Transactional Memory for Fortran

IBM XL Fortran for Blue Gene/Q, V140 (technology preview)
## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1. Transactional memory</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2. Compiler option reference</td>
<td>3</td>
</tr>
<tr>
<td>-qtmm</td>
<td>3</td>
</tr>
<tr>
<td>Chapter 3. Environment variables for</td>
<td>5</td>
</tr>
<tr>
<td>transactional memory</td>
<td></td>
</tr>
<tr>
<td>TM_MAX_NUM_ROLLBACK</td>
<td>5</td>
</tr>
<tr>
<td>TM_REPORT_STAT_ENABLE</td>
<td>5</td>
</tr>
<tr>
<td>TM_REPORT_NAME</td>
<td>5</td>
</tr>
<tr>
<td>TM_REPORT_LOG</td>
<td>6</td>
</tr>
<tr>
<td>TM_ENABLE_INTERRUPT_ON_CONFLICT</td>
<td>6</td>
</tr>
<tr>
<td>Chapter 4. Compiler directives for</td>
<td>9</td>
</tr>
<tr>
<td>parallel processing</td>
<td></td>
</tr>
<tr>
<td>Chapter 5. Routines for transactional</td>
<td>13</td>
</tr>
<tr>
<td>memory</td>
<td></td>
</tr>
<tr>
<td>Nesting OpenMP, transactional memory, and</td>
<td>9</td>
</tr>
<tr>
<td>thread-level speculative execution</td>
<td></td>
</tr>
<tr>
<td>TM_ATOMIC / END TM_ATOMIC</td>
<td>10</td>
</tr>
<tr>
<td>Index</td>
<td>17</td>
</tr>
<tr>
<td>tm_get_stats(stats)</td>
<td>13</td>
</tr>
<tr>
<td>tm_get_all_stats(stats)</td>
<td>14</td>
</tr>
<tr>
<td>tm_print_stats()</td>
<td>15</td>
</tr>
<tr>
<td>reset_speculation_mode()</td>
<td>15</td>
</tr>
</tbody>
</table>
Chapter 1. Transactional memory

Transactional memory is a model for controlling concurrent memory accesses in the scope of parallel programming.

In parallel programming, concurrency control ensures that threads running in parallel do not update the same resources at the same time. Traditionally, the concurrency control of shared memory data is through locks, for example, mutex locks. A thread acquires a lock before modifying the shared data, and releases the lock afterward. A lock-based synchronization can lead to some performance issues because threads might need to wait to update lock-protected data.

Transactional memory is an alternative to lock-based synchronization. It attempts to simplify parallel programming by grouping read and write operations and running them like a single operation. Transactional memory is like database transactions where all shared memory accesses and their effects are either committed all together or discarded as a group. All threads can enter the critical region simultaneously. If there are conflicts in accessing the shared memory data, threads try accessing the shared memory data again or are stopped without updating the shared memory data. Therefore, transactional memory is also called a lock-free synchronization. Transactional memory can be a competitive alternative to lock-based synchronization.

A transactional memory system must hold the following properties across the entire execution of a concurrent program:

**Atomicity**
All speculative memory updates of a transaction are either committed or discarded as a unit.

**Consistency**
The memory operations of a transaction take place in order. Transactions are committed one transaction at a time.

**Isolation**
Memory updates are not visible outside of a transaction until the transaction commits data.

**Transactional memory on Blue Gene/Q**

On Blue Gene/Q, the transactional memory model is implemented in the hardware to access all the memory up to the 16 GB boundary.

Transactions are implemented through regions of code that you can designate to be single operations for the system. The regions of code that implement the transactions are called transactional atomic regions.

Transactional memory is enabled with the `-qtm` compiler option, and requires thread safe compilation mode.

**Execution modes**

When transactional memory is activated in Blue Gene/Q, transactions are run in one of the following operating modes:
• Speculation mode (default)
• Irrevocable mode

Each mode applies to an entire transactional atomic region.

Speculation mode
Under speculation mode, Kernel address space, devices I/Os, and most memory-mapped I/Os are protected from the irrevocable actions except when the `safe_mode` clause is specified. The transaction goes into irrevocable mode if such an action occurs to guarantee the correct result.

