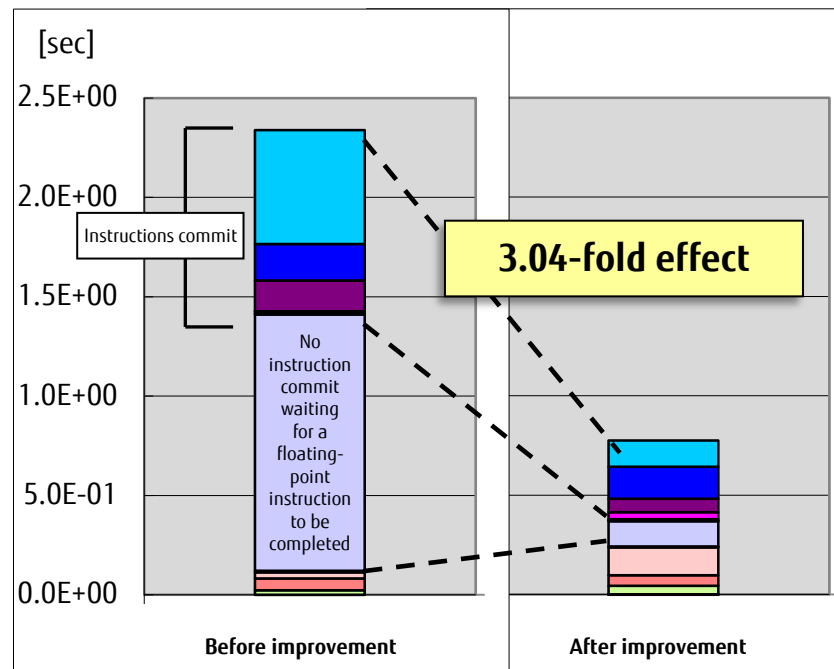
Source code after improvement (optimization control line tuning)

```

1      program sub_f
2
3      real,dimension(100000),target::x
4      integer :: kmax
5      real,dimension(:),pointer::a,b
6  1 pp 8v    do i=1,100000
7  1 p 8v      x(i)=i
8  1 p 8v      end do
9      a=>x(1:10000)
10     b=>x(10001:20000)
11     kmax = 1000000
12     call start_collection("1")
13     !$omp parallel
14  1      do k=1,kmax
15  1      !$omp do
16  1      !ocl noalias
17  2 p 6v      do i=1,10000
18  2 p 6v          a(i)=2.0/b(i)+1.0
19  2 p 6v      end do
20  1      end do

```

Notifies compiler about no data dependency between a(i) and b(i)



SIMD optimization reduced effective instruction.

	Effective instruction
Before improvement	1.05E+11
After improvement	3.35E+10

	SIMD instruction rate (effective instruction)	SIMD floating point instruction rate (/SIMD target floating point instruction)	SIMD integer instruction rate (/SIMD target integer instruction)	SIMD load-store instruction rate (/SIMD target load-store instruction)
Before improvement	0.00%	0.00%	0.00%	0.00%
After improvement	75.36%	99.36%	41.78%	91.78%

Here, specify the following optimization control line.

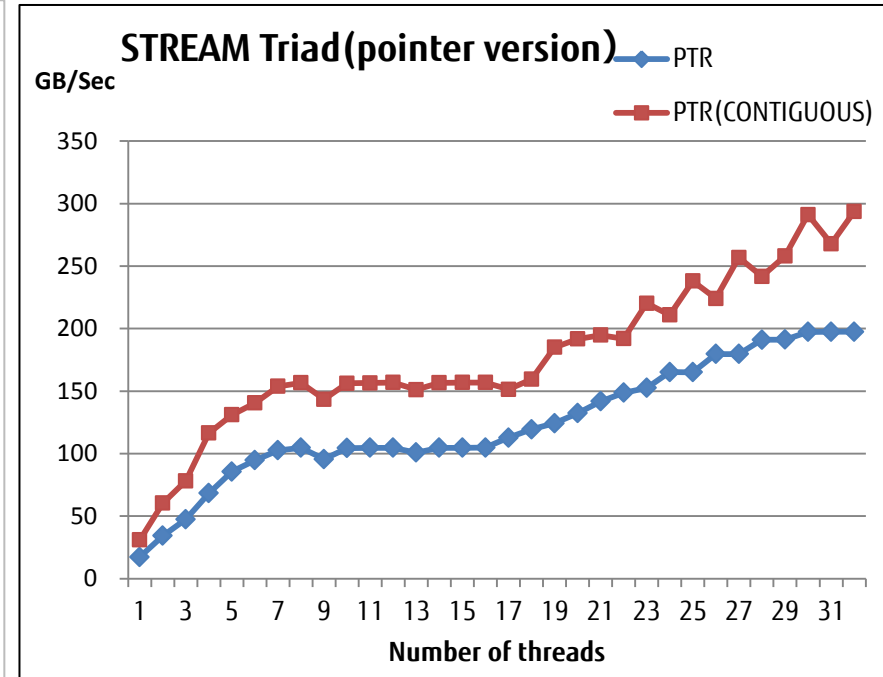
Optimization control specifiers	Meaning	Optimization control line that can be specified			
		Program unit	DO loop unit	Statement unit	Array assignment statement unit
NOALIAS	Gives an instruction that pointer variables are not to share memory areas with other variables.	Yes	Yes	No	No

Speed-up by CONTIGUOUS attribute

Optimization against arrays with pointer attribute may be promoted if CONTIGUOUS attribute be added to them.

STREAM Triad(pointer version)

```
-----
127      DOUBLE PRECISION, dimension(:), pointer,contiguous :: a,b,c
248  1   f  !$OMP PARALLEL DO
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< SOFTWARE PIPELINING
      <<< PREFETCH   : 10
      <<< c: 4, b: 4, a: 2
      <<< XFILL      : 2
      <<< a: 2
      <<< Loop-information End >>>
249  2   p   fv      DO 60 j = 1,n
250  2   p   fv      a(j) = b(j) + scalar*c(j)
251  2   p   fv      60  CONTINUE
```



In above case, XFILL optimization has been applied by specifying CONTIGUOUS attribute, then performance improved.

CONTIGUOUS attribute(Specification introduced by Fortran2008)

CONTIGUOUS attribute can be specified to shape-assumed arrays or pointer arrays,

- in case of shape-assumed arrays, they should associate with actual arguments having CONTIGUOUS attribute.
- in case of pointer arrays, they should associate with targets with CONTIGUOUS attribute.

Hindering Factor: Improvement of a Loop with a Few Iterations

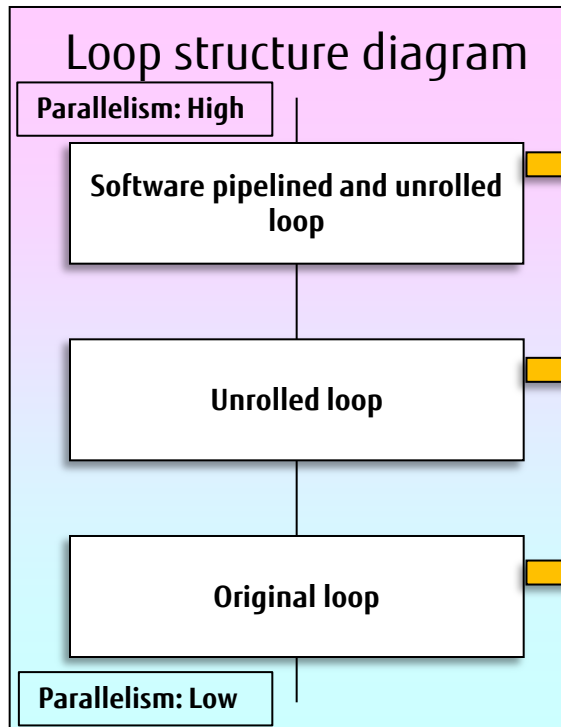
- Loop with a Few Iterations
- Specification of an Appropriate Number of Unrollings and Suppression of Software Pipelining
- Cloning
 - Before Improvement by Cloning
 - Cloning Source Tuning

Loop with a Few Iterations

Even for a loop for which software pipelining was facilitated, if the number of loop iterations is small, instruction scheduling may have a small effect.

Example `do i = 1, n
 b(i) = a(i) + c
 enddo`

Assumptions in this example
Loop iteration count, $n = 40$
Number of unrollings = 8
Software pipelining
4SIMD



if (process whose iteration count is value (160 or more) presented by software pipelining) then

jwd8205o-i "sample.f90", line 4: At a loop iteration count of 160 or more, a loop to which software pipelining is applied is selected at the execution time.

Since the loop iteration count in this example is 40, this loop is not executed.

else if (process whose iteration count is multiple of 32 iterations, among those remaining with above iteration count) then

8 unrollings x 4 (SIMD) = 32

For the loop in this example, the processes for $i = 1$ to 32 are executed in this loop.

else

For the loop in this example, the remaining processes for $i = 33$ to 40 are executed in this loop.

The above diagram shows the multiple loop structure created by the compiler. In the structure, the higher the layer, the higher the parallelism at the instruction level.

As described above, if the loop iteration count is small, instruction scheduling has a small effect since loops with high parallelism at the instruction level are not executed.

Specification of an Appropriate Number of Unrollings and Suppression of Software Pipelining

- Specification of an Appropriate Number of Unrollings and Suppression of Software Pipelining (Before Improvement)
- Specification of an Appropriate Number of Unrollings and Suppression of Software Pipelining (After Improvement)
- Specification of an Appropriate Number of Unrollings and Suppression of Software Pipelining (Optimization Control Line)
- Specification of an Appropriate Number of Unrollings and Suppression of Software Pipelining (Compiler Options)

Unrolling and software pipelining do not function effectively because the number of iterations is small. Consequently, the following is a frequent event: No instruction commit waiting for a floating-point instruction to be completed.

Source code before improvement

```
<<< Loop-information Start >>>
```

```
<<< [PARALLELIZATION]
```

```
<<< Standard iteration count: 534
```

```
<<< [OPTIMIZATION]
```

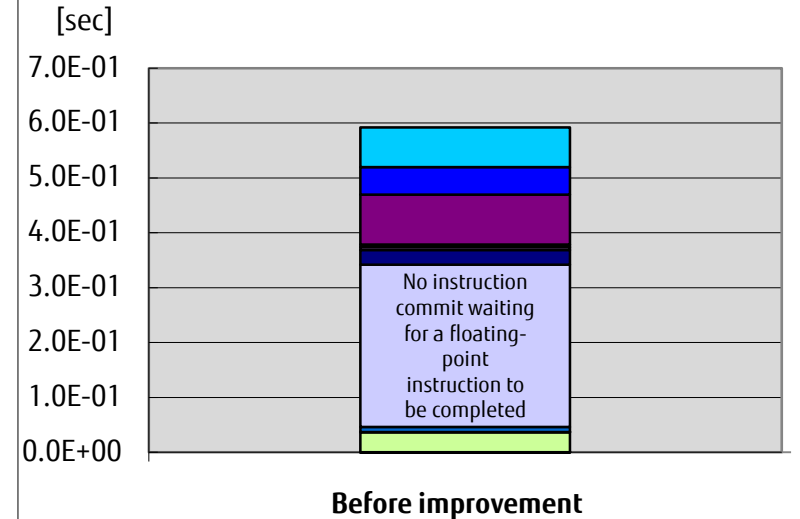
```
<<< SIMD(VL: 4)
```

```
<<< SOFTWARE PIPELINING
```

```
<<< Loop-information End >>>
```

Number of unrollings is 6,
with loop iterations count of
n = 16

```
42 1 pp 6v      do i = 1, n
43 1 p 6v        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 6v      enddo
47             End
```



Problem: The loop iteration count is small.

- Software pipelining loops are not executed.

Although "SOFTWARE PIPELINING" is displayed as optimization information, software pipelining loops are not executed because the number of iterations is small.

jwd82050-i "sample.f", line 42: At a loop iteration count of 120 or more, a loop to which software pipelining is applied is selected at the execution time.

