

User Manual of the OpenTM Compiler and Runtime

Woongki Baek
Computer Systems Laboratory
Stanford University
{*wkbaek*}@*stanford.edu*

1 Introduction

This document provides basic information on how to set up and use the OpenTM compiler and runtime environment.

2 OpenTM Constructs and Runtime Routines

The following OpenTM constructs are supported by the current version of the OpenTM compiler. Future version of the OpenTM compiler will support more complete set of the OpenTM constructs discussed in [2].

Transaction Boundaries: OpenTM introduces the **transaction** construct to specify the boundaries of strongly isolated transactions. The syntax is:

```
#pragma omp transaction
```

```
structured-block
```

If the compiler is instructed to retarget the application code for software or hybrid TMs, the code enclosed by a transaction will be annotated with TM barriers. Currently, the OpenTM compiler generates unordered and closed-nested transactions with the **transaction** construct. Future version of the OpenTM compiler will support ordered transactions and open nesting. The definition of *structured-block* is the same as in OpenMP [1].

Transactional Functions: OpenTM introduces the **tm_function** construct to specify the *transactional functions* that might be called within transactions [8]. Once a function declaration is marked with the **tm_function** construct, the compiler generates a TM clone of the function that is annotated with TM barriers.

```

/*TM function declaration*/
#pragma omp tm_function
void foo(void);
...
foo();
...
...
/*Runtime checking*/
if (in_transaction())
    /*Call to TM clone*/
    ___TM___foo();
else
    foo();
...

```

Figure 1. Code generation for the `tm_function` construct.

```

#pragma omp tm_function
    function-declaration

```

Figure 1 illustrates how the compiler generates the code for the `tm_function` construct. At a call site of a function, the compiler generates the runtime checking code to decide whether the code is being executed within a transaction or not. If in a transaction, a TM clone of the function will be called. Otherwise, the original function will be called. Note that the compiler does not currently support calls to transactional functions using function pointers.

Runtime Routines: Currently, there are two OpenTM-specific user-level runtime routines.

- `omp_abort()`: The use of this runtime routine will abort the transaction executing the current code.
- `omp_in_transaction()`: The return value of this runtime routine indicates whether the code is being executed in a transaction or not.

3 Compiling the OpenTM Compiler and Runtime

The implementation of the OpenTM compiler is based on the GCC 4.3.0 (experimental version). The compiler and runtime have been tested on two combinations of an operating system and x86 processors so far: (1) CentOS 5 on Intel Pentium 4 processors and (2) CentOS 5 on AMD Opteron processors. The compilation procedure for the OpenTM compiler is identical to that of the GCC compiler. All the OpenTM compiler features are merged into the main compilation procedure of the GCC compiler. Hence, refer to [5] for more complete information on the compilation procedure.

Currently, the OpenTM runtime library is not merged into the main compilation procedure, so it should be compiled separately. The source files are primarily from the GNU OpenMP (GOMP) implementation [6] and reside

in `gcc/libotm` directory in the GCC source tree structure. `Makefile.stm` is used to generate the binary for real machines while `Makefile.*.sim` files are for the simulator. Programmers should link application code to the OpenTM runtime binary to generate the executables.

4 Using the OpenTM Environment

4.1 Downloading the Required Software Components

The following three software components are required to use the OpenTM environment.

- **OpenTM compiler and runtime:** The OpenTM compiler and runtime should be downloaded and compiled. Refer to Section 3 for further information.
- **TL2:** Currently, TL2 [4] is the only software TM system supported by the OpenTM environment on x86-based real machines. Refer to [7] for more detailed information on the x86 version of TL2.
- **STAMP benchmark suite:** STAMP is a new benchmark suite designed for TM research [7]. Currently, all of the eight STAMP applications are also available in the OpenTM version.

4.2 Specifying Compiler Options

To compile the OpenTM application code, users first enable the OpenMP features by using the `-fopenmp` compiler option because the OpenTM reuses all the features in the GOMP implementation. Besides, there are a few additional OpenTM-specific compiler options discussed below.

