xCalls Guide

Revision 0.2

1 About this document

This document explains how to use the *xCalls* API to enable I/O and system calls inside memory transactions.

1.1 Conventions and Symbols

The following symbols are used throughout this document.

- `$XCALLS_HOME` The directory where the *xCalls* library is located

1.2 Related information

Related documents include:

- *xCalls*: Safe I/O in Memory Transactions, Submitted to OSDI 2008, Work in Progress
- Intel C++ STM Compiler Prototype Edition 2.0 Language Extensions and User's Guide

2 Introduction

2.1 xCalls

The xCall interface enables executing I/O and system calls within memory transactions. Similar to database transactions, memory transactions allow a programmer focus on the logic of their program and let the system ensure that transactions are atomic and isolated. However, most transactional memory systems only support transactions within a user-mode process. When a transaction performs I/O or accesses kernel resources, the atomicity and isolation guarantees from the TM system do not apply to the kernel.

The xCall interface provides isolation and atomicity for kernel data objects with a new programming API for making system calls within transactions. Rather than making every system call transactional, the xCall API handles the common cases of file access, communication, and synchronization. xCalls provide isolation for kernel resources with sentinels, which are revocable lightweight user-level locks. A transaction acquires a sentinel exclusively when it accesses kernel resource, such as a file, through a system call. Competing threads must block until the transaction completes and release the sentinel. xCalls provide atomicity for system calls through a combination of *deferral*, delaying execution until the transaction commits, and *compensation*, calling back into the kernel to undo the side effects of a previous call. However, the xCall interface specifies *when* every call executes, so programmers are aware when the side effects of an xCall become visible. Finally, xCalls return errors asynchronously, after the transaction completes, when a deferred system call or compensation fails.

2.2 Availability

Currently the xCall interface is compatible with the Intel C/C++ software transactional memory (STM) compiler prototype edition 2.0. The source code of the library is available through the SVN repository:

Repository Root:	`/p/multifacet/projects/naxos/NAXOS_SVN_ROOT`
Project:	`xCalls/txc/v2/`

3 Build Process

3.1 Building Library

TxLib is built by invoking the SCons software construction tool in the `$XCALLS_HOME` directory.

```
% scons
```

The built process can be parameterized using the following set of arguments:

- `mode=[debug, release]` Builds either a library with debugging support or an optimized version for use in productivity environments.
- `linkage=[dynamic, static]` Builds either a dynamic shared library (.so) or a static archive (.a).
- `stats=[enable,disable]` Builds the library with profiling support.
- `test=[disable,enable]` Builds the library and then tests the library using the unit tests found under `$XCALLS_HOME`/tests

The first option is the default one.

3.2 Building Applications

Makefiles used to build applications using the xCalls library should be modified in accordance to the guidelines presented here.

3.2.1 Symbol definitions

The following symbols must be defined when invoking the compiler:

- `_GNU_SOURCE`
- `TXC_XCALLS_ENABLE`
- `TM_CALLABLE="__attribute__ ((tm_callable))"`
- `TM_WAIVER="__attribute__ ((tm_pure))"`

Invocation example

```
icc  -Qtm_enabled -D_GNU_SOURCE -DTXC_XCALLS_ENABLE
-DTM_CALLABLE="__attribute__ ((tm_callable))"
-DTM_WAIVER="__attribute__ ((tm_pure))"
```

3.2.2 Header files

The xCall interface is available through the following header file:

- `txc.h`

These files are available in `$XCALLS_HOME/include`

Invocation example

```
icc -I txc/include
```

Important Note The icc compiler by default uses built-in versions of common function such as `printf, memcpy`, etc called *intrinsic* functions. This also includes the transactional version of memory allocation routines (`malloc, realloc, calloc, free`). However, we have noticed that the compiler might generate incorrect code when using the built-in versions inside transactions. For example, we came across a case where the compiler would silently without any warning message inline `memset` inside a transaction but incorrectly not instrument the memory references produced by the inlining. To avoid such unexpected problems we recommend passing the `-fno-builtin` option to the compiler. However this has the side effect that the compiler does not use the transactional version of memory allocators. TxLib provides wrappers that use the internal transactional memory allocators. See section **??** for more information.

4 Programming Interface

4.1 TM Interface

The Transactional Memory (TM) interface provides wrapper macros and functions for defining and manipulating transactions which allow the safe invocation of *xCalls*. This interface is intended to be used both by the application and *xCalls*library programmer.