- The number of unrollings is not appropriate.

6 unrollings x 4 (SIMD) = 24 The unrolled loop is not even executed once.

The original loop is executed for all 16 iterations.

The execution of the original loop has a significant effect on performance.

Appropriate instruction scheduling was done with a specified number of unrollings appropriate to the number of iterations and with software pipelining suppressed. This resulted in the following event being significantly reduced: No instruction commit waiting for a floating-point instruction to be completed.

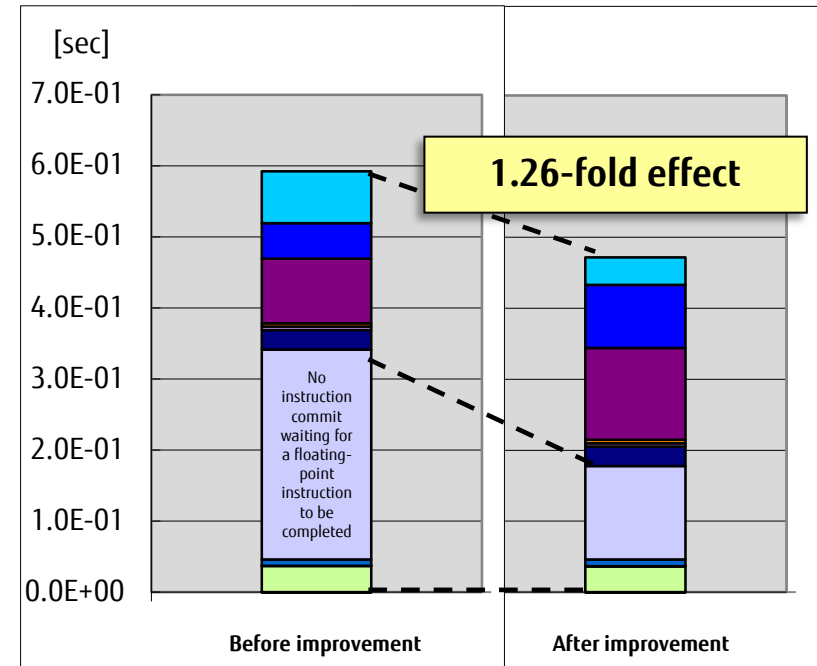
Source code after improvement (optimization control line tuning)

```

42      !ocl unroll(4)
43      !ocl noswp
      <<< Loop-information Start >>>
      <<< [PARALLELIZATION]
      <<< Standard iteration count: 534
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< Loop-information End >>>
44  1 pp 4v      do i = 1, n
45  1 p 4v      b(i) = c0 + a(i)*(c1 + a(i)*(c2 + a(i)*(c3 + a(i)*
46  1          & (c4 + a(i)*(c5 + a(i)*(c6 + a(i)*(c7 + a(i)*
47  1          & (c8 + a(i)*c9))))))
48  1 p 4v      enddo
49      End
  
```

Specifies 4 as appropriate number of unrollings

Suppresses software pipelining



From the specification of 4 for the number of unrollings
 4 unrollings x 4 (SIMD) = 16
 The unrolled loop is executed for all 16 iterations.

Here, specify the following optimization control line.

Optimization specifier	Meaning	Optimization control line that can be specified			
		Program unit	DO loop unit	Statement unit	Array assignment statement unit
UNROLL(<i>m1</i>)	Unrolls a DO loop. <i>m1</i> is a decimal number in a range of 2 to 100 that indicates the number of unrollings (multiplicity).	No	Yes	No	No
NOSWP	Disables the software pipelining function.	Yes	Yes	No	Yes

You can achieve effects similar to source tuning by specifying the following compiler options.

Compiler options	Description of function
-Kunroll[=<i>N</i>]	Gives an instruction to optimize loop unrolling. Specify an upper limit in <i>N</i> for the number of loop unrollings. You can specify a value in a range of 2 to 100 for <i>N</i> . If the specification of <i>N</i> is omitted, the compiler automatically determines the optimal value. If the -00 or -01 option is valid, the default is -Knounroll. If -02 or a higher option is valid, the default is -Kunroll.
-Knoswp	Gives an instruction not to optimize software pipelining.

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

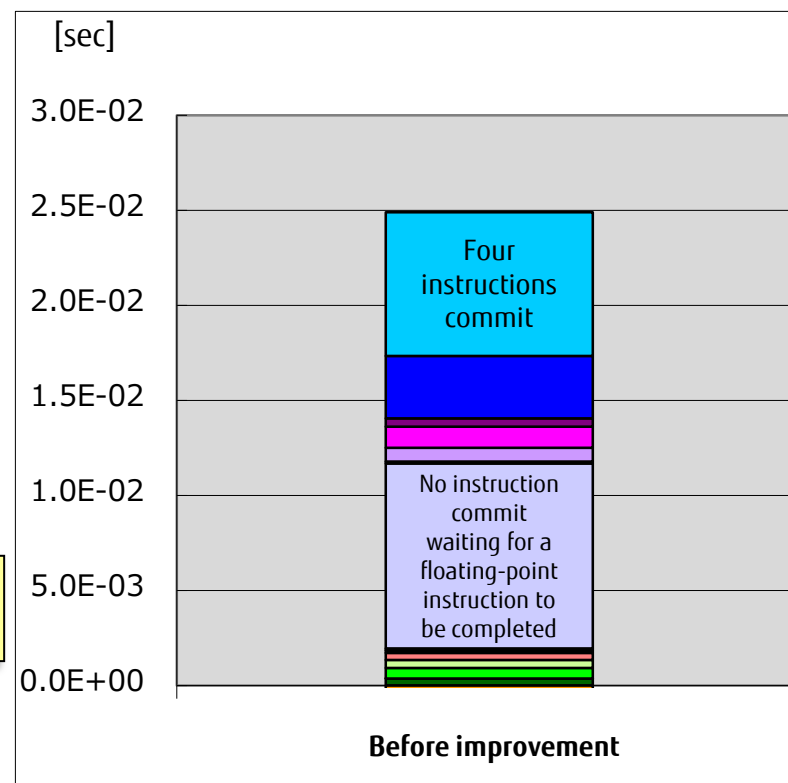
\$ frtpx -Kfast,parallel sample.f90 -Kunroll=8,noswp

Before Improvement by Cloning

Even though the number of iterations of the innermost loop depends disproportionately on a specific condition, it is just a variable. For this reason, optimizations such as full unrolling are hindered, the number of instructions increases, and the following is a frequent event: No instruction commit waiting for a floating-point instruction to be completed.

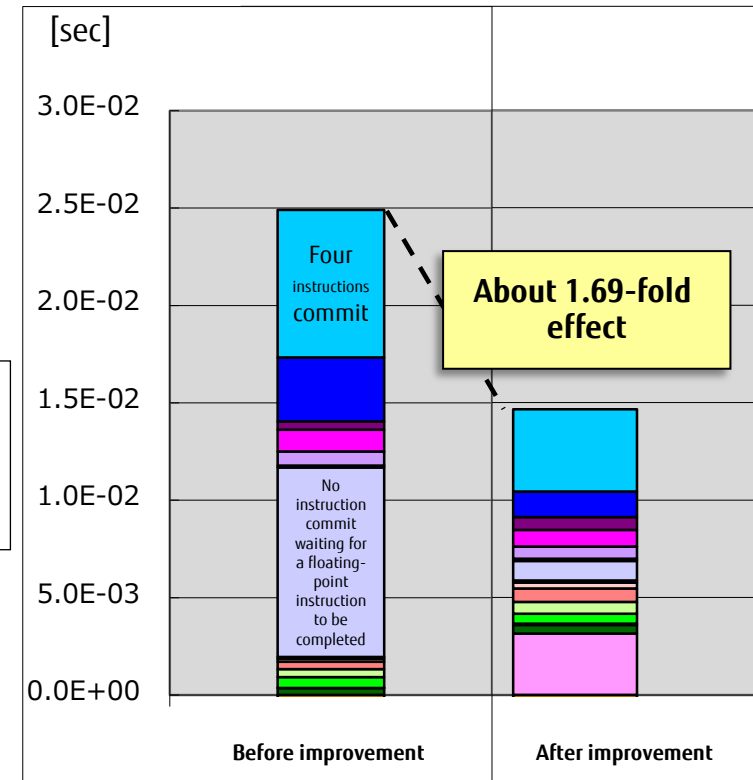
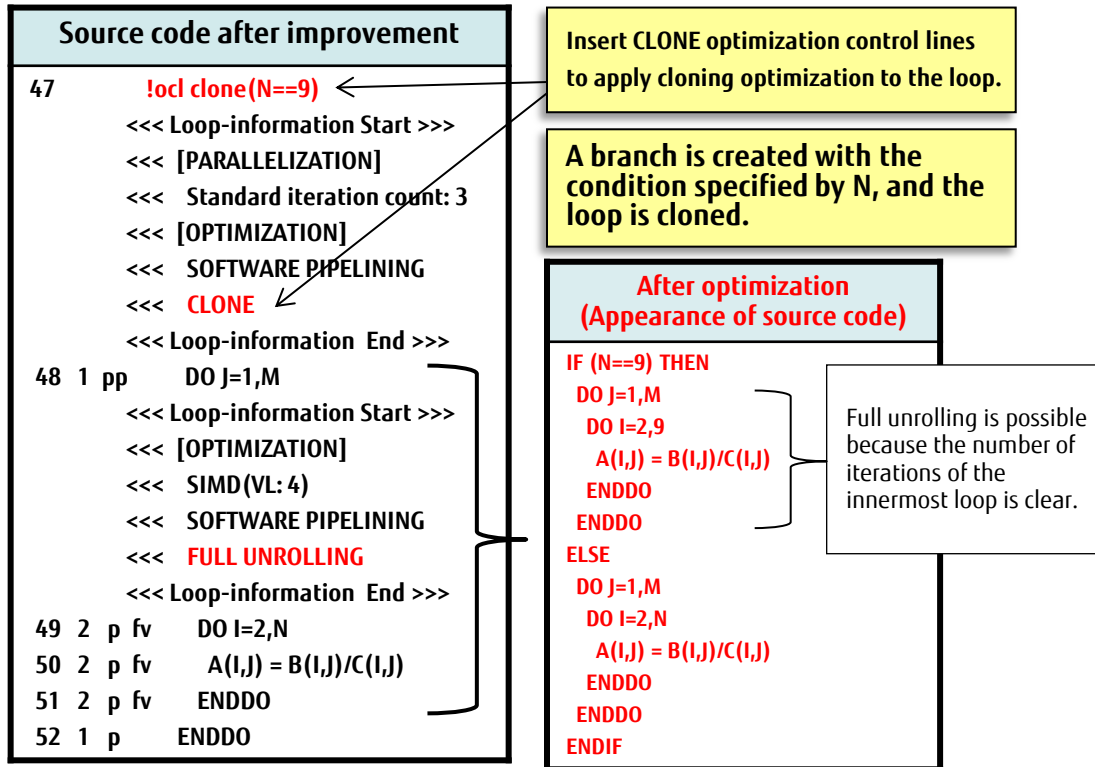
Source code before improvement			
<<< Loop-information Start >>>			
<<< [PARALLELIZATION]			
<<< Standard iteration count: 3			
<<< Loop-information End >>>			
48	1	pp	DO J=1,M
<<< Loop-information Start >>>			
<<< [OPTIMIZATION]			
<<< SIMD(VL: 4)			
<<< SOFTWARE PIPELINING			
<<< Loop-information End >>>			
49	2	p 6v	DO I=2,N ←
50	2	p 6v	A(I,J) = B(I,J)/C(I,J)
51	2	p 6v	ENDDO
52	1	p	ENDDO

Cases where N is 9 account for 100%.



Cloning Source Tuning

Clone optimization control lines are specified to create a conditional branch in the innermost loop using variable values and to facilitate full unrolling and other optimizations. This results in significant improvement of the following event: No instruction commit waiting for a floating-point instruction to be completed.



	Effective instruction
Before improvement	1.34E+08
After improvement	6.71E+07

Facilitating optimization reduced effective instruction.