- `-ftl2:` Using this option instructs the OpenTM compiler to generate the compiled code to a software (e.g., TL2 [4]) or a hybrid (e.g., SigTM [3]) TM systems. More specifically, this option enables the TM annotation feature within the code enclosed by the `transaction` construct and TM cloning of functions using the `tm_function` construct. Future version of the OpenTM compiler will retarget the application code to more various TM systems. If this option is unspecified, the compiler assumes generates the code for hardware TM systems, so does not perform any TM annotation or cloning.
- `-fopentm-sim:` If this option is unspecified, the OpenTM compiler generates the code that runs on x86-based real machines.

The OpenTM compiler has been tested using the `-O3` option with the STAMP applications [7]. Through testings, we have identified the TM cloning feature sometimes does not work correctly with the function inlining-related optimizations (e.g., tail call elimination). Thus, disabling the function inlining feature is recommended using the `-fno-inline-functions` option.

In general, a good, typical example on specifying compiler options can be found in the `Makefile.stm.otm` in the `common` directory of the STAMP benchmark suite.

4.3 Linking

To generate an executable, users should link the application code to the compiled OpenTM runtime library. Again, a typical example on linking the OpenTM runtime library can be found in the `Defines.common.otm.mk` and `Makefile.stm.otm` files in the `common` directory of the STAMP benchmark suite.

4.4 Example: Compiling STAMP applications

Before compiling STAMP applications, the compiler and runtime of OpenTM and TL2 should be compiled. Since, so far, the OpenTM environment has been tested with the 32-bit mode on the x86 architecture, it is recommended to use the `-m32` option when users compile and link TL2 and STAMP applications even on 64-bit machines. For instance, users can use the `otm` rule in `Makefile` of TL2 to compile the TL2 code in the 32-bit mode, as below.

```
make otm
```

Users also need to modify `Defines.common.otm.mk` file in `common` directory of STAMP to modify the environment variables such as `OTM_DIR`, `OTM_COMPILER_DIR`, `LIBOTM_DIR`, and `LINUX_BIN_DIR`. The information provided in the `Defines.common.otm.mk` file is one example of a use case for a machine at Stanford. After this step, users can use `Makefile.stm.otm` in the directory of each application to compile it using the OpenTM environment. Refer to [7] for more detailed information on the STAMP benchmark suite itself.

4.5 Using Other STM Systems

Current implementation of OpenTM supports TL2 as the underlying STM system. However, there are a few possible ways to extend the compiler and runtime of OpenTM to work with other STM systems. First, users can

modify the OpenTM runtime routines such as `GOMP_transaction_start` or `omp_stm_shared_read` in `libotm/transaction.c` file. Currently, such routines are implemented to call the functions implemented in the TL2 STM library. By using the interface of other STM in those routines, users can use the compiler and runtime of OpenTM for that STM. Second, users can directly modify the compiler code to generate function calls directly using the interface of a target STM. While it can increase the implementation complexity and programming efforts, it can remove the overhead of calling the OpenTM runtime routines.

References

- [1] The OpenMP Application Program Interface Specification, version 2.5. <http://www.openmp.org>, May 2005.
- [2] W. Baek, C. Cao Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The OpenTM transactional application programming interface. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 376–387, Washington, DC, USA, 2007. IEEE Computer Society.
- [3] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. June 2007.
- [4] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC'06: Proceedings of the 20th International Symposium on Distributed Computing*, March 2006.
- [5] GCC, the GNU Compiler Collection. <http://gcc.gnu.org/>.
- [6] D. Novillo. Openmp and automatic parallelization in gcc. In *Proceedings of the 2006 GCC Developers' Summit*, pages 135–144, June 2006.
- [7] STAMP: Stanford transactional applications for multi-processing. <http://stamp.stanford.edu>.
- [8] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, March 2007.