Wrapper Macros :

`XACT_BEGIN(tag)`	Begins a transaction block. The `tag` field is mandatory and it is used to name the transaction block.
`XACT_END(tag)`	Ends a transaction block
`XACT_RETRY`	Forces abort and restart of the current transaction.
`GET_TRANSACTION_MODE (xact_mode)`	Returns the transactional mode of the active thread (non-transactional, optimistic/pessimistic transaction, irrevocable transaction)
`XACT_SWITCH_TO_ IRREVOCABLE`	Forces the current transaction to execute in *xCalls* irrevocable mode.

Since `XACT_BEGIN` and `XACT_END` wrap the `__tm_atomic` keyword, they must be lexically scoped. For example the following is illegal:

```
__attribute__ ((tm_callable))
void goo() {
  ...
  XACT_END(xact_foo)
}

void foo() {
  XACT_BEGIN(xact_foo)
  goo();
}
```

Functions :

`void txc_global_init()` Initializes the TM and xCalls system. Must be called before using any of the xCalls in an application.

`void txc_thread_init()` Initializes the xCalls transaction descriptor associated with each thread in an application. A thread must call it before using any of the xCalls.

```
void txc_register_commit_action(
         txc_result_t (*action)(unsigned int, unsigned int *),
         unsigned int num_args,
         unsigned int *arg_list)
```

Registers function `action` to be called when the transaction commits. The function accepts a list of parameters in array form which is passed to function `action` when invoking it on commit. `action` accepts two arguments; the first is the length of the parameter list and the second the list in array form. *Note*: Currently the parameters can be only of integral type.

```
void txc_register_compensating_action(
         txc_result_t (*action)(unsigned int, unsigned int *),
          unsigned int num_args,
          unsigned int *arg_list)
```

Registers function `action` to be called when the transaction aborts. The function accepts a list of parameters in array form which is passed to function `action` when invoking it on abort.

Usage Example

```
#include <txc/transaction.h>
#include <txc/wrappers.h>

void child_main(void *arg)
{
  txc_global_init();

  XACT_BEGIN(xact_child)

  XACT_END(xact_child)
}

void main()
{
  txc_thread_init();

  pthread_create(&child, NULL, child_main, NULL);
}
```

4.2 Sentinels Interface

The sentinel interface provides macros for manipulating the sentinel abstraction. This interface is intended to be used by the *xCalls* library programmer.

`SENTINEL_ACQUIRE_EXCLUSIVE(sentinel)`	Acquires sentinel in exclusive mode.
`SENTINEL_ACQUIRE_SHARED(sentinel)`	Acquires sentinel in shared mode.
`SENTINEL_RELEASE(sentinel)`	Releases sentinel.

4.3 xCalls Interface

This interface is intended to be used by the application programmer. Its description is given in the document ”xCalls: Safe I/O in Memory Transactions”

4.3.1 Memory Allocation

When passing the `-fno-builtin` option to the compiler, the compiler does not use the transactional version of memory allocators. To resolve this issue, TxLib provides the wrappers `txc_malloc, txc_realloc, txc_calloc, txc_free` which use the internal transactional memory allocators provided by the STM.

5 Good Practice Guidelines

Here we present some *good practice guidelines* based on our experience working with the prototype edition of Intel's compiler (ICC) and the STM runtime.

1. Since macros XACT_BEGIN and XACT_END wrap the atomic construct, we need to ensure that the only exit point of the transaction is the XACT_END. For example in the following case return is replaced with a goto statement to ensure proper exit from the transaction block.

```
void foo() {
  XACT_BEGIN(xact1)
    ...
    if (X>0) {
      return;
    }
    ...
  XACT_END
}
```

   transformed into

```
void foo() {
  XACT_BEGIN(xact1)
    ...
    if (X>0) {
      goto xact1_end;
    }
    ...
xact1_end:
  XACT_END(xact1)
}
```

2. Use macro symbols TM_CALLABLE and TM_WAIVER instead of __attribute__ ((tm_callable)) and __attribute__ ((tm_pure)) since this provides independence from the compiler conventions which may unexpectedly change. In fact we have seen a change between prototype versions 1.0 and 2.0 in the way a TM waived function is defined; annotation tm_waiver has been replaced by tm_pure.

3. ICC provides the __tm_waiverblock construct to escape statements. For example in the following piece of code the compiler does not instrument the accesses to shared variables X and Y and does not generate any call for transition to irrevocable mode before printf is called.

```
XACT_BEGIN(xact1)
    ...
    __tm_waiver {
      X = Y + 1;
      printf("%d", X);
    }
    ...
XACT_END(xact1)
}
```

Note: It looks like the compiler is buggy and sometimes it does intrument statements wrappped inside a `__tm_waiver`block construct so bare this in mind.