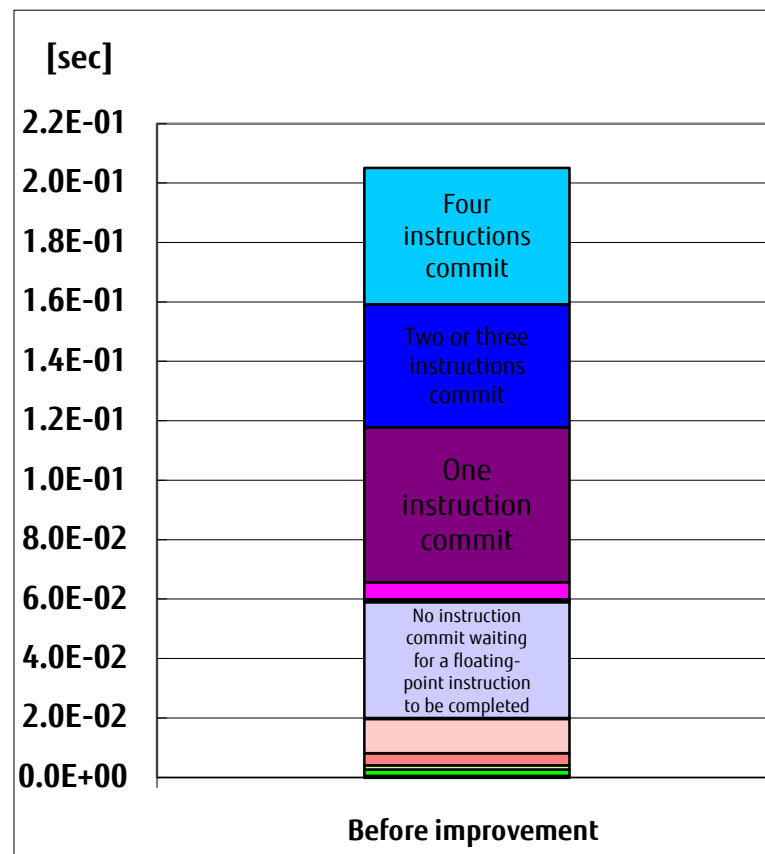
Various Optimizations

- Rerolling
 - Rerolling (Before Improvement)
 - Effects of Rerolling (Source Tuning)
- Facilitation of SIMD Optimization through Changes to Simple Variables
- Inline expansion: procedures using associated allocatable variable

Rerolling (Before Improvement)

There are many instruction commits because SIMD optimization is unavailable due to the dependency in the innermost loop.

Source code after improvement (source tuning)	
<<< Loop-information Start >>>	
<<< [PARALLELIZATION]	
<<< Standard iteration count: 2	
<<< Loop-information End >>>	
42 1 pp	DO K=1,M
<<< Loop-information Start >>>	
<<< [OPTIMIZATION]	
<<< SOFTWARE PIPELINING	
<<< Loop-information End >>>	
43 2 p 4s	DO J=1,N
44 2 p 4s	a(1,J,K) = b(1,J,K) + a(1,J-1,K)
45 2 p 4s	a(2,J,K) = b(2,J,K) + a(2,J-1,K)
46 2 p 4s	a(3,J,K) = b(3,J,K) + a(3,J-1,K)
47 2 p 4s	a(4,J,K) = b(4,J,K) + a(4,J-1,K)
48 2 p 4s	ENDDO
49 1 p	ENDDO
50	end subroutine rerolling



SIMD

	SIMD instruction rate (effective instruction)	SIMD floating point instruction rate (/SIMD target floating point instruction)	SIMD integer instruction rate (/SIMD target integer instruction)	SIMD load-store instruction rate (/SIMD target load-store instruction)
Before improvement	0.00%	0.00%	0.00%	0.00%

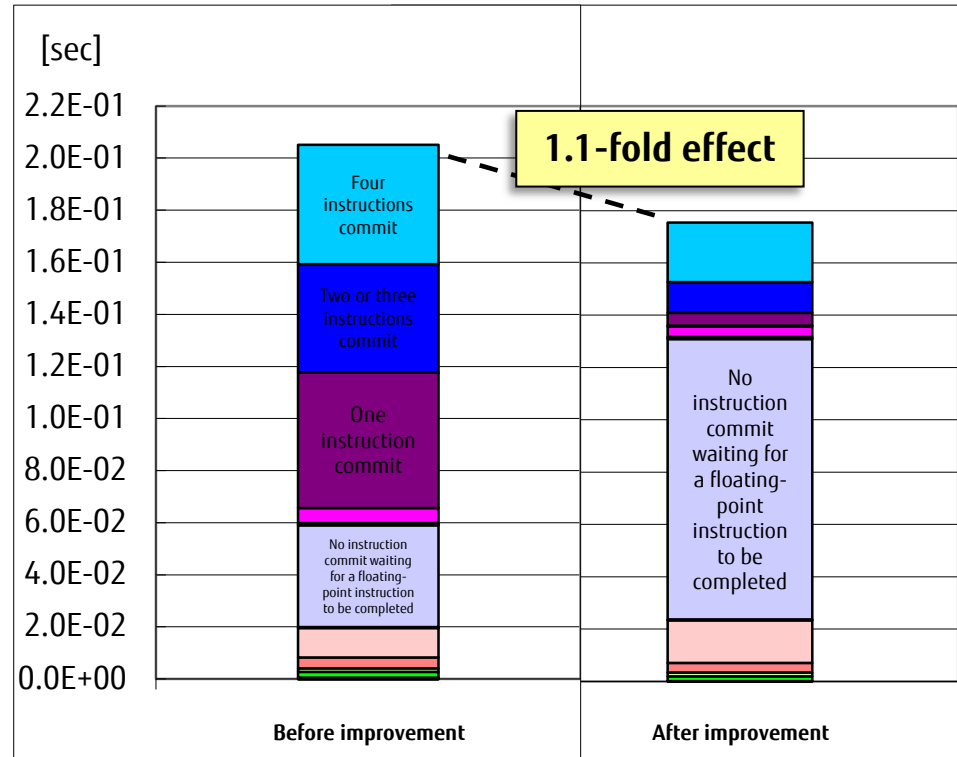
There is no SIMD optimization.

Effects of Rerolling (Source Tuning)

Rewriting unrolled statements into loop statements (returning them to a loop) facilitates SIMD optimization. This results in a decreased effective instruction and improved performance.

Source code after improvement	
<<< Loop-information Start >>>	
<<< [PARALLELIZATION]	
<<< Standard iteration count: 2	
<<< Loop-information End >>>	
42 1 pp	DO K=1,M
43 1	!OCL NOUNROLL
<<< Loop-information Start >>>	
<<< [OPTIMIZATION]	
<<< SOFTWARE PIPELINING	
<<< Loop-information End >>>	
44 2 p	DO J=1,N
<<< Loop-information Start >>>	
<<< [OPTIMIZATION]	
<<< SIMD(VL: 4)	
<<< Loop-information End >>>	
45 3 p v	DO I=1,4
46 3 p v	a(I,J,K) = b(I,J,K) + a(I,J-1,K)
47 3 p v	ENDDO
48 2 p	ENDDO
49 1 p	ENDDO
50	end subroutine rerolling

Suppresses loop unrolling



	Effective instruction
Before improvement	1.17E+10
After improvement	4.43E+09

SIMD optimization reduced effective instruction.

SIMD

	SIMD instruction rate (effective instruction)	SIMD floating point instruction rate (/SIMD target floating point instruction)	SIMD integer instruction rate (/SIMD target integer instruction)	SIMD load-store instruction rate (/SIMD target load-store instruction)
Before improvement	0.00%	0.00%	0.00%	0.00%
After improvement	55.34%	100.00%	0.00%	96.68%

Changing arrays with constant subscripts to simple variables may facilitate SIMD optimization.

Example: Program change to simple variables

Before correction

```
do i = 1 , N
  :
  a(1) = b(1,i)
  a(2) = b(2,i)
  a(3) = b(3,i)
  a(4) = b(4,i)
  a(5) = b(5,i)
  a(6) = b(6,i)
  :
  x = u(2) * a(1) + u(3) * a(2)
  y = u(2) * a(3) + u(3) * a(4)
  z = u(2) * a(4) + u(3) * a(6)
  :
end do
```

After correction

```
do i = 1 , N
  :
  a1 = b(1,i)
  a2 = b(2,i)
  a3 = b(3,i)
  a4 = b(4,i)
  a5 = b(5,i)
  a6 = b(6,i)
  :
  x = u(2) * a1 + u(3) * a2
  y = u(2) * a3 + u(3) * a4
  z = u(2) * a4 + u(3) * a6
  :
end do
```

Since arrays a(1) to a(6) are handled practically as local variables, they are changed to simple variables a1 to a6.

- **Procedure containing use-association of allocatable variable are not target of inline expansion. If allocatable variable is not referred to, copying the module and modifying to delete the use-association make the procedure will be target of inline expansion.**

```
module mod2
  real(8), dimension(:), allocatable :: ql
end module mod2
```

```
module mod1
  use mod2
  real(8) :: a(100)
  contains
    subroutine sub1(n)
      do i=1,n
        a(i) = 5
      end do
    end subroutine sub1
```

Allocatable
variable ql is
not referred to
in sub1

```
    subroutine sub3(n)
      do i=1,n
        ql(i) = 5
      end do
    end subroutine sub3
  end module mod1
```

Inline
expansion of
sub1 is preferred

```
  subroutine sub2()
    use mod1
    call sub1(100)
  end subroutine sub2
```

jwd2483i-i "b.f90", line 25, column 8: Module procedure 'sub1' not expanded inline, because it use associates an allocatable variable.

```
module mod2
  real(8), dimension(:), allocatable :: ql
end module mod2
```

```
module mod1
  use mod2
  subroutine sub3(n)
    do i=1,n
      ql(i) = 5
    end do
  end subroutine sub3
end module mod1
```

Sub1 is copied and
modified as
another procedure
in module mod11

```
module mod11
  real(8) :: a(100)
  contains
    subroutine sub1(n)
      do i=1,n
        a(i) = 5
      end do
    end subroutine sub1
  end module mod11
```

Copied and
modified module
mod11 is used

```
subroutine sub2()
  use mod11
  call sub1(100)
end subroutine sub2
```

jwd8101o-i "c.f90", line 28: 'sub1' is expanded inline.

Thread Parallelization Processing Tuning

- Thread Parallelization ratio Improvement
- Execution Efficiency Improvement of Thread Parallelization Processing

Thread Parallelization Ratio Improvement

- What Is the Thread Parallelization Ratio?
- Thread Parallelization Ratio Improvement

What Is the Thread Parallelization Ratio?

The parallelization ratio means the percentage that can be executed in parallel in one parallel execution sequence.

One parallel execution sequence

Sequential execution part

Part that can be executed in parallel

Two parallel execution sequences

Sequential execution part

Part that can be executed in parallel

In two parallel execution sequences, the part that can be executed in parallel becomes 1/2 of one parallel execution sequence.

Amdahl's law shows the relationship between the thread parallelization ratio and scalability in n parallel execution sequences.