Irrevocable mode
System calls, irrevocable operations such as I/O operations, and OpenMP constructs trigger transactions to go into irrevocable mode, which serializes transactions. Transactions are also running in irrevocable mode when the maximum number of transaction rollbacks has been reached.

Under irrevocable mode, each memory update of a thread is committed instantaneously instead of at the end of the transaction. Therefore, memory updates are immediately visible to other threads. If the transaction becomes irrevocable, the threads run nonspeculatively.

Using variables and synchronization constructs with transactional memory

The semantics of transactional memory ensure that the effects of transactions of a thread are visible to other threads only after the transactions commit data or become irrevocable. When you use variables or synchronization constructs inside transactions, be careful when the volatile and regular variables that are visible to other threads are updated.

Data races when using transactional memory

A data race might happen if a memory location is accessed concurrently from both the following types of code sections:
• A transactional atomic region that is not nested in other critical sections
• A lock-based critical section of another thread

For example, the atomicity of a lock-based critical section might be broken when the transaction happens in the middle of the critical section. The atomicity of the transaction might also be broken if the transaction becomes irrevocable and is interleaved with the critical section.

The data race happens because each transactional atomic region can be thought of as using a different lock. In contrast, the `#pragma omp critical` directive uses one lock for all critical regions in the same parallel region.

Related information
• The `-qtm` compiler option
• Routines for transactional memory
• Environment variables for transactional memory
• `TM_ATOMIC / END TM_ATOMIC`
Chapter 2. Compiler option reference

-qtm

Category
Optimization and tuning

@PROCESS
None.

Purpose
Enables support for transactional memory.

Syntax

```
-q tm
```

Defaults
-qnotm

Usage
The -qtm option requires the thread safe compilation mode. Use -qtm with the bgxlf_r thread safe compiler invocation command.

Related information
- Transactional memory
- TM_ATOMiC
- Routines for transactional memory
- Environment variables for transactional memory
Chapter 3. Environment variables for transactional memory

The environment variables for transactional memory have no effect unless transactional memory is enabled with the 
“-qtm” compiler option.

**TM_MAX_NUM_ROLLBACK**

The TM_MAX_NUM_ROLLBACK environment variable indicates the maximum number a thread of a particular transactional atomic region can roll back before the thread goes into irrevocable mode.

```
10
```

The default value is 10.

TM_MAX_NUM_ROLLBACK = 0 forces the thread to enter irrevocable mode.

TM_MAX_NUM_ROLLBACK = -1 sets the thread to run infinite rollbacks.

**TM_REPORT_STAT_ENABLE**

The TM_REPORT_STAT_ENABLE environment variable enables or disables statistics query routines for transactional memory.

```
YES
```

The value is not case sensitive.

When TM_REPORT_STAT_ENABLE is set to YES, you can retrieve the statistics through the following routines:

- “tm_get_stats(stats)” on page 13
- “tm_get_all_stats(stats)” on page 14

**Related information**

- Routines for transactional memory

**TM_REPORT_NAME**

The TM_REPORT_NAME environment variable specifies the name and location of the statistics log file for transactional memory.

```
file_path
```
If TM_REPORT_NAME is not set, the log file is placed in the current working directory and is named \texttt{tm_report.log}. The extension \texttt{pid} is the ID of the process that called \texttt{tm_print_stats}.

**Related information**
- Routines for transactional memory

### TM_REPORT_LOG

The TM_REPORT_LOG environment variable specifies how to create the statistics log file for transactional memory.

By default, TM_REPORT_LOG is not defined and no log file is created.

\[
\begin{array}{c|c|c|c}
\text{TM_REPORT_LOG} & \text{SUMMARY} & \text{FUNC} & \text{ALL} \\
\hline
\text{SUMMARY} & & & \\
\text{FUNC} & & & \\
\text{ALL} & & & \\
\text{VERBOSE} & & & \\
\end{array}
\]

The value is not case sensitive.