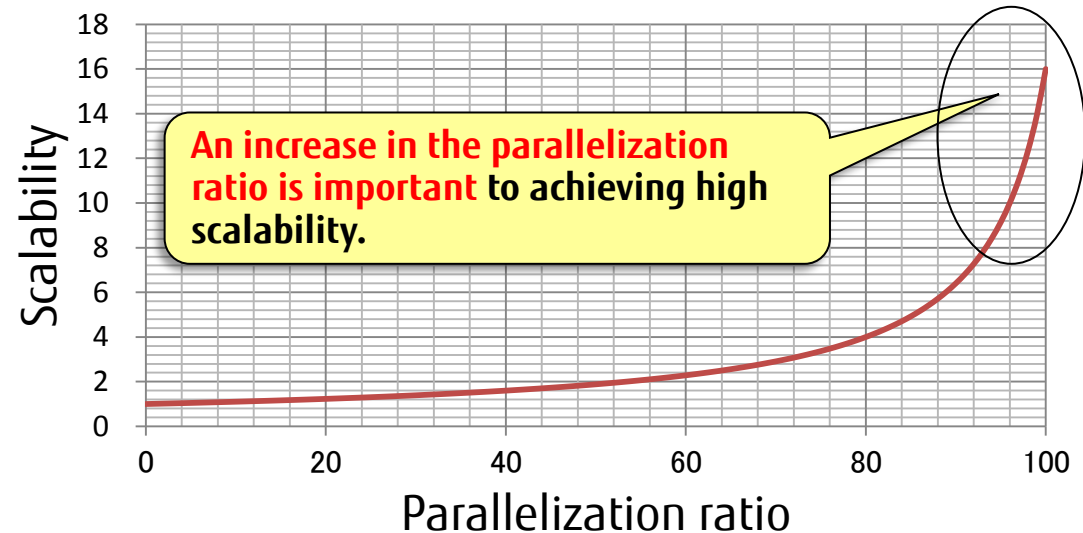
■ Amdahl's law

$$\text{Scalability} = \frac{1}{(1 - p) + \frac{p}{n}}$$

■ p : Parallelization ratio

■ n : Number of parallels

$n = 16$ (ideally, 16 parallel processes)



Thread Parallelization Ratio Improvement

- Loop That Has an Unclear Definition Reference Relationship
- Loop Containing Pointer Variables
- Loop That Has Data Dependency

■ !OCL NORECURRENCE

The main processing system cannot determine whether applying loop slicing to array a will cause a problem in the following problem, because the subscript expression of array a is an element of another array, y(j). If the programmer knows that loop slicing of array a will not cause a problem, the programmer can use parallelization by specifying **the NORECURRENCE specifier**.

Source code before improvement	Source code after improvement
<pre> 5 6 1 s 8s do i=1,1000 7 1 m 8m a(y(i))=a(y(i))+b(i) 8 1 p 8v end do 9 END </pre> <p>jwd5228p-i "a.f90", line 7: This DO loop cannot be parallelized because the definition reference order of data differs from the order of sequential execution.</p> <p>jwd6228s-i "a.f90", line 7: This DO loop cannot be SIMD-optimized because the definition reference order of data differs from the order of sequential execution.</p>	<pre> 5 !ocl norecurrence(a) <<< Loop-information Start >>> <<< [PARALLELIZATION] <<< Standard iteration count: 800 <<< [OPTIMIZATION] <<< SIMD(VL: 4) <<< Loop-information End >>> 6 1 pp 8v do i=1,1000 7 1 p 8v a(y(i))=a(y(i))+b(i) 8 1 p 8v end do </pre>

!Note!

- If the NORECURRENCE specifier is specified for an array for which loop slicing is not possible, the main processing system may apply the wrong loop slicing.
- If the array name is omitted, the specification is valid for all arrays in the target section.

■ !OCL NOALIAS

Data dependency is unclear and there is no parallelization because the memory areas occupied by the pointer variables are determined at the execution time. If the programmer knows that the pointer variables do not point to the same memory area, the programmer can use parallelization by specifying the **NOALIAS** specifier.

Source code before improvement		Source code after improvement	
1	real,dimension(100000),target::x	1	real,dimension(100000),target::x
2	real,dimension(:),pointer::a,b	2	real,dimension(:),pointer::a,b
3	a=>x(1:10000)	3	a=>x(1:10000)
4	b=>x(10001:20000)	4	b=>x(10001:20000)
5		5	!ocl noalias
6	1 s s do i=1,100000		<<< Loop-information Start >>>
7	1 s s b(i) = a(i)+1.0		<<< [PARALLELIZATION]
8	1 s s end do		<<< Standard iteration count: 1143
9	end		<<< [OPTIMIZATION]
			<<< SIMD(VL: 8)
			<<< SOFTWARE PIPELINING
			<<< Loop-information End >>>
		6	1 pp 8v do i=1,100000
		7	1 p 8v b(i) = a(i)+1.0
		8	1 p 8v end do
		9	end

jwd5228p-i "a.f90", line 7: This DO loop cannot be parallelized because the definition reference order of data differs from the order of sequential execution.

jwd6228s-i "a.f90", line 7: This DO loop cannot be SIMD-optimized because the definition reference order of data differs from the order of sequential execution.

■ Parallelization through peeling

The following loop is not parallelized because it has dependency regarding array *a* when *i* = 1 and when *i* = *n*. To facilitate parallelization, eliminate the dependency by placing the beginning and end parts of the loop outside the loop.

Source code before improvement	Source code after improvement
<pre>4 1 s 8s do i=1,n 5 1 s 8m a(i)=a(1)+b(i)+a(n) 6 1 s 8v end do</pre> <p>jwd5202p-i "a.f90", line 5: This DO loop cannot be parallelized because the definition reference order of data differs from the order of sequential execution. (Name: a)</p> <p>jwd5208p-i "a.f90", line 5: The definition reference order is unknown and the reference order may differ from the order of sequential execution, so this DO loop cannot be parallelized. (Name: a)</p>	<pre>4 a(1)=a(1)+b(1)+a(n) <<< Loop-information Start >>> <<< [PARALLELIZATION] <<< Standard iteration count: 800 <<< [OPTIMIZATION] <<< SIMD(VL: 8) <<< SOFTWARE PIPELINING <<< Loop-information End >>> 5 1 pp 8v do i=2,n-1 6 1 p 8v a(i)=a(1)+b(i)+a(n) 7 1 p 8v enddo 8 a(n)=a(1)+b(n)+a(n)</pre>

Execution Efficiency Improvement of Thread Parallelization Processing

- Improvement in False Sharing
- Improvement in Load Imbalance

Improvement in False Sharing

- What Is False Sharing?
- False Sharing (Before Improvement)
- False Sharing (Source Tuning)

What Is False Sharing?

False sharing is a phenomenon in which cache lines between threads are frequently invalidated or copied back.

Example assuming four-thread parallelization processing

Source code example

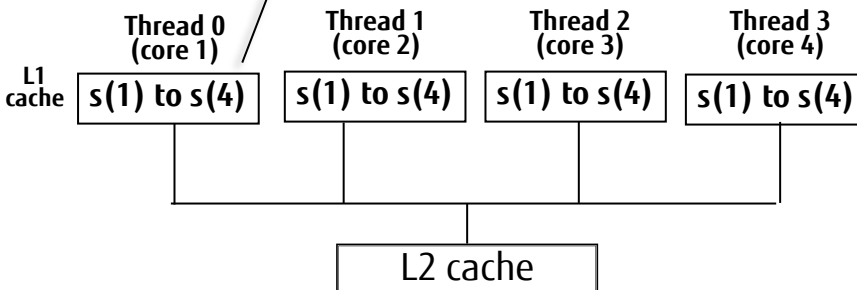
```
1      subroutine sub(s,a,b,ni,nj)
2      real*8 a(ni,nj),b(ni,nj)
3      real*8 s(nj)
4
5  1 pp      do j = 1, nj
6  1 p        s(j)=0.0
7  2 p 8v      do i = 1, ni
8  2 p 8v        s(j)=s(j)+a(i,j)*b(i,j)
9  2 p 8v      end do
10 1 p        end do
11
12          end
```

nj=4
ni=2000

Initial state

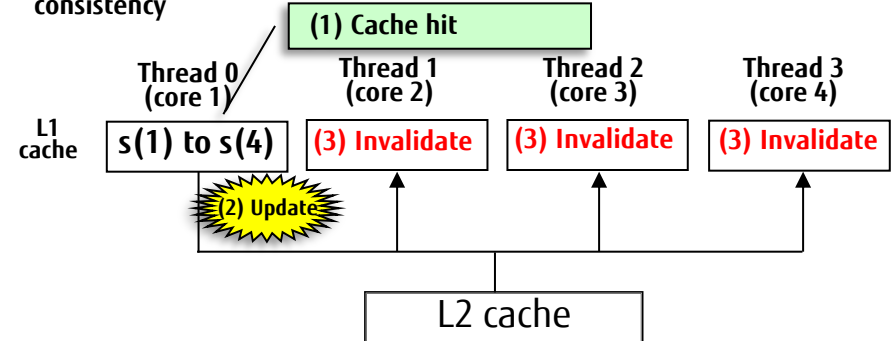
Data is placed in cache in units of cache lines.

Each thread reads the same cache lines, including s(1) to s(4).



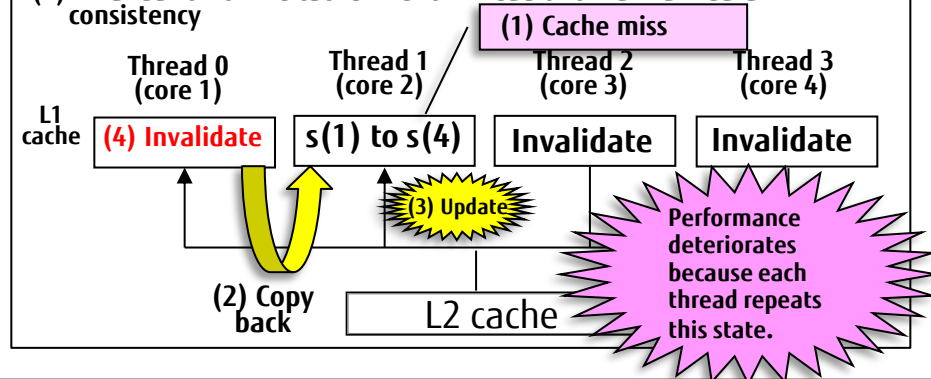
Thread 0 instruction to update s(1)

- (1) Cache hit
- (2) Thread 0 completion of the update of s(1)
- (3) Invalidation of the cache lines of threads 1 to 3 to maintain data consistency



Thread 1 instruction to update s(2)

- (1) Cache miss
- (2) Copy back of the cache line from thread 0 to thread 1
- (3) Thread 1 completion of the update of s(2)
- (4) Invalidation of the cache line for thread 0 to maintain data consistency



False Sharing (Before Improvement)

False sharing occurs because the number of iterations of j, which is the parallelized dimension, is small at 16 and data in array a shares cache lines between threads. Consequently, data access wait is a frequent event.

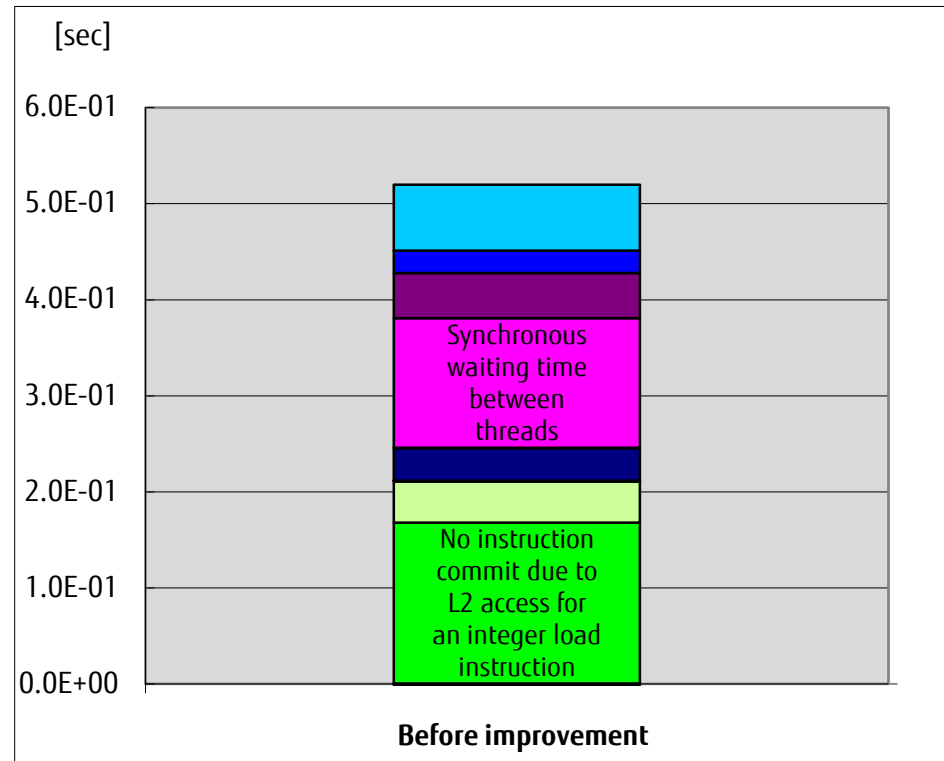
Source code before improvement

```

22      subroutine sub(flag)
23      integer*8 ij,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
30      !$omp parallel
31      !$omp do
32  1 p      do i=1,m
33      <<< Loop-information Start >>>
34      <<< [OPTIMIZATION]
35      <<< PREFETCH :16
36      <<< b:16
37      <<< Loop-information End >>>
38  2 p 8s      do j=1,n
39  3 p 8m      if(flag(i,j).eq.1)then
40  3 p 8s      a(i,j)=b(j,i)
41  3 p 8v      endif
42  2 p 8v      enddo
43  1 p      enddo
44      !$omp end parallel
45
46      end
    
```

Number of iterations of parallelized dimension: 16

False sharing occurrence



Cache

	L1I miss rate (effective instruction)	L1D miss rate(/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	24.98%	7.21E+08	0.21%	99.79%	0.00%	0.00%	1.05E+04	355.62	0.01

The percentage of L1D misses is high, and false sharing has occurred.