- **SUMMARY**
  - The statistics log file is generated only at the end of the program.
- **FUNC**
  - The statistics log file is generated and updated at each call to the \texttt{tm_print_stats} routine.
- **ALL**
  - The statistics log file is generated and updated at each call to the \texttt{tm_print_stats} routine and at the end of the program.
- **VERBOSE**
  - The statistics log file is generated and updated at each call to the \texttt{tm_print_stats} routine and at the end of the program. The generated report file also includes the addresses of memory access conflicts during the speculation.

If TM_REPORT_LOG is set to either SUMMARY, ALL, or VERBOSE, a log file that contains the following statistics is also created when the program ends:

- It contains the cumulative statistics for transactional atomic regions that each hardware thread has run. The statistics follow the Structure of statistic counters.
- It contains a summary that shows the sum of all the statistic counters for all the hardware threads.

**Related information**
- Routines for transactional memory

### TM_ENABLE_INTERRUPT_ON_CONFLICT

The TM_ENABLE_INTERRUPT_ON_CONFLICT environment variable indicates whether to generate interrupts on conflicts.

\[
\begin{array}{c|c|c}
\text{TM_ENABLE_INTERRUPT_ON_CONFLICT} & \text{NO} & \text{YES} \\
\end{array}
\]
The default value is **NO**. The value is not case sensitive.

Conflict resolution usually happens at the end of transactions. Use this environment variable to generate interrupts for access conflicts in transactional atomic regions. This option is more useful for long running transactions because the conflict resolution logic immediately starts inside the interrupt handler of the kernel.

**Note:** Interrupts are always generated for speculative buffer overflows or supervised-mode violations.
Chapter 4. Compiler directives for parallel processing

You can use parallelization directives to exert control over parallelization.

- The OMP directives have effect only when parallelization is enabled with the -qsmp compiler option.
- The TM directives have effect only when transactional memory is enabled with the -qtm compiler option.
- The SPECULATIVE directives have effect only when thread-level speculative execution is enabled with the -qsmp=speculative compiler option.

Nesting OpenMP, transactional memory, and thread-level speculative execution

This section describes how you can mix parallel regions. The following types of parallel regions can be used without restrictions in the same program if they are not nested:

- OpenMP
- Transactional memory (TM)
- Thread-level speculative execution (SE)

They can also be nested but with some restrictions. The following table describes the behavior of different nesting scenarios.

Table 1. Nesting rules for OpenMP, TM, and SE

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Description</th>
<th>Runtime action</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN SE BEGIN SE END SE</td>
<td>An SE region is nested inside an SE region.</td>
<td>The SE nesting is flattened. The inner nested SE region is run speculatively by one thread as part of the parallel outer SE region. The clauses specified on the inner SE region are still effective.</td>
</tr>
<tr>
<td>BEGIN SE BEGIN TM END TM END SE</td>
<td>A TM region is nested inside an SE region.</td>
<td>The TM region is run speculatively in SE mode as part of the outer SE region.</td>
</tr>
<tr>
<td>BEGIN SE BEGIN OpenMP END OpenMP END SE</td>
<td>An OpenMP region is nested inside an SE region.</td>
<td>An OpenMP region running in parallel inside the speculative SE region causes the SE region to be stopped. The stopped SE region is rolled back and run nonspeculatively. The inner OpenMP region is run nonspeculatively by multiple threads.</td>
</tr>
<tr>
<td>BEGIN TM BEGIN SE END SE END TM</td>
<td>An SE region is nested inside a TM region.</td>
<td>The inner SE region is run speculatively in TM mode by one thread as part of the outer TM region.</td>
</tr>
<tr>
<td>BEGIN TM BEGIN TM END TM END TM</td>
<td>A TM region is nested inside a TM region.</td>
<td>The TM nesting is flattened. The inner nested TM regions are run speculatively as part of the outer TM region.</td>
</tr>
</tbody>
</table>
Table 1. Nesting rules for OpenMP, TM, and SE (continued)