False Sharing (Source Tuning)

False sharing can be avoided through loop interchange and parallelization outside the loop. This results in a decrease in the number of L1 cache misses and an improvement in data access wait.

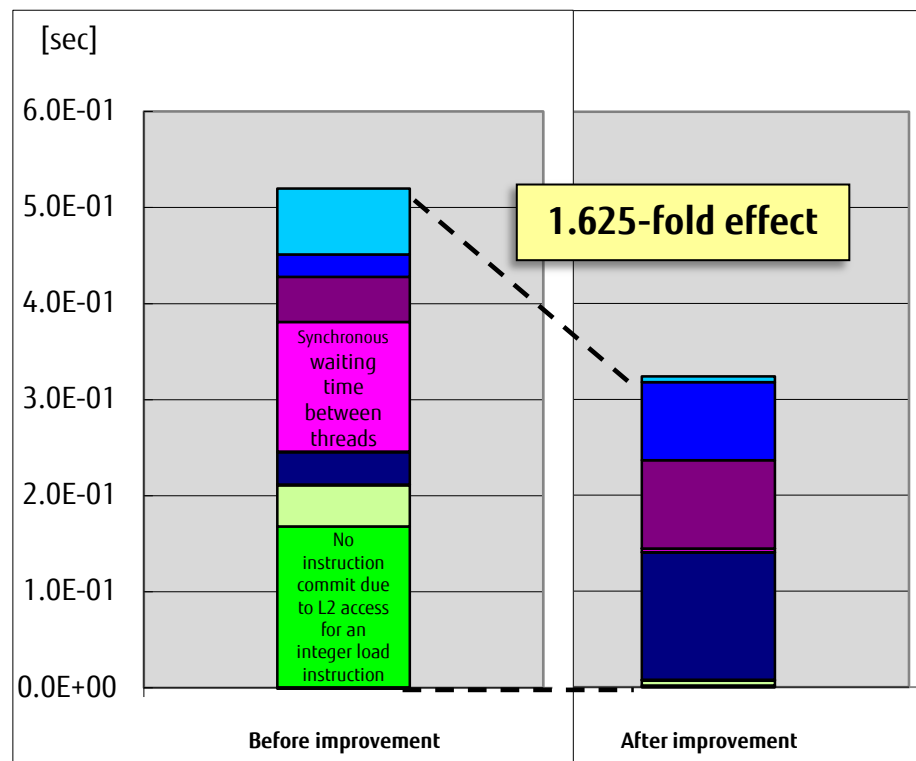
Source code after improvement (source tuning)

```

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
30      !$omp parallel
31      !$omp do
32  1 p      do j=1,n
33          <<< Loop-information Start >>>
34          <<< [OPTIMIZATION]
35          <<< FULL UNROLLING
36          <<< Loop-information End >>>
37  2 p fs      do i=1,m
38  3 p fm          if(flag(i,j).eq.1)then
39  3 p fs              a(i,j)=b(j,i)
40  3 p fv          endif
41  2 p fv      enddo
42  1 p      enddo
43      !$omp end parallel
44
45      end
    
```

Loop interchange and parallelization outside loop

Avoids false sharing



Cache

	L1I miss rate (effective instruction)	L1D miss rate(/Load-store instruction)	L1D miss	L1D miss dm rate(/L1D miss)	L1D miss hwpf rate(/L1D miss)	L1D miss swpf rate(/L1D miss)	L2 miss rate(/Load-store instruction)	L2 miss	L2 throughput (GB/sec)	Memory throughput (GB/sec)
Before improvement	0.00%	24.98%	7.21E+08	0.21%	99.79%	0.00%	0.00%	1.05E+04	355.62	0.01
After improvement	0.00%	1.59%	4.59E+07	2.65%	97.35%	0.00%	0.00%	1.09E+04	36.28	0.02

Avoiding false sharing reduced the L1D miss and increased performance.

Improvement in Load Imbalance

- Triangular Loop
- Loops with Irregular Amount of Calculation
- Small Loop Iteration Count of a Parallelized Dimension

Triangular Loop

- What Is a Triangular Loop?
- Triangular Loop (Before Improvement)
- Triangular Loop (OpenMP Tuning)

What Is a Triangular Loop?

A triangular loop is a loop in which the initial value and end value of an inner loop are determined by the control variable of an outer loop. If that loop is divided into blocks that are executed in parallel, **a load imbalance** occurs.

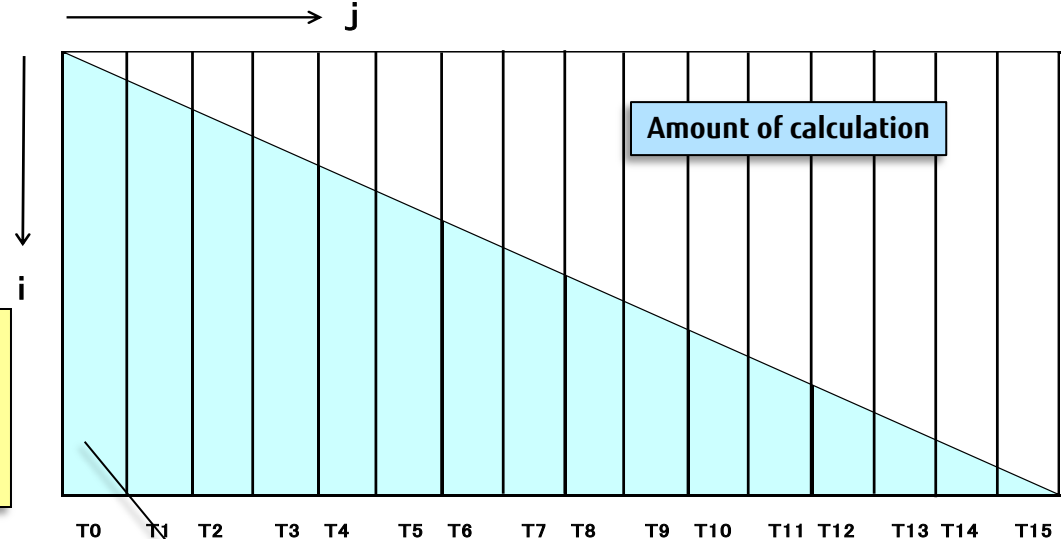
Source code example

```
subroutine sub()
  integer*8 i,j,n
  parameter(n=512)
  real*8 a(n+1,n),b(n+1,n),c(n+1,n)
  common a,b,c

  !$omp parallel do
    do j=1,n
      do i=j,n
        a(i,j)=b(i,j)+c(i,j)
      enddo
    enddo
  end
```

Initial value of inner loop determined by control variable of outer loop

*** Load imbalance:**
This is a phenomenon in which the parallel processing load varies between individual threads.



A load imbalance occurs because thread 0 has the greatest amount of calculation and thread 7 has the smallest processing amount.

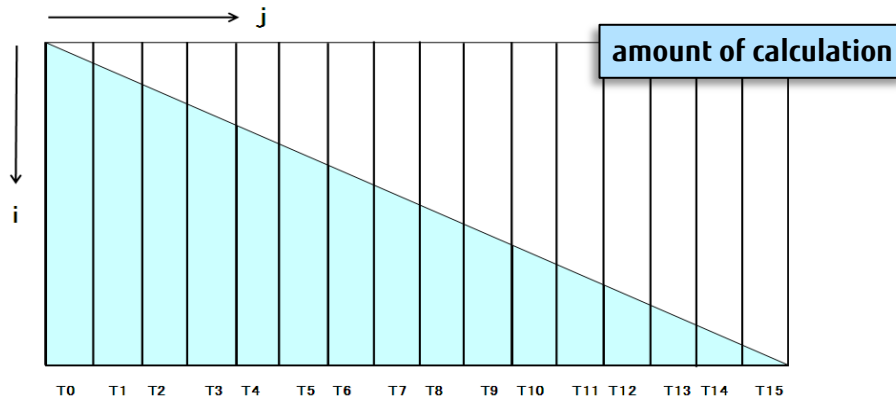
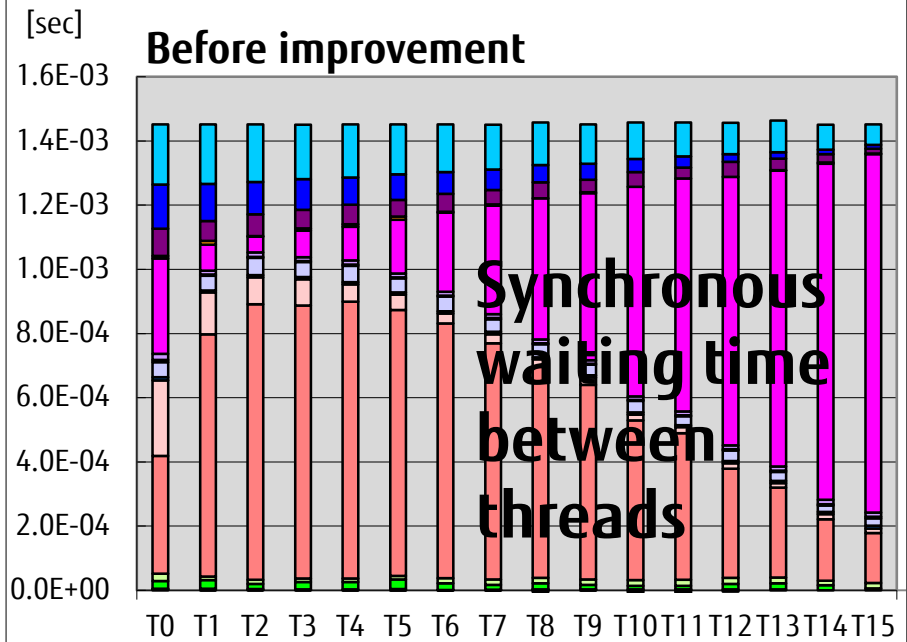
Triangular Loop (Before Improvement)

A load imbalance occurs because the amount of calculation varies for individual threads. Consequently, the following is a frequent event: Synchronous waiting time between threads.

Source code before improvement

```
28      subroutine sub()
29      integer*8 i,j,n
30      parameter(n=512)
31      real*8 a(n+1,n),b(n+1,n),c(n+1,n)
32      common a,b,c
33
34      !$omp parallel do
35 1 p      do j=1,n
    <<< Loop-information Start >>>
    <<< [OPTIMIZATION]
    <<< SIMD(VL: 4)
    <<< SOFTWARE PIPELINING
    <<< Loop-information End >>>
36 2 p 8v      do i=j,n
37 2 p 8v      a(i,j)=b(i,j)+c(i,j)
38 2 p 8v      enddo
39 1 p      enddo
40
41      end
```

Triangular loop



Poor load balance between different threads!