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Description</th>
<th>Runtime action</th>
</tr>
</thead>
</table>
| BEGIN TM
  BEGIN OpenMP
  END OpenMP
  END TM | An OpenMP region is nested inside a TM region. | An OpenMP region running in parallel inside the speculative TM region causes the TM region to be stopped. The stopped transaction is then rolled back and run non-speculatively. The inner OpenMP region is run non-speculatively by multiple threads. |
| BEGIN OpenMP
  BEGIN SE
  END SE
  END OpenMP | An SE region is nested inside an OpenMP region. | The SE region is run by one thread in non-speculative mode. The outer OpenMP region is run in parallel. |
| BEGIN OpenMP
  BEGIN TM
  END TM
  END OpenMP | A TM region is nested inside an OpenMP region. | The outer OpenMP region is run in parallel and the TM region is run speculatively. |
| BEGIN TM
  END TM
  BEGIN SE
  END SE | A program contains separate TM and SE regions. | The first region is run in TM mode. The second region is run in SE mode by one thread. If the reset_speculation_mode routine is called after the first TM region, the second SE region is run in parallel speculatively. |
| BEGIN SE
  END SE
  BEGIN TM
  END TM | A program contains separate SE and TM regions. | The first region is run in SE mode. The second region is run in TM irreversible mode. If the reset_speculation_mode routine is called after the first SE region, the second TM region is run in parallel speculatively. |

**TM_ATOMIC / END TM_ATOMIC**

The **TM_ATOMIC** directive indicates a transactional atomic region.

**Syntax**

```fortran
!TM$ TM_ATOMIC
  [safe_mode]
!TM$ END TM_ATOMIC
```

**safe_mode**

Using the **safe_mode** clause reduces overhead and increases performance. However, if **safe_mode** is specified, irreversible actions are not checked at runtime. The run result is undefined if an irreversible action occurs during the execution.

**Usage**

The transactional memory directive is enabled with the `-qtm` compiler option.

This directive must be placed immediately before the code block of the transactional atomic region.
A transactional atomic region must be a structured block as defined in the OpenMP 3.0 specification. A structured block is one of the following constructs:

- A block of executable statements with a single entry at the top and a single exit at the bottom
- An OpenMP construct

Notes:

- Entry to the transactional atomic region cannot be the result of a branch.
- Branching out of a transactional atomic region is supported. For example, you can use STOP statements in a transactional atomic region.
- You can use a **GO TO** statement to transfer control within a transactional atomic region, but not into or out of a transactional atomic region.
- When you use the **END=**, **ERR=**, or **EOR=** I/O statement specifier for branching, specify a statement that is within the same transactional atomic region as the I/O statement.

Transactional atomic regions can be nested. However, the atomicity, consistency, and isolation properties are provided at the outermost transactional atomic region.

- Stopping an inner transaction causes the corresponding outer transaction to stop.
- An inner transaction does not commit data at the end of its transactional atomic region. Instead, the data of the inner transaction is committed later when the data of the corresponding outer transaction is committed.
- If a conflict occurs in a nested transaction, the thread is rolled back to the beginning of the outermost transaction.

Example

```
USE OMP_LIB

INTEGER :: i
INTEGER :: data_arr(100)

!$OMP PARALLEL DO
DO i = 1, 100

! Use the TM_ATOMIC directive to indicate a transactional atomic region.
!TM$ TM_ATOMIC
  data_arr(i) = i + 1
!TM$ END TM_ATOMIC

END DO
!$OMP END PARALLEL DO
```

Related information

- The **-qtm** compiler option
- Execution modes
- Routines for transactional memory
- Environment variables for transactional memory
- "STOP"
Chapter 5. Routines for transactional memory

To call the routines that are specific to transactional memory, you must use the speculation_util intrinsic module.

The routines for transactional memory have no effect unless transactional memory is enabled with the `-qtm` compiler option.

**Structure of statistic counters**

The following structure that contains the statistic counters can be used with the transactional memory routines. It is declared in the speculation_util intrinsic module.

```fortran
TYPE TmReport
  SEQUENCE
    ! Thread ID
    INTEGER(8) hwThreadId

    ! Total number of transactions
    INTEGER(8) totalTransactions

    ! Total number of rollbacks for transactional memory threads
    INTEGER(8) totalRollbacks

    ! Total number of serialization caused by JMV conflicts
    INTEGER(8) totalSerializedJMV

    ! Total number of serialization caused by TM_MAX_NUM_ROLLBACK reached
    INTEGER(8) totalSerializedMAXRB

    ! Total number of serialization caused by buffer overflow, hardware race, ! and concurrent regions of transactional memory and thread-level speculative ! execution in the same process
    INTEGER(8) totalSerializedOTHER
END TYPE
```

**tm_get_stats(stats)**

**Purpose**

With the `tm_get_stats` routine, you can retrieve statistical information for the transactional memory of a particular hardware thread in your program.