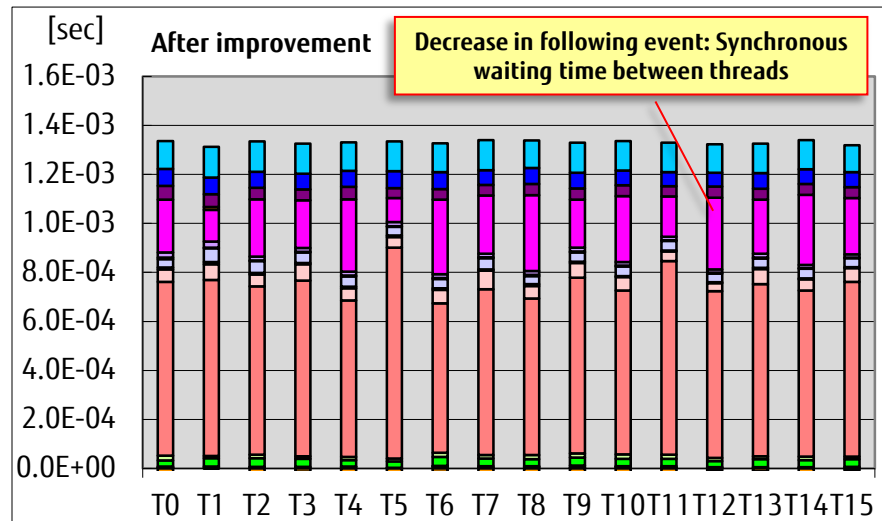
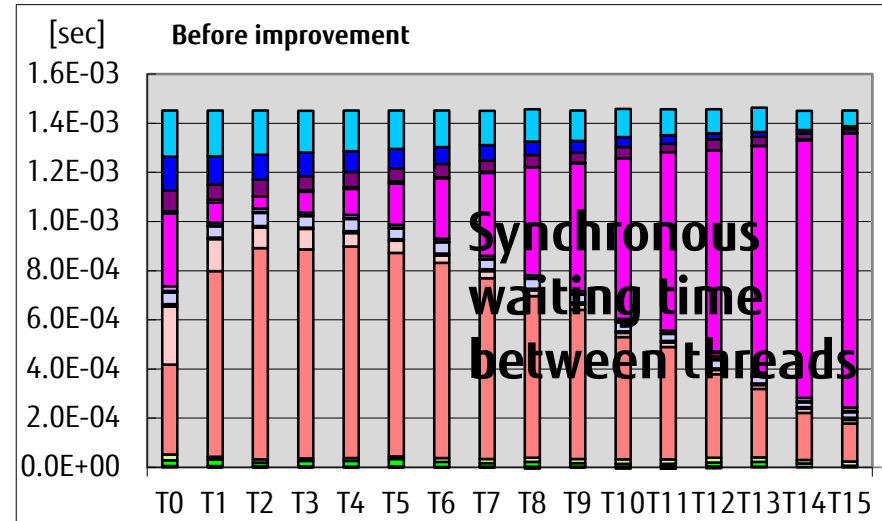
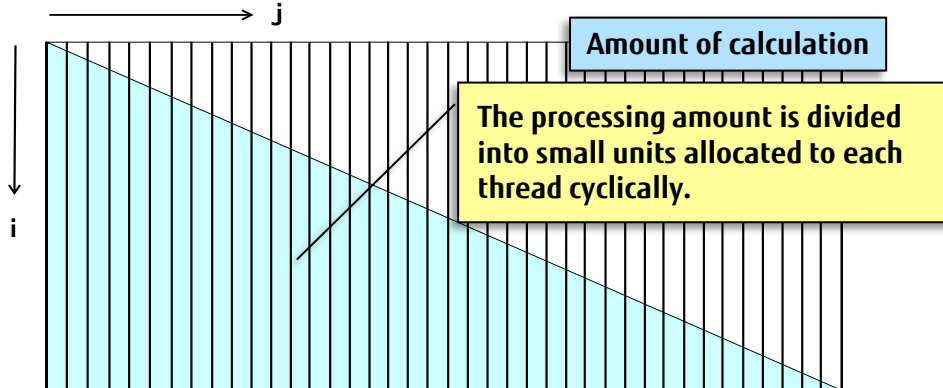
Triangular Loop (OpenMP Tuning)

After **the processing amount is divided into small units allocated cyclically**, the amount of calculation of each thread is uniform. The result is a load imbalance improvement and a decrease in the following event: Synchronous waiting time between threads.

Source code after improvement

```
28      subroutine sub()  
29      integer*8 i,j,n  
30      parameter(n=512)  
31      real*8 a(n+1,n),b(n+1,n),c(n+1,n)  
32      common a,b,c  
33  
34      !$omp parallel do schedule(static,1)  
35  1 p      do j=1,n  
36      <<< Loop-information Start >>>  
37      <<< [OPTIMIZATION]  
38      <<< SIMD (VL: 4)  
39      <<< SOFTWARE PIPELINING  
40      <<< Loop-information End >>>  
41      do i=j,n  
42      a(i,j)=b(i,j)+c(i,j)  
43      enddo  
44  enddo  
45  end
```

Triangular loop



Loops with Irregular Amount of Calculation

- Loop Containing an IF Construct (Before Improvement)
- Loop Containing an IF Construct (OpenMP Tuning)
- Loop with an Irregular Amount of Calculation (Before Improvement)
- Loop with an Irregular Amount of Calculation (OpenMP Tuning)

Loop Containing an IF Construct (Before Improvement)

The loop contains an IF construct. In this case, even if the amount of calculation varies between different threads, cyclic division with static specified as the scheduling method may not solve a load imbalance. Consequently, the following is a frequent event: Synchronous waiting time between threads.

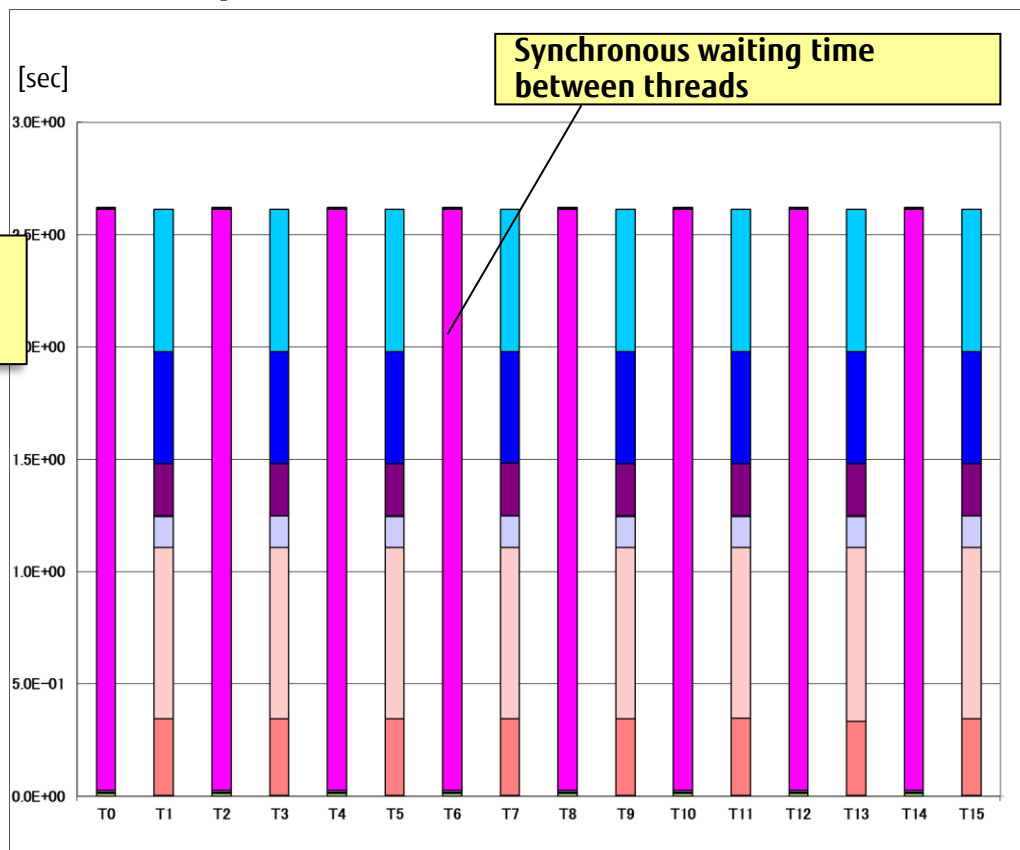
Source code before improvement

```
1  subroutine sub(a,b,s,n,m,nn)
2    real*8 a(m,n),b(m,n)
3    real s
4    !$omp parallel do schedule(static,1)
5  1 p    do k=1,nn
6  2 p      if( mod(k,2) .eq. 0 ) then
7  3 p        do j=1,n
8  4 p 8v          do i=1,m
9  4 p 8v            a(i,j) = a(i,j)*b(i,j)*s
10 4 p 8v          enddo
11 3 p        enddo
12 2 p      endif
13 1 p    enddo
14  end subroutine sub
```

Executes processing in
loop of only odd
threads

```
:
26  program main
27    parameter(n=100)
28    parameter(m=1000)
29    parameter(nn=1000000)
30    real*8 a(m,n),b(m,n)
31    call init(a,b,n,m)
32    call sub(a,b,2.0,n,m,nn)
```

Before improvement



Poor load balance between different threads!

Loop Containing an IF Construct (OpenMP Tuning)

Load imbalance improves with dynamic as the scheduling method, since it allows a thread that completed processing earlier to execute the next process.

Source code after improvement

```
1  subroutine sub(a,b,s,n,m,nn)
2    real*8 a(m,n),b(m,n)
3    real s
4    !$omp parallel do schedule(dynamic,1)
5  1 p  do k=1,nn
6  2 p  if( mod(k,2) .eq. 0 ) then
7  3 p  do j=1,n
```

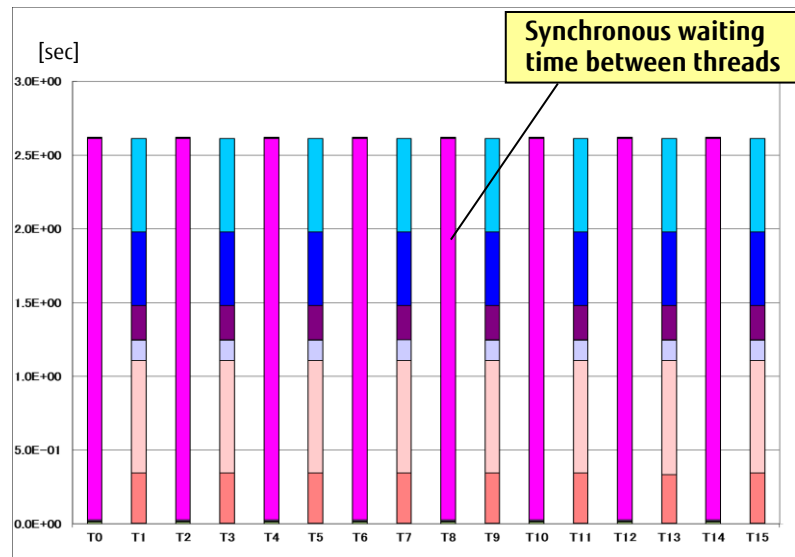
```
<<< Loop-information Start
<<< [OPTIMIZATION]
<<< SIMD(VL: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End
```

Specifying dynamic allows a thread that has completed processing earlier to execute the next process.

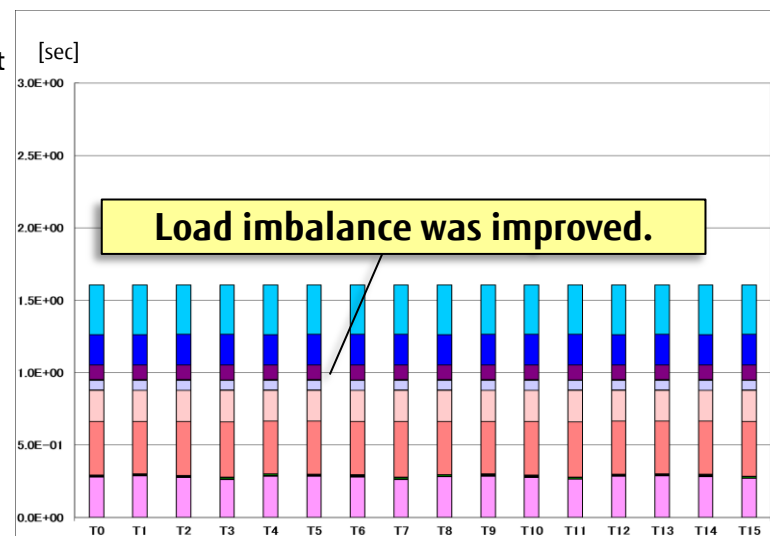
```
8  4 p 8v do i=1,m
9  4 p 8v a(i,j) = a(i,j)*b(i,j)*s
10 4 p 8v enddo
11 3 p enddo
12 2 p endif
13 1 p enddo
14 end subroutine sub
```

```
:
26 program main
27 parameter(n=100)
28 parameter(m=1000)
29 parameter(nn=1000000)
30 real*8 a(m,n),b(m,n)
31 call init(a,b,n,m)
32 call sub(a,b,2.0,n,m,nn)
33 end program main
```

Before improvement



After improvement



Loop with an Irregular Amount of Calculation(Before Improvement)

Even if the amount of calculation fluctuates irregularly, cyclic division with static specified as the scheduling method may not solve a load imbalance. Consequently, the following is a frequent event: Synchronous waiting time between threads.

Source code before improvement

```
1      subroutine init(a,b,ie,n)
2      :
3      <<< Loop-information Start >>>
4      <<< [OPTIMIZATION]
5      <<< FUSED
6      <<< Loop-information End >>>
7
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
```

A value is set in array **ie** as the evaluation loop end value only for an even number of iterations.

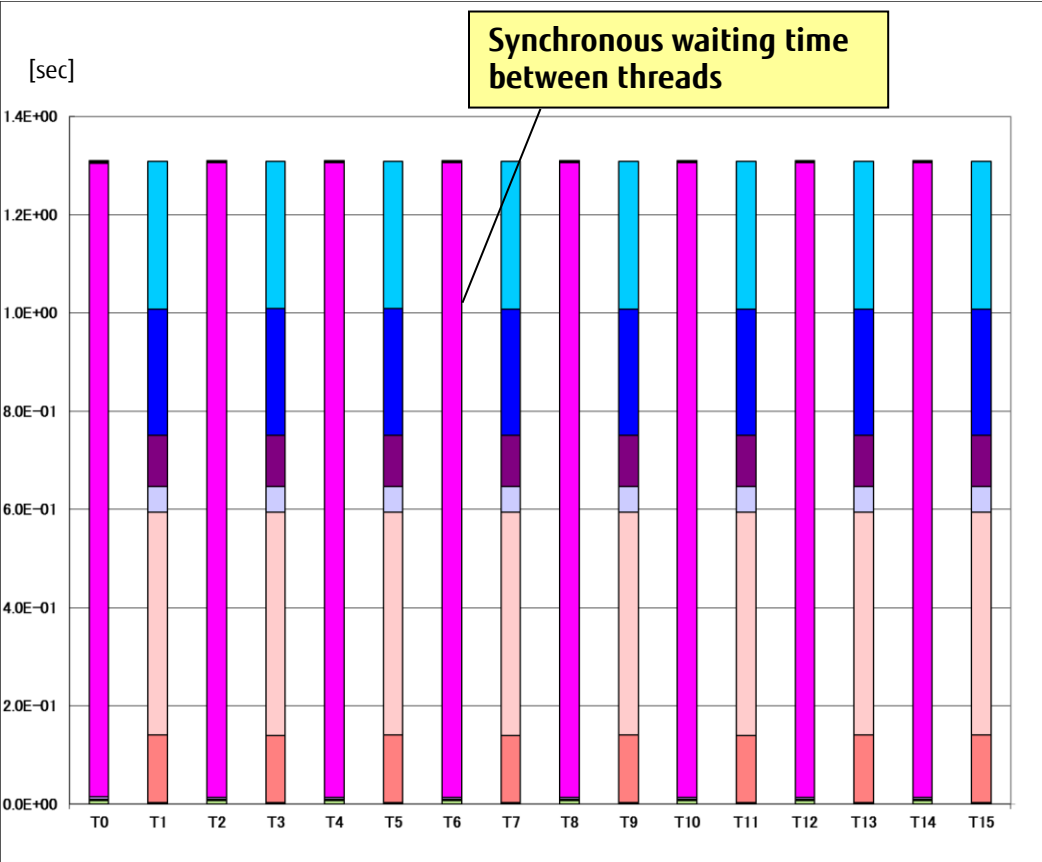
```
16      subroutine sub(a,b,s)
17      real a(n),b(n),s
18      integer ie(n)
19      !$omp parallel do schedule(static,1)
20 1 p      do j=1,n
21      <<< Loop-information Start >>>
22      <<< [OPTIMIZATION]
23      <<< SIMD(VL: 8)
24      <<< SOFTWARE PIPELINING
25      <<< Loop-information End >>>
26
27 2 p 8v      do i=1,ie(j)
28 2 p 8v          a(i) = a(i)*b(i)*s
29 2 p 8v      enddo
30 1 p      enddo
31      end subroutine sub
```

Evaluation loop

```
27      program main
28      parameter(n=1000000)
29      call init(a,b,ie,n)
30
31      call sub(a,b,2.0,ie,n)
32
33
```

The innermost loop has 100,000 iterations only when control variable **j** is an even number.

Before improvement



Poor load balance between different threads!

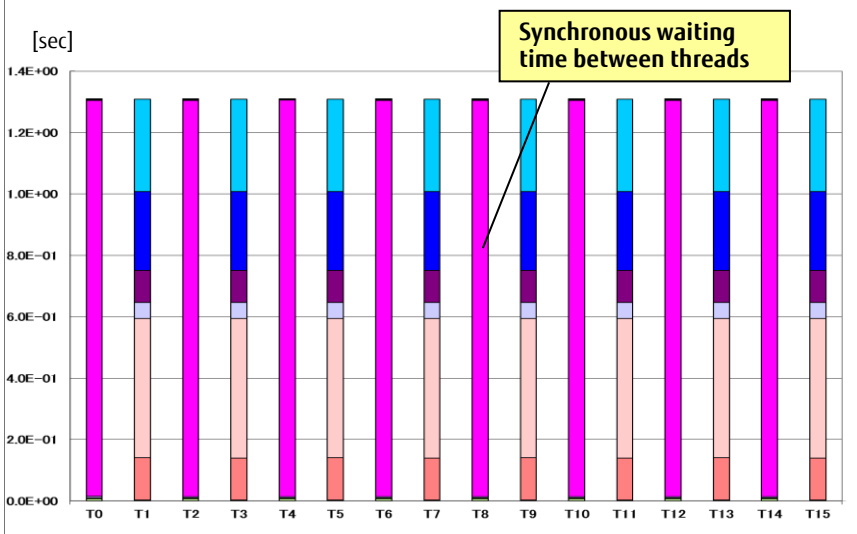
Loop with an Irregular Amount of Calculation (OpenMP Tuning)

Load imbalance improves with dynamic as the scheduling method, since it allows a thread that completed processing earlier to execute the next process.

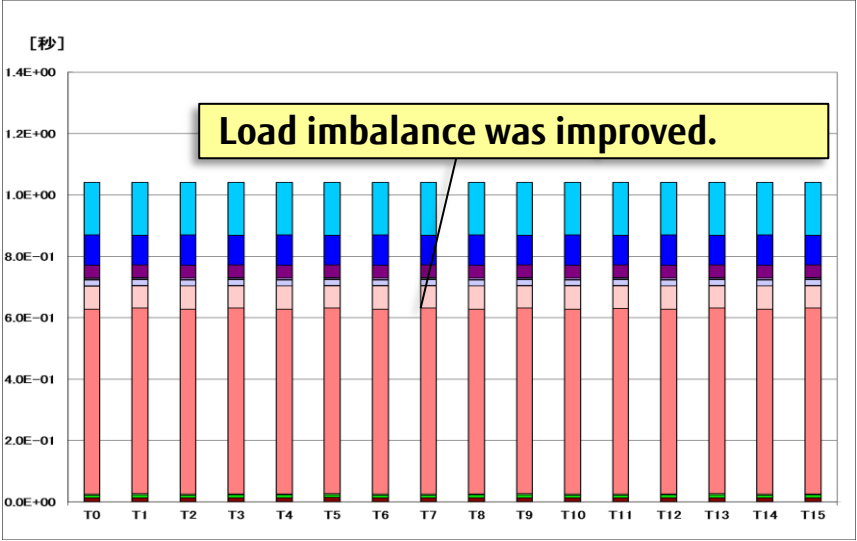
Source code after improvement

```
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: 8)
      <<< SOFTWARE PIPELINING
      <<< Loop-information End >>>
21 2 p 8v      do i=1,ie(j)
22 2 p 8v          a(i) = a(i)*b(i)*s
23 2 p 8v      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)
```

Before improvement



After improvement



Specifying dynamic allows a thread that has completed processing earlier to execute the next process.

Small Loop Iteration Count of a Parallelized Dimension

- Parallelization in an Appropriate Parallelized Dimension (Before Improvement)
- Parallelization in an Appropriate Parallelized Dimension (Optimization Control Line Tuning)
- Parallelization in an Appropriate Parallelized Dimension (Compiler Options Tuning)

If the loop iteration count of the parallelized dimension is small and unknown at the compile time, a load imbalance occurs under the following condition: the number of iterations is smaller than the number of thread parallelization processes (16 parallel processes in this example). Consequently, the following is a frequent event: Synchronous waiting time between threads.

Source code before improvement

```

<<< Loop-information Start >>>
<<< [PARALLELIZATION]
<<< Standard iteration count: 2
<<< Loop-information End >>>
34  1  pp      do k=1,l
35  2  p        do j=1,m
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< SIMD(VL: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End >>>
36  3  p 8v      do i=1,n
37  3  p 8v        a(i,j,k)=b(i,j,k)+c(i,j,k)
38  3  p 8v          enddo
39  2  p          enddo
40  1  p          enddo
41
42      end

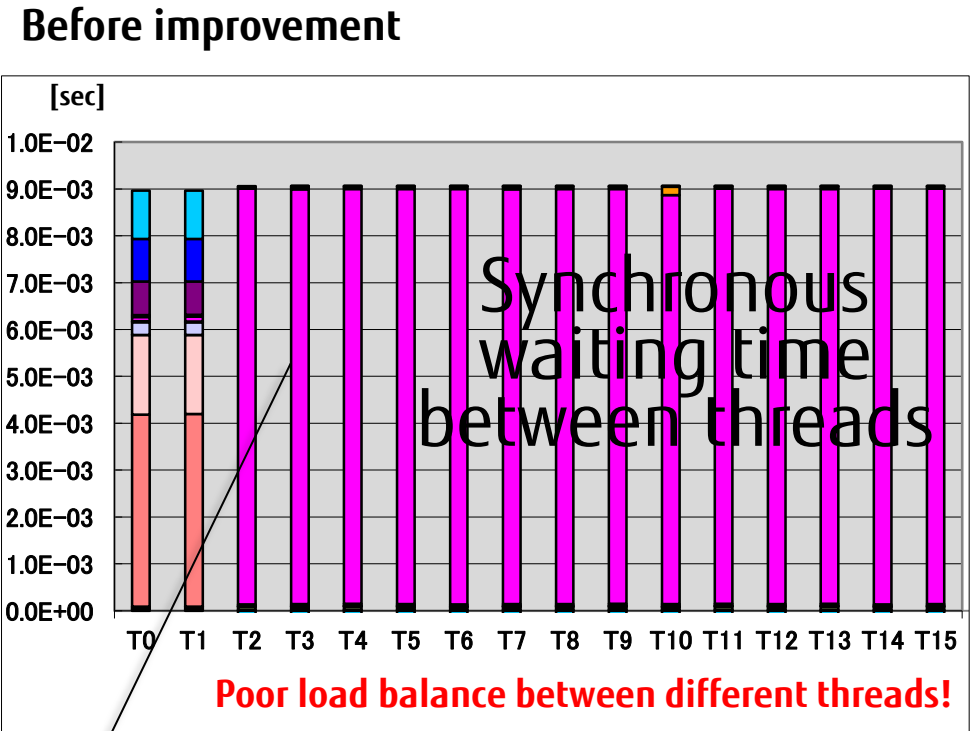
```