**Class**

Subroutine.

**Argument Type and Attributes**

`stats`  
An INTENT(OUT) TmReport variable.
Result Type and Attributes

None.

Result Value

None.

Usage

When the **tm_get_stats** routine is called, it updates the argument with cumulative statistics of all the transactional atomic regions that a particular hardware thread has run.

To use the **tm_get_stats** routine, you must set the `TMREPORT_STAT_ENABLE` environment variable to YES.

Related information

- Environment variables for transactional memory

---

**tm_get_all_stats(stats)**

**Purpose**

With the **tm_get_all_stats** routine, you can retrieve statistical information for the transactional memory of all hardware threads in a program.

**Class**

Subroutine.

**Argument Type and Attributes**

- **stats**
  
  An INTENT(OUT) TmReport variable.

**Result Type and Attributes**

None.

**Result Value**

None.

**Usage**

When the **tm_get_all_stats** routine is called, it updates the argument with cumulative statistics of all the transactional atomic regions that all hardware threads have run.

To use the **tm_get_all_stats** routine, you must set the `TMREPORT_STAT_ENABLE` environment variable to YES.

You must use this routine outside of parallel regions.
Related information

- Environment variables for transactional memory

---

**tm_print_stats()**

**Purpose**

With the `tm_print_stats` routine, you can write statistics for the transactional memory of a particular hardware thread to a log file.

**Class**

Subroutine.

**Example**

```fortran
!TM$ TM_ATOMIC
code-block
!TM$ END TM_ATOMIC

CALL TM_PRINT_STATS()
```

**Usage**

After a transactional atomic region, you can call the `tm_print_stats` routine to write cumulative statistics of all the transactional atomic regions that a particular hardware thread has run to a log file.

To enable the statistics log file to be generated at each call to the `tm_print_stats` routine, you must set the `TM_REPORT_LOG` variable to FUNC or ALL.

By default, the log file is named `tm_report.pid`, where `pid` is the ID of the process that called `tm_print_stats`. The log file is saved in the current working directory of the program that uses transactional memory. To override the defaults, you can specify the `TM_REPORT_NAME` environment variable.

**Note:** If you use this routine in a transactional atomic region, it causes the transaction running in irrevocable mode.

**Related information**

- Environment variables for transactional memory

---

**reset_speculation_mode()**

**Purpose**

With the `reset_speculation_mode` routine, you can switch mode in between thread-level speculative execution (SE) and transactional memory (TM) at runtime.

**Class**

Subroutine.
Example
...
!SEP$ SPECULATIVE DO
do-loop
!SEP$ END SPECULATIVE DO

CALL RESET_SPECULATION_MODE()

!TM$ TM_ATOMIC
code-block
!TM$ END TM_ATOMIC
...

Usage
You must use the reset_speculation_mode routine after a TM or SE region.

To switch the mode, make sure all the TM or SE executions before the call are completed; otherwise, it might result in undefined behavior.

Related information
• “Nesting OpenMP, transactional memory, and thread-level speculative execution” on page 9
Index

C
compiler directives
  TM_ATOMIC 10
compiler option
  -qtm 3

D
directives
  nesting 9

E
environment variables
  TM_ENABLE_INTERRUPT_ON_CONFLICT
  TM_MAX_NUM_ROLLBACK 5
  TM_REPORT_LOG 6
  TM_REPORT_NAME 5
  TM_REPORT_STAT_ENABLE 5

I
intrinsic procedures
  reset_speculation_mode 15
  tm_get_all_stats 14

T
transactional memory 1

intrinsic procedures (continued)
  tm_get_stats 13
  tm_print_stats 15

S
  safe mode 1
  supervised mode 1