I=2

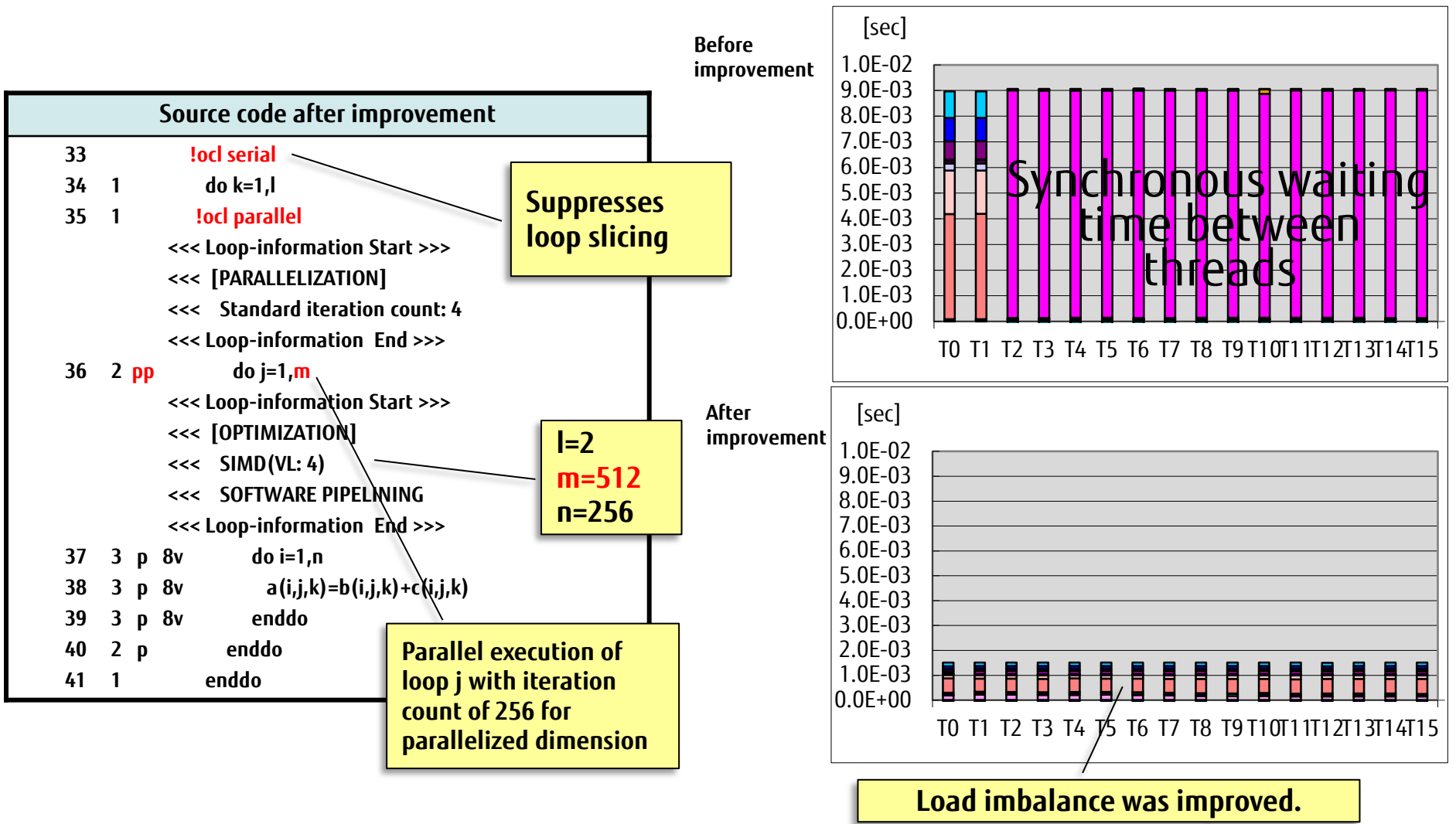
m=512

n=256

A load imbalance occurs because the number of iterations of k in the parallelized dimension is 2.



Specifying the **SERIAL and PARALLEL specifiers** realized parallelization in an appropriate dimension and improved load imbalance.



With **the compiler options -Kdynamic_iteration** specified, an appropriate parallelized dimension was automatically selected at the execution time, and load imbalance was improved.

Source code after improvement

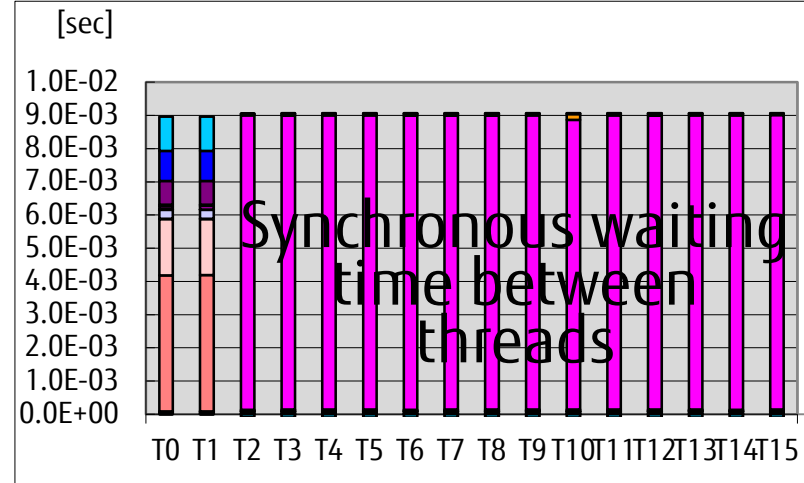
```

<<< Loop-information Start >>>
<<< [PARALLELIZATION]
<<< Standard iteration count: 2
<<< Loop-information End >>>
34  1 pp      do k=1,l
<<< Loop-information Start >>>
<<< [PARALLELIZATION]
<<< Standard iteration count: 4
<<< Loop-information End >>>
35  2 pp      do j=1,m
<<< Loop-information Start >>>
<<< [PARALLELIZATION]
<<< Standard iteration count: 728
<<< [OPTIMIZATION]
<<< SIMD(VL: 4)
<<< SOFTWARE PIPELINING
<<< Loop-information End >>>
36  3 pp 8v    do i=1,n
37  3 p 8v      a(i,j,k)=b(i,j,k)+c(i,j,k)
38  3 p 8v      enddo
39  2 p        enddo
40  1 p        enddo
41
42              end
    
```

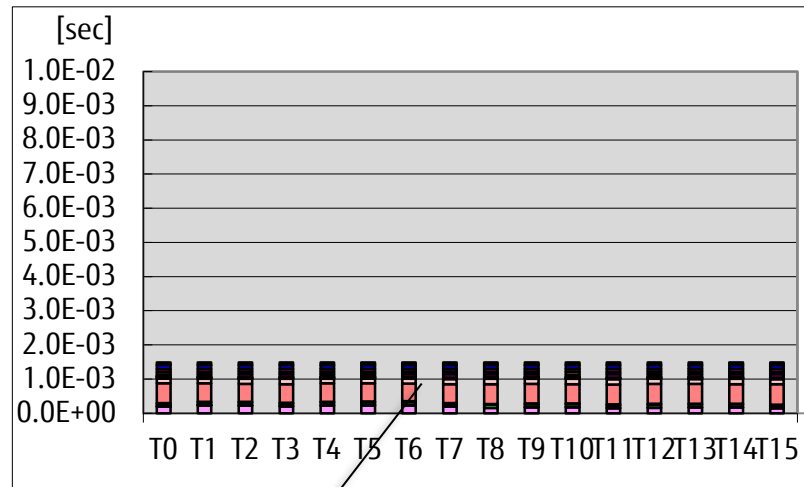
**l=2
m=512
n=256**

Parallel execution attempted from outer loop, but loop k has small number of iterations at 2, so inner loop j with 512 iterations is executed in parallel

Before improvement



After improvement



Load imbalance was improved.

Usage Taking SSL2 Library Performance into Account (DGEMM)

- DGEMM Parameters
- DGEMM Parameters Appropriate to the FX100

DGEMM Parameters

■ The following is a list of parameters for calling DGEMM.

■ $C := \text{ALPHA} \times \text{op}(A) \times \text{op}(B) + \text{BETA} \times C$

■ `DGEMM(TRANSA, TRANSB, M, N, K,
ALPHA, A, LDA B, LDB, BETA, C, LDC)`

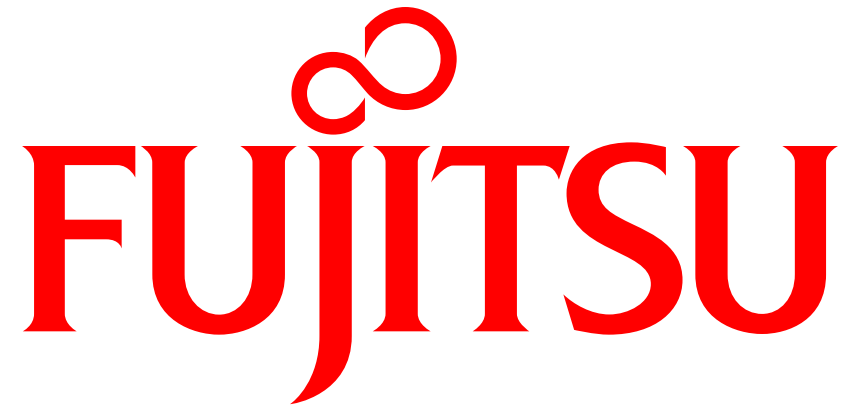
Argument	Meaning
TRANSA, TRANSB	They specify 'N' (do not transpose), 'T' (transpose), or 'C' (conjugate transpose).
M, N, K	Integers indicating the matrix size
ALPHA, BETA	Scalar values used in operation
A, B, C	A: M x K matrix B: K x N matrix C: M x N matrix
LDA, LDB, LDC	They specify the size of the 1st dimension of arrays A, B, and C, respectively.

DGEMM Parameters Appropriate to the FX100

- The recommended number of processes in a node is 2.
 - Performance is good with 16 or 32 threads, which enable utilization of a sector cache.
(This is because the sector cache can effectively use L2\$.)
- We recommend that M, N, and K be as large as possible.
That reduces the overhead of matrix copying done internally and the impact of the remaining part of a processing unit. Therefore, if they cannot be made larger by any means, try to improve efficiency as described below.
 - M should be a multiple of 32.
 - This is because the DGEMM kernel focuses on cases with processing in units of 32 elements (SIMD width of 4 x 8 registers) by combining 8 SIMD registers in the M direction.
 - N should be a multiple of 64 (when there are 16 threads).
 - This is because the DGEMM kernel focuses on cases with processing in units of 4 columns in the N direction.
 - If the size per thread is a multiple of 4 as a result of dividing N by the number of threads, the kernel is always used efficiently. However, if there is a remainder, efficiency decreases slightly.
 - K should be an even number.
 - This is because the DGEMM kernel focuses on cases with processing in units of 2 elements in the K direction.
- We recommend avoiding multiples of 2048 for LDA, LDB, and LDC.
 - This is because a multiple of 2048 may cause L1D\$ thrashing.

Revision History

Version	Date	Revised section	Details
2.0	April 25, 2016	-	- First published



shaping tomorrow with you